tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt A L G O R I T H M S

I N T R O D U C T I O N T O

T H I R D E D I T I O N

T H O M A S H.

C H A R L E S E.

R O N A L D L.

C L I F F O R D S T E I N

R I V E S T

L E I S E R S O N

C O R M E N

Introduction to Algorithms

Third Edition

Thomas H. Cormen

Charles E. Leiserson

Ronald L. Rivest

Clifford Stein

Introduction to Algorithms

Third Edition

The MIT Press

Cambridge, Massachusetts London, England

c 2009 Massachusetts Institute of Technology

All rights reserved. No part of this book may be reproduced in any form or by any electronic or mechanical means

(including photocopying, recording, or information storage and retrieval) without permission in writing from the

publisher.

For information about special quantity discounts, please email special sales@mitpress.mit.edu.

This book was set in Times Roman and Mathtime Pro 2 by the authors.

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Introduction to algorithms / Thomas H. Cormen . . . [et al.].—3rd ed.

p. cm.

Includes bibliographical references and index.

ISBN 978-0-262-03384-8 (hardcover : alk. paper)—ISBN 978-0-262-53305-8 (pbk. : alk. paper)

1. Computer programming. 2. Computer algorithms. I. Cormen, Thomas H.

QA76.6.I5858 2009

005.1—dc22

2009008593

10 9 8 7 6 5 4 3 2

Contents

Preface xiii

I Foundations

Introduction 3

1 The Role of Algorithms in Computing 5

1.1 Algorithms 5

1.2 Algorithms as a technology 11

2 Getting Started 16

2.1 Insertion sort 16

2.2 Analyzing algorithms 23

2.3 Designing algorithms 29

3 Growth of Functions 43

3.1 Asymptotic notation 43

3.2 Standard notations and common functions 53

4 Divide-and-Conquer 65

4.1 The maximum-subarray problem 68

4.2 Strassen’s algorithm for matrix multiplication 75

4.3 The substitution method for solving recurrences 83

4.4 The recursion-tree method for solving recurrences 88

4.5 The master method for solving recurrences 93

? 4.6 Proof of the master theorem 97

5 Probabilistic Analysis and Randomized Algorithms 114

5.1 The hiring problem 114

5.2 Indicator random variables 118

5.3 Randomized algorithms 122

? 5.4 Probabilistic analysis and further uses of indicator random variables

130

vi Contents

II Sorting and Order Statistics

Introduction 147

6 Heapsort 151

6.1 Heaps 151

6.2 Maintaining the heap property 154

6.3 Building a heap 156

6.4 The heapsort algorithm 159

6.5 Priority queues 162

7 Quicksort 170

7.1 Description of quicksort 170

7.2 Performance of quicksort 174

7.3 A randomized version of quicksort 179

7.4 Analysis of quicksort 180

8 Sorting in Linear Time 191

8.1 Lower bounds for sorting 191

8.2 Counting sort 194

8.3 Radix sort 197

8.4 Bucket sort 200

9 Medians and Order Statistics 213

9.1 Minimum and maximum 214

9.2 Selection in expected linear time 215

9.3 Selection in worst-case linear time 220

III Data Structures

Introduction 229

10 Elementary Data Structures 232

10.1 Stacks and queues 232

10.2 Linked lists 236

10.3 Implementing pointers and objects 241

10.4 Representing rooted trees 246

11 Hash Tables 253

11.1 Direct-address tables 254

11.2 Hash tables 256

11.3 Hash functions 262

11.4 Open addressing 269

? 11.5 Perfect hashing 277

Contents vii

12 Binary Search Trees 286

12.1 What is a binary search tree? 286

12.2 Querying a binary search tree 289

12.3 Insertion and deletion 294

? 12.4 Randomly built binary search trees 299

13 Red-Black Trees 308

13.1 Properties of red-black trees 308

13.2 Rotations 312

13.3 Insertion 315

13.4 Deletion 323

14 Augmenting Data Structures 339

14.1 Dynamic order statistics 339

14.2 How to augment a data structure 345

14.3 Interval trees 348

IV Advanced Design and Analysis Techniques

Introduction 357

15 Dynamic Programming 359

15.1 Rod cutting 360

15.2 Matrix-chain multiplication 370

15.3 Elements of dynamic programming 378

15.4 Longest common subsequence 390

15.5 Optimal binary search trees 397

16 Greedy Algorithms 414

16.1 An activity-selection problem 415

16.2 Elements of the greedy strategy 423

16.3 Huffman codes 428

? 16.4 Matroids and greedy methods 437

? 16.5 A task-scheduling problem as a matroid 443

17 Amortized Analysis 451

17.1 Aggregate analysis 452

17.2 The accounting method 456

17.3 The potential method 459

17.4 Dynamic tables 463

viii Contents

V Advanced Data Structures

Introduction 481

18 B-Trees 484

18.1 Deﬁnition of B-trees 488

18.2 Basic operations on B-trees 491

18.3 Deleting a key from a B-tree 499

19 Fibonacci Heaps 505

19.1 Structure of Fibonacci heaps 507

19.2 Mergeable-heap operations 510

19.3 Decreasing a key and deleting a node 518

19.4 Bounding the maximum degree 523

20 van Emde Boas Trees 531

20.1 Preliminary approaches 532

20.2 A recursive structure 536

20.3 The van Emde Boas tree 545

21 Data Structures for Disjoint Sets 561

21.1 Disjoint-set operations 561

21.2 Linked-list representation of disjoint sets 564

21.3 Disjoint-set forests 568

? 21.4 Analysis of union by rank with path compression 573

VI Graph Algorithms

Introduction 587

22 Elementary Graph Algorithms 589

22.1 Representations of graphs 589

22.2 Breadth-ﬁrst search 594

22.3 Depth-ﬁrst search 603

22.4 Topological sort 612

22.5 Strongly connected components 615

23 Minimum Spanning Trees 624

23.1 Growing a minimum spanning tree 625

23.2 The algorithms of Kruskal and Prim 631

Contents ix

24 Single-Source Shortest Paths 643

24.1 The Bellman-Ford algorithm 651

24.2 Single-source shortest paths in directed acyclic graphs 655

24.3 Dijkstra’s algorithm 658

24.4 Difference constraints and shortest paths 664

24.5 Proofs of shortest-paths properties 671

25 All-Pairs Shortest Paths 684

25.1 Shortest paths and matrix multiplication 686

25.2 The Floyd-Warshall algorithm 693

25.3 Johnson’s algorithm for sparse graphs 700

26 Maximum Flow 708

26.1 Flow networks 709

26.2 The Ford-Fulkerson method 714

26.3 Maximum bipartite matching 732

? 26.4 Push-relabel algorithms 736

? 26.5 The relabel-to-front algorithm 748

VII Selected Topics

Introduction 769

27 Multithreaded Algorithms 772

27.1 The basics of dynamic multithreading 774

27.2 Multithreaded matrix multiplication 792

27.3 Multithreaded merge sort 797

28 Matrix Operations 813

28.1 Solving systems of linear equations 813

28.2 Inverting matrices 827

28.3 Symmetric positive-deﬁnite matrices and least-squares approximation

832

29 Linear Programming 843

29.1 Standard and slack forms 850

29.2 Formulating problems as linear programs 859

29.3 The simplex algorithm 864

29.4 Duality 879

29.5 The initial basic feasible solution 886

x Contents

30 Polynomials and the FFT 898

30.1 Representing polynomials 900

30.2 The DFT and FFT 906

30.3 Efﬁcient FFT implementations 915

31 Number-Theoretic Algorithms 926

31.1 Elementary number-theoretic notions 927

31.2 Greatest common divisor 933

31.3 Modular arithmetic 939

31.4 Solving modular linear equations 946

31.5 The Chinese remainder theorem 950

31.6 Powers of an element 954

31.7 The RSA public-key cryptosystem 958

? 31.8 Primality testing 965

? 31.9 Integer factorization 975

32 String Matching 985

32.1 The naive string-matching algorithm 988

32.2 The Rabin-Karp algorithm 990

32.3 String matching with ﬁnite automata 995

? 32.4 The Knuth-Morris-Pratt algorithm 1002

33 Computational Geometry 1014

33.1 Line-segment properties 1015

33.2 Determining whether any pair of segments intersects 1021

33.3 Finding the convex hull 1029

33.4 Finding the closest pair of points 1039

34 NP-Completeness 1048

34.1 Polynomial time 1053

34.2 Polynomial-time veriﬁcation 1061

34.3 NP-completeness and reducibility 1067

34.4 NP-completeness proofs 1078

34.5 NP-complete problems 1086

35 Approximation Algorithms 1106

35.1 The vertex-cover problem 1108

35.2 The traveling-salesman problem 1111

35.3 The set-covering problem 1117

35.4 Randomization and linear programming 1123

35.5 The subset-sum problem 1128

Contents xi

VIII Appendix: Mathematical Background

Introduction 1143

A Summations 1145

A.1 Summation formulas and properties 1145

A.2 Bounding summations 1149

B Sets, Etc. 1158

B.1 Sets 1158

B.2 Relations 1163

B.3 Functions 1166

B.4 Graphs 1168

B.5 Trees 1173

C Counting and Probability 1183

C.1 Counting 1183

C.2 Probability 1189

C.3 Discrete random variables 1196

C.4 The geometric and binomial distributions 1201

? C.5 The tails of the binomial distribution 1208

D Matrices 1217

D.1 Matrices and matrix operations 1217

D.2 Basic matrix properties 1222

Bibliography 1231

Index 1251

Preface

Before there were computers, there were algorithms. But now that there are com-

puters, there are even more algorithms, and algorithms lie at the heart of computing.

This book provides a comprehensive introduction to the modern study of com-

puter algorithms. It presents many algorithms and covers them in considerable

depth, yet makes their design and analysis accessible to all levels of readers. We

have tried to keep explanations elementary without sacriﬁcing depth of coverage

or mathematical rigor.

Each chapter presents an algorithm, a design technique, an application area, or a

related topic. Algorithms are described in English and in a pseudocode designed to

be readable by anyone who has done a little programming. The book contains 244

ﬁgures—many with multiple parts—illustrating how the algorithms work. Since

we emphasize efﬁciency as a design criterion, we include careful analyses of the

running times of all our algorithms.

The text is intended primarily for use in undergraduate or graduate courses in

algorithms or data structures. Because it discusses engineering issues in algorithm

design, as well as mathematical aspects, it is equally well suited for self-study by

technical professionals.

In this, the third edition, we have once again updated the entire book. The

changes cover a broad spectrum, including new chapters, revised pseudocode, and

a more active writing style.

To the teacher

We have designed this book to be both versatile and complete. You should ﬁnd it

useful for a variety of courses, from an undergraduate course in data structures up

through a graduate course in algorithms. Because we have provided considerably

more material than can ﬁt in a typical one-term course, you can consider this book

to be a “buffet” or “smorgasbord” from which you can pick and choose the material

that best supports the course you wish to teach.

xiv Preface

You should ﬁnd it easy to organize your course around just the chapters you

need. We have made chapters relatively self-contained, so that you need not worry

about an unexpected and unnecessary dependence of one chapter on another. Each

chapter presents the easier material ﬁrst and the more difﬁcult material later, with

section boundaries marking natural stopping points. In an undergraduate course,

you might use only the earlier sections from a chapter; in a graduate course, you

might cover the entire chapter.

We have included 957 exercises and 158 problems. Each section ends with exer-

cises, and each chapter ends with problems. The exercises are generally short ques-

tions that test basic mastery of the material. Some are simple self-check thought

exercises, whereas others are more substantial and are suitable as assigned home-

work. The problems are more elaborate case studies that often introduce new ma-

terial; they often consist of several questions that lead the student through the steps

required to arrive at a solution.

Departing from our practice in previous editions of this book, we have made

publicly available solutions to some, but by no means all, of the problems and ex-

ercises. Our Web site, http://mitpress.mit.edu/algorithms/, links to these solutions.

You will want to check this site to make sure that it does not contain the solution to

an exercise or problem that you plan to assign. We expect the set of solutions that

we post to grow slowly over time, so you will need to check it each time you teach

the course.

We have starred (?) the sections and exercises that are more suitable for graduate

students than for undergraduates. A starred section is not necessarily more difﬁ-

cult than an unstarred one, but it may require an understanding of more advanced

mathematics. Likewise, starred exercises may require an advanced background or

more than average creativity.

To the student

We hope that this textbook provides you with an enjoyable introduction to the

ﬁeld of algorithms. We have attempted to make every algorithm accessible and

interesting. To help you when you encounter unfamiliar or difﬁcult algorithms, we

describe each one in a step-by-step manner. We also provide careful explanations

of the mathematics needed to understand the analysis of the algorithms. If you

already have some familiarity with a topic, you will ﬁnd the chapters organized so

that you can skim introductory sections and proceed quickly to the more advanced

material.

This is a large book, and your class will probably cover only a portion of its

material. We have tried, however, to make this a book that will be useful to you

now as a course textbook and also later in your career as a mathematical desk

reference or an engineering handbook.

Preface xv

What are the prerequisites for reading this book?

You should have some programming experience. In particular, you should un-

derstand recursive procedures and simple data structures such as arrays and

linked lists.

You should have some facility with mathematical proofs, and especially proofs

by mathematical induction. A few portions of the book rely on some knowledge

of elementary calculus. Beyond that, Parts I and VIII of this book teach you all

the mathematical techniques you will need.

We have heard, loud and clear, the call to supply solutions to problems and

exercises. Our Web site, http://mitpress.mit.edu/algorithms/, links to solutions for

a few of the problems and exercises. Feel free to check your solutions against ours.

We ask, however, that you do not send your solutions to us.

To the professional

The wide range of topics in this book makes it an excellent handbook on algo-

rithms. Because each chapter is relatively self-contained, you can focus in on the

topics that most interest you.

Most of the algorithms we discuss have great practical utility. We therefore

address implementation concerns and other engineering issues. We often provide

practical alternatives to the few algorithms that are primarily of theoretical interest.

If you wish to implement any of the algorithms, you should ﬁnd the transla-

tion of our pseudocode into your favorite programming language to be a fairly

straightforward task. We have designed the pseudocode to present each algorithm

clearly and succinctly. Consequently, we do not address error-handling and other

software-engineering issues that require speciﬁc assumptions about your program-

ming environment. We attempt to present each algorithm simply and directly with-

out allowing the idiosyncrasies of a particular programming language to obscure

its essence.

We understand that if you are using this book outside of a course, then you

might be unable to check your solutions to problems and exercises against solutions

provided by an instructor. Our Web site, http://mitpress.mit.edu/algorithms/, links

to solutions for some of the problems and exercises so that you can check your

work. Please do not send your solutions to us.

To our colleagues

We have supplied an extensive bibliography and pointers to the current literature.

Each chapter ends with a set of chapter notes that give historical details and ref-

erences. The chapter notes do not provide a complete reference to the whole ﬁeld

xvi Preface

of algorithms, however. Though it may be hard to believe for a book of this size,

space constraints prevented us from including many interesting algorithms.

Despite myriad requests from students for solutions to problems and exercises,

we have chosen as a matter of policy not to supply references for problems and

exercises, to remove the temptation for students to look up a solution rather than to

ﬁnd it themselves.

Changes for the third edition

What has changed between the second and third editions of this book? The mag-

nitude of the changes is on a par with the changes between the ﬁrst and second

editions. As we said about the second-edition changes, depending on how you

look at it, the book changed either not much or quite a bit.

A quick look at the table of contents shows that most of the second-edition chap-

ters and sections appear in the third edition. We removed two chapters and one

section, but we have added three new chapters and two new sections apart from

these new chapters.

We kept the hybrid organization from the ﬁrst two editions. Rather than organiz-

ing chapters by only problem domains or according only to techniques, this book

has elements of both. It contains technique-based chapters on divide-and-conquer,

dynamic programming, greedy algorithms, amortized analysis, NP-Completeness,

and approximation algorithms. But it also has entire parts on sorting, on data

structures for dynamic sets, and on algorithms for graph problems. We ﬁnd that

although you need to know how to apply techniques for designing and analyzing al-

gorithms, problems seldom announce to you which techniques are most amenable

to solving them.

Here is a summary of the most signiﬁcant changes for the third edition:

We added new chapters on van Emde Boas trees and multithreaded algorithms,

and we have broken out material on matrix basics into its own appendix chapter.

We revised the chapter on recurrences to more broadly cover the divide-and-

conquer technique, and its ﬁrst two sections apply divide-and-conquer to solve

two problems. The second section of this chapter presents Strassen’s algorithm

for matrix multiplication, which we have moved from the chapter on matrix

operations.

We removed two chapters that were rarely taught: binomial heaps and sorting

networks. One key idea in the sorting networks chapter, the 0-1 principle, ap-

pears in this edition within Problem 8-7 as the 0-1 sorting lemma for compare-

exchange algorithms. The treatment of Fibonacci heaps no longer relies on

binomial heaps as a precursor.

Preface xvii

We revised our treatment of dynamic programming and greedy algorithms. Dy-

namic programming now leads off with a more interesting problem, rod cutting,

than the assembly-line scheduling problem from the second edition. Further-

more, we emphasize memoization a bit more than we did in the second edition,

and we introduce the notion of the subproblem graph as a way to understand

the running time of a dynamic-programming algorithm. In our opening exam-

ple of greedy algorithms, the activity-selection problem, we get to the greedy

algorithm more directly than we did in the second edition.

The way we delete a node from binary search trees (which includes red-black

trees) now guarantees that the node requested for deletion is the node that is

actually deleted. In the ﬁrst two editions, in certain cases, some other node

would be deleted, with its contents moving into the node passed to the deletion

procedure. With our new way to delete nodes, if other components of a program

maintain pointers to nodes in the tree, they will not mistakenly end up with stale

pointers to nodes that have been deleted.

The material on ﬂow networks now bases ﬂows entirely on edges. This ap-

proach is more intuitive than the net ﬂow used in the ﬁrst two editions.

With the material on matrix basics and Strassen’s algorithm moved to other

chapters, the chapter on matrix operations is smaller than in the second edition.

We have modiﬁed our treatment of the Knuth-Morris-Pratt string-matching al-

gorithm.

We corrected several errors. Most of these errors were posted on our Web site

of second-edition errata, but a few were not.

Based on many requests, we changed the syntax (as it were) of our pseudocode.

We now use “D” to indicate assignment and “==” to test for equality, just as C,

C++, Java, and Python do. Likewise, we have eliminated the keywords do and

then and adopted “//” as our comment-to-end-of-line symbol. We also now use

dot-notation to indicate object attributes. Our pseudocode remains procedural,

rather than object-oriented. In other words, rather than running methods on

objects, we simply call procedures, passing objects as parameters.

We added 100 new exercises and 28 new problems. We also updated many

bibliography entries and added several new ones.

Finally, we went through the entire book and rewrote sentences, paragraphs,

and sections to make the writing clearer and more active.

xviii Preface

Web site

You can use our Web site, http://mitpress.mit.edu/algorithms/, to obtain supple-

mentary information and to communicate with us. The Web site links to a list of

known errors, solutions to selected exercises and problems, and (of course) a list

explaining the corny professor jokes, as well as other content that we might add.

The Web site also tells you how to report errors or make suggestions.

How we produced this book

Like the second edition, the third edition was produced in L A T

E

X 2". We used the

Times font with mathematics typeset using the MathTime Pro 2 fonts. We thank

Michael Spivak from Publish or Perish, Inc., Lance Carnes from Personal TeX,

Inc., and Tim Tregubov from Dartmouth College for technical support. As in the

previous two editions, we compiled the index using Windex, a C program that we

wrote, and the bibliography was produced with BIBTEX. The PDF ﬁles for this

book were created on a MacBook running OS 10.5.

We drew the illustrations for the third edition using MacDraw Pro, with some

of the mathematical expressions in illustrations laid in with the psfrag package

for L A TEX 2". Unfortunately, MacDraw Pro is legacy software, having not been

marketed for over a decade now. Happily, we still have a couple of Macintoshes

that can run the Classic environment under OS 10.4, and hence they can run Mac-

Draw Pro—mostly. Even under the Classic environment, we ﬁnd MacDraw Pro to

be far easier to use than any other drawing software for the types of illustrations

that accompany computer-science text, and it produces beautiful output. 1 Who

knows how long our pre-Intel Macs will continue to run, so if anyone from Apple

is listening: Please create an OS X-compatible version of MacDraw Pro!

Acknowledgments for the third edition

We have been working with the MIT Press for over two decades now, and what a

terriﬁc relationship it has been! We thank Ellen Faran, Bob Prior, Ada Brunstein,

and Mary Reilly for their help and support.

We were geographically distributed while producing the third edition, working

in the Dartmouth College Department of Computer Science, the MIT Computer

1

We investigated several drawing programs that run under Mac OS X, but all had signiﬁcant short-

comings compared with MacDraw Pro. We brieﬂy attempted to produce the illustrations for this

book with a different, well known drawing program. We found that it took at least ﬁve times as long

to produce each illustration as it took with MacDraw Pro, and the resulting illustrations did not look

as good. Hence the decision to revert to MacDraw Pro running on older Macintoshes.

Preface xix

Science and Artiﬁcial Intelligence Laboratory, and the Columbia University De-

partment of Industrial Engineering and Operations Research. We thank our re-

spective universities and colleagues for providing such supportive and stimulating

environments.

Julie Sussman, P.P.A., once again bailed us out as the technical copyeditor. Time

and again, we were amazed at the errors that eluded us, but that Julie caught. She

also helped us improve our presentation in several places. If there is a Hall of Fame

for technical copyeditors, Julie is a sure-ﬁre, ﬁrst-ballot inductee. She is nothing

short of phenomenal. Thank you, thank you, thank you, Julie! Priya Natarajan also

found some errors that we were able to correct before this book went to press. Any

errors that remain (and undoubtedly, some do) are the responsibility of the authors

(and probably were inserted after Julie read the material).

The treatment for van Emde Boas trees derives from Erik Demaine’s notes,

which were in turn inﬂuenced by Michael Bender. We also incorporated ideas

from Javed Aslam, Bradley Kuszmaul, and Hui Zha into this edition.

The chapter on multithreading was based on notes originally written jointly with

Harald Prokop. The material was inﬂuenced by several others working on the Cilk

project at MIT, including Bradley Kuszmaul and Matteo Frigo. The design of the

multithreaded pseudocode took its inspiration from the MIT Cilk extensions to C

and by Cilk Arts’s Cilk++ extensions to C++.

We also thank the many readers of the ﬁrst and second editions who reported

errors or submitted suggestions for how to improve this book. We corrected all the

bona ﬁde errors that were reported, and we incorporated as many suggestions as

we could. We rejoice that the number of such contributors has grown so great that

we must regret that it has become impractical to list them all.

Finally, we thank our wives—Nicole Cormen, Wendy Leiserson, Gail Rivest,

and Rebecca Ivry—and our children—Ricky, Will, Debby, and Katie Leiserson;

Alex and Christopher Rivest; and Molly, Noah, and Benjamin Stein—for their love

and support while we prepared this book. The patience and encouragement of our

families made this project possible. We affectionately dedicate this book to them.

THOMAS H. CORMEN Lebanon, New Hampshire

CHARLES E. LEISERSON Cambridge, Massachusetts

RONALD L. RIVEST Cambridge, Massachusetts

CLIFFORD STEIN New York, New York

February 2009

Introduction to Algorithms

Third Edition

I Foundations

Introduction

This part will start you thinking about designing and analyzing algorithms. It is

intended to be a gentle introduction to how we specify algorithms, some of the

design strategies we will use throughout this book, and many of the fundamental

ideas used in algorithm analysis. Later parts of this book will build upon this base.

Chapter 1 provides an overview of algorithms and their place in modern com-

puting systems. This chapter deﬁnes what an algorithm is and lists some examples.

It also makes a case that we should consider algorithms as a technology, along-

side technologies such as fast hardware, graphical user interfaces, object-oriented

systems, and networks.

In Chapter 2, we see our ﬁrst algorithms, which solve the problem of sorting

a sequence of n numbers. They are written in a pseudocode which, although not

directly translatable to any conventional programming language, conveys the struc-

ture of the algorithm clearly enough that you should be able to implement it in the

language of your choice. The sorting algorithms we examine are insertion sort,

which uses an incremental approach, and merge sort, which uses a recursive tech-

nique known as “divide-and-conquer.” Although the time each requires increases

with the value of n, the rate of increase differs between the two algorithms. We

determine these running times in Chapter 2, and we develop a useful notation to

express them.

Chapter 3 precisely deﬁnes this notation, which we call asymptotic notation. It

starts by deﬁning several asymptotic notations, which we use for bounding algo-

rithm running times from above and/or below. The rest of Chapter 3 is primarily

a presentation of mathematical notation, more to ensure that your use of notation

matches that in this book than to teach you new mathematical concepts.

4 Part I Foundations

Chapter 4 delves further into the divide-and-conquer method introduced in

Chapter 2. It provides additional examples of divide-and-conquer algorithms, in-

cluding Strassen’s surprising method for multiplying two square matrices. Chap-

ter 4 contains methods for solving recurrences, which are useful for describing

the running times of recursive algorithms. One powerful technique is the “mas-

ter method,” which we often use to solve recurrences that arise from divide-and-

conquer algorithms. Although much of Chapter 4 is devoted to proving the cor-

rectness of the master method, you may skip this proof yet still employ the master

method.

Chapter 5 introduces probabilistic analysis and randomized algorithms. We typ-

ically use probabilistic analysis to determine the running time of an algorithm in

cases in which, due to the presence of an inherent probability distribution, the

running time may differ on different inputs of the same size. In some cases, we

assume that the inputs conform to a known probability distribution, so that we are

averaging the running time over all possible inputs. In other cases, the probability

distribution comes not from the inputs but from random choices made during the

course of the algorithm. An algorithm whose behavior is determined not only by its

input but by the values produced by a random-number generator is a randomized

algorithm. We can use randomized algorithms to enforce a probability distribution

on the inputs—thereby ensuring that no particular input always causes poor perfor-

mance—or even to bound the error rate of algorithms that are allowed to produce

incorrect results on a limited basis.

Appendices A–D contain other mathematical material that you will ﬁnd helpful

as you read this book. You are likely to have seen much of the material in the

appendix chapters before having read this book (although the speciﬁc deﬁnitions

and notational conventions we use may differ in some cases from what you have

seen in the past), and so you should think of the Appendices as reference material.

On the other hand, you probably have not already seen most of the material in

Part I. All the chapters in Part I and the Appendices are written with a tutorial

ﬂavor.

1 The Role of Algorithms in Computing

What are algorithms? Why is the study of algorithms worthwhile? What is the role

of algorithms relative to other technologies used in computers? In this chapter, we

will answer these questions.

1.1 Algorithms

Informally, an algorithm is any well-deﬁned computational procedure that takes

some value, or set of values, as input and produces some value, or set of values, as

output. An algorithm is thus a sequence of computational steps that transform the

input into the output.

We can also view an algorithm as a tool for solving a well-speciﬁed computa-

tional problem. The statement of the problem speciﬁes in general terms the desired

input/output relationship. The algorithm describes a speciﬁc computational proce-

dure for achieving that input/output relationship.

For example, we might need to sort a sequence of numbers into nondecreasing

order. This problem arises frequently in practice and provides fertile ground for

introducing many standard design techniques and analysis tools. Here is how we

formally deﬁne the sorting problem:

Input: A sequence of n numbers ha 1 ; a 2 ; : : : ; a n i.

Output: A permutation (reordering) ha 0

1

; a 0

2

; : : : ; a 0

n

i of the input sequence such

that a 0

1

a 0

2

a 0

n

.

For example, given the input sequence h31; 41; 59; 26; 41; 58i, a sorting algorithm

returns as output the sequence h26; 31; 41; 41; 58; 59i. Such an input sequence is

called an instance of the sorting problem. In general, an instance of a problem

consists of the input (satisfying whatever constraints are imposed in the problem

statement) needed to compute a solution to the problem.

6 Chapter 1 The Role of Algorithms in Computing

Because many programs use it as an intermediate step, sorting is a fundamental

operation in computer science. As a result, we have a large number of good sorting

algorithms at our disposal. Which algorithm is best for a given application depends

on—among other factors—the number of items to be sorted, the extent to which

the items are already somewhat sorted, possible restrictions on the item values,

the architecture of the computer, and the kind of storage devices to be used: main

memory, disks, or even tapes.

An algorithm is said to be correct if, for every input instance, it halts with the

correct output. We say that a correct algorithm solves the given computational

problem. An incorrect algorithm might not halt at all on some input instances, or it

might halt with an incorrect answer. Contrary to what you might expect, incorrect

algorithms can sometimes be useful, if we can control their error rate. We shall see

an example of an algorithm with a controllable error rate in Chapter 31 when we

study algorithms for ﬁnding large prime numbers. Ordinarily, however, we shall

be concerned only with correct algorithms.

An algorithm can be speciﬁed in English, as a computer program, or even as

a hardware design. The only requirement is that the speciﬁcation must provide a

precise description of the computational procedure to be followed.

What kinds of problems are solved by algorithms?

Sorting is by no means the only computational problem for which algorithms have

been developed. (You probably suspected as much when you saw the size of this

book.) Practical applications of algorithms are ubiquitous and include the follow-

ing examples:

The Human Genome Project has made great progress toward the goals of iden-

tifying all the 100,000 genes in human DNA, determining the sequences of the

3 billion chemical base pairs that make up human DNA, storing this informa-

tion in databases, and developing tools for data analysis. Each of these steps

requires sophisticated algorithms. Although the solutions to the various prob-

lems involved are beyond the scope of this book, many methods to solve these

biological problems use ideas from several of the chapters in this book, thereby

enabling scientists to accomplish tasks while using resources efﬁciently. The

savings are in time, both human and machine, and in money, as more informa-

tion can be extracted from laboratory techniques.

The Internet enables people all around the world to quickly access and retrieve

large amounts of information. With the aid of clever algorithms, sites on the

Internet are able to manage and manipulate this large volume of data. Examples

of problems that make essential use of algorithms include ﬁnding good routes

on which the data will travel (techniques for solving such problems appear in

1.1 Algorithms 7

Chapter 24), and using a search engine to quickly ﬁnd pages on which particular

information resides (related techniques are in Chapters 11 and 32).

Electronic commerce enables goods and services to be negotiated and ex-

changed electronically, and it depends on the privacy of personal informa-

tion such as credit card numbers, passwords, and bank statements. The core

technologies used in electronic commerce include public-key cryptography and

digital signatures (covered in Chapter 31), which are based on numerical algo-

rithms and number theory.

Manufacturing and other commercial enterprises often need to allocate scarce

resources in the most beneﬁcial way. An oil company may wish to know where

to place its wells in order to maximize its expected proﬁt. A political candidate

may want to determine where to spend money buying campaign advertising in

order to maximize the chances of winning an election. An airline may wish

to assign crews to ﬂights in the least expensive way possible, making sure that

each ﬂight is covered and that government regulations regarding crew schedul-

ing are met. An Internet service provider may wish to determine where to place

additional resources in order to serve its customers more effectively. All of

these are examples of problems that can be solved using linear programming,

which we shall study in Chapter 29.

Although some of the details of these examples are beyond the scope of this

book, we do give underlying techniques that apply to these problems and problem

areas. We also show how to solve many speciﬁc problems, including the following:

We are given a road map on which the distance between each pair of adjacent

intersections is marked, and we wish to determine the shortest route from one

intersection to another. The number of possible routes can be huge, even if we

disallow routes that cross over themselves. How do we choose which of all

possible routes is the shortest? Here, we model the road map (which is itself

a model of the actual roads) as a graph (which we will meet in Part VI and

Appendix B), and we wish to ﬁnd the shortest path from one vertex to another

in the graph. We shall see how to solve this problem efﬁciently in Chapter 24.

We are given two ordered sequences of symbols, X D hx 1 ; x 2 ; : : : ; x m i and

Y D hy 1 ; y 2 ; : : : ; y n i, and we wish to ﬁnd a longest common subsequence of

X and Y . A subsequence of X is just X with some (or possibly all or none) of

its elements removed. For example, one subsequence of hA; B; C; D; E; F; Gi

would be hB; C; E; Gi. The length of a longest common subsequence of X

and Y gives one measure of how similar these two sequences are. For example,

if the two sequences are base pairs in DNA strands, then we might consider

them similar if they have a long common subsequence. If X has m symbols

and Y has n symbols, then X and Y have 2 m

and 2 n

possible subsequences,

8 Chapter 1 The Role of Algorithms in Computing

respectively. Selecting all possible subsequences of X and Y and matching

them up could take a prohibitively long time unless m and n are very small.

We shall see in Chapter 15 how to use a general technique known as dynamic

programming to solve this problem much more efﬁciently.

We are given a mechanical design in terms of a library of parts, where each part

may include instances of other parts, and we need to list the parts in order so

that each part appears before any part that uses it. If the design comprises n

parts, then there are nŠ possible orders, where nŠ denotes the factorial function.

Because the factorial function grows faster than even an exponential function,

we cannot feasibly generate each possible order and then verify that, within

that order, each part appears before the parts using it (unless we have only a

few parts). This problem is an instance of topological sorting, and we shall see

in Chapter 22 how to solve this problem efﬁciently.

We are given n points in the plane, and we wish to ﬁnd the convex hull of

these points. The convex hull is the smallest convex polygon containing the

points. Intuitively, we can think of each point as being represented by a nail

sticking out from a board. The convex hull would be represented by a tight

rubber band that surrounds all the nails. Each nail around which the rubber

band makes a turn is a vertex of the convex hull. (See Figure 33.6 on page 1029

for an example.) Any of the 2 n

subsets of the points might be the vertices

of the convex hull. Knowing which points are vertices of the convex hull is

not quite enough, either, since we also need to know the order in which they

appear. There are many choices, therefore, for the vertices of the convex hull.

Chapter 33 gives two good methods for ﬁnding the convex hull.

These lists are far from exhaustive (as you again have probably surmised from

this book’s heft), but exhibit two characteristics that are common to many interest-

ing algorithmic problems:

1. They have many candidate solutions, the overwhelming majority of which do

not solve the problem at hand. Finding one that does, or one that is “best,” can

present quite a challenge.

2. They have practical applications. Of the problems in the above list, ﬁnding the

shortest path provides the easiest examples. A transportation ﬁrm, such as a

trucking or railroad company, has a ﬁnancial interest in ﬁnding shortest paths

through a road or rail network because taking shorter paths results in lower

labor and fuel costs. Or a routing node on the Internet may need to ﬁnd the

shortest path through the network in order to route a message quickly. Or a

person wishing to drive from New York to Boston may want to ﬁnd driving

directions from an appropriate Web site, or she may use her GPS while driving.

1.1 Algorithms 9

Not every problem solved by algorithms has an easily identiﬁed set of candidate

solutions. For example, suppose we are given a set of numerical values represent-

ing samples of a signal, and we want to compute the discrete Fourier transform of

these samples. The discrete Fourier transform converts the time domain to the fre-

quency domain, producing a set of numerical coefﬁcients, so that we can determine

the strength of various frequencies in the sampled signal. In addition to lying at

the heart of signal processing, discrete Fourier transforms have applications in data

compression and multiplying large polynomials and integers. Chapter 30 gives

an efﬁcient algorithm, the fast Fourier transform (commonly called the FFT), for

this problem, and the chapter also sketches out the design of a hardware circuit to

compute the FFT.

Data structures

This book also contains several data structures. A data structure is a way to store

and organize data in order to facilitate access and modiﬁcations. No single data

structure works well for all purposes, and so it is important to know the strengths

and limitations of several of them.

Technique

Although you can use this book as a “cookbook” for algorithms, you may someday

encounter a problem for which you cannot readily ﬁnd a published algorithm (many

of the exercises and problems in this book, for example). This book will teach you

techniques of algorithm design and analysis so that you can develop algorithms on

your own, show that they give the correct answer, and understand their efﬁciency.

Different chapters address different aspects of algorithmic problem solving. Some

chapters address speciﬁc problems, such as ﬁnding medians and order statistics in

Chapter 9, computing minimum spanning trees in Chapter 23, and determining a

maximum ﬂow in a network in Chapter 26. Other chapters address techniques,

such as divide-and-conquer in Chapter 4, dynamic programming in Chapter 15,

and amortized analysis in Chapter 17.

Hard problems

Most of this book is about efﬁcient algorithms. Our usual measure of efﬁciency

is speed, i.e., how long an algorithm takes to produce its result. There are some

problems, however, for which no efﬁcient solution is known. Chapter 34 studies

an interesting subset of these problems, which are known as NP-complete.

Why are NP-complete problems interesting? First, although no efﬁcient algo-

rithm for an NP-complete problem has ever been found, nobody has ever proven

10 Chapter 1 The Role of Algorithms in Computing

that an efﬁcient algorithm for one cannot exist. In other words, no one knows

whether or not efﬁcient algorithms exist for NP-complete problems. Second, the

set of NP-complete problems has the remarkable property that if an efﬁcient algo-

rithm exists for any one of them, then efﬁcient algorithms exist for all of them. This

relationship among the NP-complete problems makes the lack of efﬁcient solutions

all the more tantalizing. Third, several NP-complete problems are similar, but not

identical, to problems for which we do know of efﬁcient algorithms. Computer

scientists are intrigued by how a small change to the problem statement can cause

a big change to the efﬁciency of the best known algorithm.

You should know about NP-complete problems because some of them arise sur-

prisingly often in real applications. If you are called upon to produce an efﬁcient

algorithm for an NP-complete problem, you are likely to spend a lot of time in a

fruitless search. If you can show that the problem is NP-complete, you can instead

spend your time developing an efﬁcient algorithm that gives a good, but not the

best possible, solution.

As a concrete example, consider a delivery company with a central depot. Each

day, it loads up each delivery truck at the depot and sends it around to deliver goods

to several addresses. At the end of the day, each truck must end up back at the depot

so that it is ready to be loaded for the next day. To reduce costs, the company wants

to select an order of delivery stops that yields the lowest overall distance traveled

by each truck. This problem is the well-known “traveling-salesman problem,” and

it is NP-complete. It has no known efﬁcient algorithm. Under certain assumptions,

however, we know of efﬁcient algorithms that give an overall distance which is

not too far above the smallest possible. Chapter 35 discusses such “approximation

algorithms.”

Parallelism

For many years, we could count on processor clock speeds increasing at a steady

rate. Physical limitations present a fundamental roadblock to ever-increasing clock

speeds, however: because power density increases superlinearly with clock speed,

chips run the risk of melting once their clock speeds become high enough. In order

to perform more computations per second, therefore, chips are being designed to

contain not just one but several processing “cores.” We can liken these multicore

computers to several sequential computers on a single chip; in other words, they are

a type of “parallel computer.” In order to elicit the best performance from multicore

computers, we need to design algorithms with parallelism in mind. Chapter 27

presents a model for “multithreaded” algorithms, which take advantage of multiple

cores. This model has advantages from a theoretical standpoint, and it forms the

basis of several successful computer programs, including a championship chess

program.

1.2 Algorithms as a technology 11

Exercises

1.1-1

Give a real-world example that requires sorting or a real-world example that re-

quires computing a convex hull.

1.1-2

Other than speed, what other measures of efﬁciency might one use in a real-world

setting?

1.1-3

Select a data structure that you have seen previously, and discuss its strengths and

limitations.

1.1-4

How are the shortest-path and traveling-salesman problems given above similar?

How are they different?

1.1-5

Come up with a real-world problem in which only the best solution will do. Then

come up with one in which a solution that is “approximately” the best is good

enough.

1.2 Algorithms as a technology

Suppose computers were inﬁnitely fast and computer memory was free. Would

you have any reason to study algorithms? The answer is yes, if for no other reason

than that you would still like to demonstrate that your solution method terminates

and does so with the correct answer.

If computers were inﬁnitely fast, any correct method for solving a problem

would do. You would probably want your implementation to be within the bounds

of good software engineering practice (for example, your implementation should

be well designed and documented), but you would most often use whichever

method was the easiest to implement.

Of course, computers may be fast, but they are not inﬁnitely fast. And memory

may be inexpensive, but it is not free. Computing time is therefore a bounded

resource, and so is space in memory. You should use these resources wisely, and

algorithms that are efﬁcient in terms of time or space will help you do so.

12 Chapter 1 The Role of Algorithms in Computing

Efﬁciency

Different algorithms devised to solve the same problem often differ dramatically in

their efﬁciency. These differences can be much more signiﬁcant than differences

due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The ﬁrst,

known as insertion sort, takes time roughly equal to c 1 n 2

to sort n items, where c 1

is a constant that does not depend on n. That is, it takes time roughly proportional

to n 2

. The second, merge sort, takes time roughly equal to c 2 n lg n, where lg n

stands for log 2 n and c 2 is another constant that also does not depend on n. Inser-

tion sort typically has a smaller constant factor than merge sort, so that c 1 < c 2 .

We shall see that the constant factors can have far less of an impact on the running

time than the dependence on the input size n. Let’s write insertion sort’s running

time as c 1 n n and merge sort’s running time as c 2 n lg n. Then we see that where

insertion sort has a factor of n in its running time, merge sort has a factor of lg n,

which is much smaller. (For example, when n D 1000, lg n is approximately 10,

and when n equals one million, lg n is approximately only 20.) Although insertion

sort usually runs faster than merge sort for small input sizes, once the input size n

becomes large enough, merge sort’s advantage of lg n vs. n will more than com-

pensate for the difference in constant factors. No matter how much smaller c 1 is

than c 2 , there will always be a crossover point beyond which merge sort is faster.

For a concrete example, let us pit a faster computer (computer A) running inser-

tion sort against a slower computer (computer B) running merge sort. They each

must sort an array of 10 million numbers. (Although 10 million numbers might

seem like a lot, if the numbers are eight-byte integers, then the input occupies

about 80 megabytes, which ﬁts in the memory of even an inexpensive laptop com-

puter many times over.) Suppose that computer A executes 10 billion instructions

per second (faster than any single sequential computer at the time of this writing)

and computer B executes only 10 million instructions per second, so that com-

puter A is 1000 times faster than computer B in raw computing power. To make

the difference even more dramatic, suppose that the world’s craftiest programmer

codes insertion sort in machine language for computer A, and the resulting code

requires 2n 2

instructions to sort n numbers. Suppose further that just an average

programmer implements merge sort, using a high-level language with an inefﬁcient

compiler, with the resulting code taking 50n lg n instructions. To sort 10 million

numbers, computer A takes

2 .10 7 / 2

instructions

10 10 instructions/second

D 20,000 seconds (more than 5.5 hours) ;

while computer B takes

1.2 Algorithms as a technology 13

50 10 7

lg 10 7

instructions

10 7 instructions/second

1163 seconds (less than 20 minutes) :

By using an algorithm whose running time grows more slowly, even with a poor

compiler, computer B runs more than 17 times faster than computer A! The advan-

tage of merge sort is even more pronounced when we sort 100 million numbers:

where insertion sort takes more than 23 days, merge sort takes under four hours.

In general, as the problem size increases, so does the relative advantage of merge

sort.

Algorithms and other technologies

The example above shows that we should consider algorithms, like computer hard-

ware, as a technology. Total system performance depends on choosing efﬁcient

algorithms as much as on choosing fast hardware. Just as rapid advances are being

made in other computer technologies, they are being made in algorithms as well.

You might wonder whether algorithms are truly that important on contemporary

computers in light of other advanced technologies, such as

advanced computer architectures and fabrication technologies,

easy-to-use, intuitive, graphical user interfaces (GUIs),

object-oriented systems,

integrated Web technologies, and

fast networking, both wired and wireless.

The answer is yes. Although some applications do not explicitly require algorith-

mic content at the application level (such as some simple, Web-based applications),

many do. For example, consider a Web-based service that determines how to travel

from one location to another. Its implementation would rely on fast hardware, a

graphical user interface, wide-area networking, and also possibly on object ori-

entation. However, it would also require algorithms for certain operations, such

as ﬁnding routes (probably using a shortest-path algorithm), rendering maps, and

interpolating addresses.

Moreover, even an application that does not require algorithmic content at the

application level relies heavily upon algorithms. Does the application rely on fast

hardware? The hardware design used algorithms. Does the application rely on

graphical user interfaces? The design of any GUI relies on algorithms. Does the

application rely on networking? Routing in networks relies heavily on algorithms.

Was the application written in a language other than machine code? Then it was

processed by a compiler, interpreter, or assembler, all of which make extensive use

14 Chapter 1 The Role of Algorithms in Computing

of algorithms. Algorithms are at the core of most technologies used in contempo-

rary computers.

Furthermore, with the ever-increasing capacities of computers, we use them to

solve larger problems than ever before. As we saw in the above comparison be-

tween insertion sort and merge sort, it is at larger problem sizes that the differences

in efﬁciency between algorithms become particularly prominent.

Having a solid base of algorithmic knowledge and technique is one characteristic

that separates the truly skilled programmers from the novices. With modern com-

puting technology, you can accomplish some tasks without knowing much about

algorithms, but with a good background in algorithms, you can do much, much

more.

Exercises

1.2-1

Give an example of an application that requires algorithmic content at the applica-

tion level, and discuss the function of the algorithms involved.

1.2-2

Suppose we are comparing implementations of insertion sort and merge sort on the

same machine. For inputs of size n, insertion sort runs in 8n 2

steps, while merge

sort runs in 64n lg n steps. For which values of n does insertion sort beat merge

sort?

1.2-3

What is the smallest value of n such that an algorithm whose running time is 100n 2

runs faster than an algorithm whose running time is 2 n

on the same machine?

Problems

1-1 Comparison of running times

For each function f .n/ and time t in the following table, determine the largest

size n of a problem that can be solved in time t, assuming that the algorithm to

solve the problem takes f .n/ microseconds.

Notes for Chapter 1 15

1 1 1 1 1 1 1

second minute hour day month year century

lg n

p

n

n

n lg n

n 2

n 3

2 n

nŠ

Chapter notes

There are many excellent texts on the general topic of algorithms, including those

by Aho, Hopcroft, and Ullman [5, 6]; Baase and Van Gelder [28]; Brassard and

Bratley [54]; Dasgupta, Papadimitriou, and Vazirani [82]; Goodrich and Tamassia

[148]; Hofri [175]; Horowitz, Sahni, and Rajasekaran [181]; Johnsonbaugh and

Schaefer [193]; Kingston [205]; Kleinberg and Tardos [208]; Knuth [209, 210,

211]; Kozen [220]; Levitin [235]; Manber [242]; Mehlhorn [249, 250, 251]; Pur-

dom and Brown [287]; Reingold, Nievergelt, and Deo [293]; Sedgewick [306];

Sedgewick and Flajolet [307]; Skiena [318]; and Wilf [356]. Some of the more

practical aspects of algorithm design are discussed by Bentley [42, 43] and Gonnet

[145]. Surveys of the ﬁeld of algorithms can also be found in the Handbook of The-

oretical Computer Science, Volume A [342] and the CRC Algorithms and Theory of

Computation Handbook [25]. Overviews of the algorithms used in computational

biology can be found in textbooks by Gusﬁeld [156], Pevzner [275], Setubal and

Meidanis [310], and Waterman [350].

2 Getting Started

This chapter will familiarize you with the framework we shall use throughout the

book to think about the design and analysis of algorithms. It is self-contained, but

it does include several references to material that we introduce in Chapters 3 and 4.

(It also contains several summations, which Appendix A shows how to solve.)

We begin by examining the insertion sort algorithm to solve the sorting problem

introduced in Chapter 1. We deﬁne a “pseudocode” that should be familiar to you if

you have done computer programming, and we use it to show how we shall specify

our algorithms. Having speciﬁed the insertion sort algorithm, we then argue that it

correctly sorts, and we analyze its running time. The analysis introduces a notation

that focuses on how that time increases with the number of items to be sorted.

Following our discussion of insertion sort, we introduce the divide-and-conquer

approach to the design of algorithms and use it to develop an algorithm called

merge sort. We end with an analysis of merge sort’s running time.

2.1 Insertion sort

Our ﬁrst algorithm, insertion sort, solves the sorting problem introduced in Chap-

ter 1:

Input: A sequence of n numbers ha 1 ; a 2 ; : : : ; a n i.

Output: A permutation (reordering) ha 0

1

; a 0

2

; : : : ; a 0

n

i of the input sequence such

that a 0

1

a 0

2

a 0

n

.

The numbers that we wish to sort are also known as the keys. Although conceptu-

ally we are sorting a sequence, the input comes to us in the form of an array with n

elements.

In this book, we shall typically describe algorithms as programs written in a

pseudocode that is similar in many respects to C, C++, Java, Python, or Pascal. If

you have been introduced to any of these languages, you should have little trouble

2.1 Insertion sort 17

2

♣ ♣

♣ 2

♣

4

♣ ♣ ♣

♣ ♣ 4

♣

5

♣ ♣ ♣

♣ ♣ 5

♣

♣

7

♣ ♣

♣ ♣

♣ ♣

♣ ♣ 7

♣

10

♣ ♣

♣ ♣

♣ ♣

♣

♣ ♣

♣ ♣

10

♣

Figure 2.1 Sorting a hand of cards using insertion sort.

reading our algorithms. What separates pseudocode from “real” code is that in

pseudocode, we employ whatever expressive method is most clear and concise to

specify a given algorithm. Sometimes, the clearest method is English, so do not

be surprised if you come across an English phrase or sentence embedded within

a section of “real” code. Another difference between pseudocode and real code

is that pseudocode is not typically concerned with issues of software engineering.

Issues of data abstraction, modularity, and error handling are often ignored in order

to convey the essence of the algorithm more concisely.

We start with insertion sort, which is an efﬁcient algorithm for sorting a small

number of elements. Insertion sort works the way many people sort a hand of

playing cards. We start with an empty left hand and the cards face down on the

table. We then remove one card at a time from the table and insert it into the

correct position in the left hand. To ﬁnd the correct position for a card, we compare

it with each of the cards already in the hand, from right to left, as illustrated in

Figure 2.1. At all times, the cards held in the left hand are sorted, and these cards

were originally the top cards of the pile on the table.

We present our pseudocode for insertion sort as a procedure called INSERTION-

SORT, which takes as a parameter an array AŒ1 : : n containing a sequence of

length n that is to be sorted. (In the code, the number n of elements in A is denoted

by A:length.) The algorithm sorts the input numbers in place: it rearranges the

numbers within the array A, with at most a constant number of them stored outside

the array at any time. The input array A contains the sorted output sequence when

the INSERTION-SORT procedure is ﬁnished.

18 Chapter 2 Getting Started

1 2 3 4 5 6

5 2 4 6 1 3 (a)

1 2 3 4 5 6

2 5 4 6 1 3 (b)

1 2 3 4 5 6

2 4 5 6 1 3 (c)

1 2 3 4 5 6

2 4 5 6 1 3 (d)

1 2 3 4 5 6

2 4 5 6 1 3 (e)

1 2 3 4 5 6

2 4 5 6 1 3 (f)

Figure 2.2 The operation of INSERTION-SORT on the array A D h5; 2; 4; 6; 1; 3i. Array indices

appear above the rectangles, and values stored in the array positions appear within the rectangles.

(a)–(e) The iterations of the for loop of lines 1–8. In each iteration, the black rectangle holds the

key taken from AŒj , which is compared with the values in shaded rectangles to its left in the test of

line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows

indicate where the key moves to in line 8. (f) The ﬁnal sorted array.

INSERTION-SORT.A/

1 for j D 2 to A:length

2 key D AŒj

3 // Insert AŒj into the sorted sequence AŒ1 : : j 1.

4 i D j 1

5 while i > 0 and AŒi > key

6 AŒi C 1 D AŒi

7 i D i 1

8 AŒi C 1 D key

Loop invariants and the correctness of insertion sort

Figure 2.2 shows how this algorithm works for A D h5; 2; 4; 6; 1; 3i. The in-

dex j indicates the “current card” being inserted into the hand. At the beginning

of each iteration of the for loop, which is indexed by j , the subarray consisting

of elements AŒ1 : : j 1 constitutes the currently sorted hand, and the remaining

subarray AŒj C 1 : : n corresponds to the pile of cards still on the table. In fact,

elements AŒ1 : : j 1 are the elements originally in positions 1 through j 1, but

now in sorted order. We state these properties of AŒ1 : : j 1 formally as a loop

invariant:

At the start of each iteration of the for loop of lines 1–8, the subarray

AŒ1 : : j 1 consists of the elements originally in AŒ1 : : j 1, but in sorted

order.

We use loop invariants to help us understand why an algorithm is correct. We

must show three things about a loop invariant:

2.1 Insertion sort 19

Initialization: It is true prior to the ﬁrst iteration of the loop.

Maintenance: If it is true before an iteration of the loop, it remains true before the

next iteration.

Termination: When the loop terminates, the invariant gives us a useful property

that helps show that the algorithm is correct.

When the ﬁrst two properties hold, the loop invariant is true prior to every iteration

of the loop. (Of course, we are free to use established facts other than the loop

invariant itself to prove that the loop invariant remains true before each iteration.)

Note the similarity to mathematical induction, where to prove that a property holds,

you prove a base case and an inductive step. Here, showing that the invariant holds

before the ﬁrst iteration corresponds to the base case, and showing that the invariant

holds from iteration to iteration corresponds to the inductive step.

The third property is perhaps the most important one, since we are using the loop

invariant to show correctness. Typically, we use the loop invariant along with the

condition that caused the loop to terminate. The termination property differs from

how we usually use mathematical induction, in which we apply the inductive step

inﬁnitely; here, we stop the “induction” when the loop terminates.

Let us see how these properties hold for insertion sort.

Initialization: We start by showing that the loop invariant holds before the ﬁrst

loop iteration, when j D 2. 1 The subarray AŒ1 : : j 1, therefore, consists

of just the single element AŒ1, which is in fact the original element in AŒ1.

Moreover, this subarray is sorted (trivially, of course), which shows that the

loop invariant holds prior to the ﬁrst iteration of the loop.

Maintenance: Next, we tackle the second property: showing that each iteration

maintains the loop invariant. Informally, the body of the for loop works by

moving AŒj 1, AŒj 2, AŒj 3, and so on by one position to the right

until it ﬁnds the proper position for AŒj (lines 4–7), at which point it inserts

the value of AŒj (line 8). The subarray AŒ1 : : j then consists of the elements

originally in AŒ1 : : j , but in sorted order. Incrementing j for the next iteration

of the for loop then preserves the loop invariant.

A more formal treatment of the second property would require us to state and

show a loop invariant for the while loop of lines 5–7. At this point, however,

1

When the loop is a for loop, the moment at which we check the loop invariant just prior to the ﬁrst

iteration is immediately after the initial assignment to the loop-counter variable and just before the

ﬁrst test in the loop header. In the case of INSERTION-SORT, this time is after assigning 2 to the

variable j but before the ﬁrst test of whether j A:length.

20 Chapter 2 Getting Started

we prefer not to get bogged down in such formalism, and so we rely on our

informal analysis to show that the second property holds for the outer loop.

Termination: Finally, we examine what happens when the loop terminates. The

condition causing the for loop to terminate is that j > A:length D n. Because

each loop iteration increases j by 1, we must have j D n C 1 at that time.

Substituting n C 1 for j in the wording of loop invariant, we have that the

subarray AŒ1 : : n consists of the elements originally in AŒ1 : : n, but in sorted

order. Observing that the subarray AŒ1 : : n is the entire array, we conclude that

the entire array is sorted. Hence, the algorithm is correct.

We shall use this method of loop invariants to show correctness later in this

chapter and in other chapters as well.

Pseudocode conventions

We use the following conventions in our pseudocode.

Indentation indicates block structure. For example, the body of the for loop that

begins on line 1 consists of lines 2–8, and the body of the while loop that begins

on line 5 contains lines 6–7 but not line 8. Our indentation style applies to

if-else statements 2 as well. Using indentation instead of conventional indicators

of block structure, such as begin and end statements, greatly reduces clutter

while preserving, or even enhancing, clarity. 3

The looping constructs while, for, and repeat-until and the if-else conditional

construct have interpretations similar to those in C, C++, Java, Python, and

Pascal. 4 In this book, the loop counter retains its value after exiting the loop,

unlike some situations that arise in C++, Java, and Pascal. Thus, immediately

after a for loop, the loop counter’s value is the value that ﬁrst exceeded the for

loop bound. We used this property in our correctness argument for insertion

sort. The for loop header in line 1 is for j D 2 to A:length, and so when

this loop terminates, j D A:length C 1 (or, equivalently, j D n C 1, since

n D A:length). We use the keyword to when a for loop increments its loop

2

In an if-else statement, we indent else at the same level as its matching if. Although we omit the

keyword then, we occasionally refer to the portion executed when the test following if is true as a

then clause. For multiway tests, we use elseif for tests after the ﬁrst one.

3

Each pseudocode procedure in this book appears on one page so that you will not have to discern

levels of indentation in code that is split across pages.

4

Most block-structured languages have equivalent constructs, though the exact syntax may differ.

Python lacks repeat-until loops, and its for loops operate a little differently from the for loops in

this book.

2.1 Insertion sort 21

counter in each iteration, and we use the keyword downto when a for loop

decrements its loop counter. When the loop counter changes by an amount

greater than 1, the amount of change follows the optional keyword by.

The symbol “//” indicates that the remainder of the line is a comment.

A multiple assignment of the form i D j D e assigns to both variables i and j

the value of expression e; it should be treated as equivalent to the assignment

j D e followed by the assignment i D j .

Variables (such as i, j , and key) are local to the given procedure. We shall not

use global variables without explicit indication.

We access array elements by specifying the array name followed by the in-

dex in square brackets. For example, AŒi indicates the ith element of the

array A. The notation “: :” is used to indicate a range of values within an ar-

ray. Thus, AŒ1 : : j indicates the subarray of A consisting of the j elements

AŒ1; AŒ2; : : : ; AŒj .

We typically organize compound data into objects, which are composed of

attributes. We access a particular attribute using the syntax found in many

object-oriented programming languages: the object name, followed by a dot,

followed by the attribute name. For example, we treat an array as an object

with the attribute length indicating how many elements it contains. To specify

the number of elements in an array A, we write A:length.

We treat a variable representing an array or object as a pointer to the data rep-

resenting the array or object. For all attributes f of an object x, setting y D x

causes y:f to equal x:f . Moreover, if we now set x:f D 3, then afterward not

only does x:f equal 3, but y:f equals 3 as well. In other words, x and y point

to the same object after the assignment y D x.

Our attribute notation can “cascade.” For example, suppose that the attribute f

is itself a pointer to some type of object that has an attribute g. Then the notation

x:f :g is implicitly parenthesized as .x:f /:g. In other words, if we had assigned

y D x:f , then x:f :g is the same as y:g.

Sometimes, a pointer will refer to no object at all. In this case, we give it the

special value NIL.

We pass parameters to a procedure by value: the called procedure receives its

own copy of the parameters, and if it assigns a value to a parameter, the change

is not seen by the calling procedure. When objects are passed, the pointer to

the data representing the object is copied, but the object’s attributes are not. For

example, if x is a parameter of a called procedure, the assignment x D y within

the called procedure is not visible to the calling procedure. The assignment

x:f D 3, however, is visible. Similarly, arrays are passed by pointer, so that

22 Chapter 2 Getting Started

a pointer to the array is passed, rather than the entire array, and changes to

individual array elements are visible to the calling procedure.

A return statement immediately transfers control back to the point of call in

the calling procedure. Most return statements also take a value to pass back to

the caller. Our pseudocode differs from many programming languages in that

we allow multiple values to be returned in a single return statement.

The boolean operators “and” and “or” are short circuiting. That is, when we

evaluate the expression “x and y” we ﬁrst evaluate x. If x evaluates to FALSE,

then the entire expression cannot evaluate to TRUE, and so we do not evaluate y.

If, on the other hand, x evaluates to TRUE, we must evaluate y to determine the

value of the entire expression. Similarly, in the expression “x or y” we eval-

uate the expression y only if x evaluates to FALSE. Short-circuiting operators

allow us to write boolean expressions such as “x ¤ NIL and x:f D y” without

worrying about what happens when we try to evaluate x:f when x is NIL.

The keyword error indicates that an error occurred because conditions were

wrong for the procedure to have been called. The calling procedure is respon-

sible for handling the error, and so we do not specify what action to take.

Exercises

2.1-1

Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on the

array A D h31; 41; 59; 26; 41; 58i.

2.1-2

Rewrite the INSERTION-SORT procedure to sort into nonincreasing instead of non-

decreasing order.

2.1-3

Consider the searching problem:

Input: A sequence of n numbers A D ha 1 ; a 2 ; : : : ; a n i and a value .

Output: An index i such that D AŒi or the special value NIL if does not

appear in A.

Write pseudocode for linear search, which scans through the sequence, looking

for . Using a loop invariant, prove that your algorithm is correct. Make sure that

your loop invariant fulﬁlls the three necessary properties.

2.1-4

Consider the problem of adding two n-bit binary integers, stored in two n-element

arrays A and B. The sum of the two integers should be stored in binary form in

2.2 Analyzing algorithms 23

an .n C 1/-element array C . State the problem formally and write pseudocode for

adding the two integers.

2.2 Analyzing algorithms

Analyzing an algorithm has come to mean predicting the resources that the algo-

rithm requires. Occasionally, resources such as memory, communication band-

width, or computer hardware are of primary concern, but most often it is compu-

tational time that we want to measure. Generally, by analyzing several candidate

algorithms for a problem, we can identify a most efﬁcient one. Such analysis may

indicate more than one viable candidate, but we can often discard several inferior

algorithms in the process.

Before we can analyze an algorithm, we must have a model of the implemen-

tation technology that we will use, including a model for the resources of that

technology and their costs. For most of this book, we shall assume a generic one-

processor, random-access machine (RAM) model of computation as our imple-

mentation technology and understand that our algorithms will be implemented as

computer programs. In the RAM model, instructions are executed one after an-

other, with no concurrent operations.

Strictly speaking, we should precisely deﬁne the instructions of the RAM model

and their costs. To do so, however, would be tedious and would yield little insight

into algorithm design and analysis. Yet we must be careful not to abuse the RAM

model. For example, what if a RAM had an instruction that sorts? Then we could

sort in just one instruction. Such a RAM would be unrealistic, since real computers

do not have such instructions. Our guide, therefore, is how real computers are de-

signed. The RAM model contains instructions commonly found in real computers:

arithmetic (such as add, subtract, multiply, divide, remainder, ﬂoor, ceiling), data

movement (load, store, copy), and control (conditional and unconditional branch,

subroutine call and return). Each such instruction takes a constant amount of time.

The data types in the RAM model are integer and ﬂoating point (for storing real

numbers). Although we typically do not concern ourselves with precision in this

book, in some applications precision is crucial. We also assume a limit on the size

of each word of data. For example, when working with inputs of size n, we typ-

ically assume that integers are represented by c lg n bits for some constant c 1.

We require c 1 so that each word can hold the value of n, enabling us to index the

individual input elements, and we restrict c to be a constant so that the word size

does not grow arbitrarily. (If the word size could grow arbitrarily, we could store

huge amounts of data in one word and operate on it all in constant time—clearly

an unrealistic scenario.)

24 Chapter 2 Getting Started

Real computers contain instructions not listed above, and such instructions rep-

resent a gray area in the RAM model. For example, is exponentiation a constant-

time instruction? In the general case, no; it takes several instructions to compute x y

when x and y are real numbers. In restricted situations, however, exponentiation is

a constant-time operation. Many computers have a “shift left” instruction, which

in constant time shifts the bits of an integer by k positions to the left. In most

computers, shifting the bits of an integer by one position to the left is equivalent

to multiplication by 2, so that shifting the bits by k positions to the left is equiv-

alent to multiplication by 2 k

. Therefore, such computers can compute 2 k

in one

constant-time instruction by shifting the integer 1 by k positions to the left, as long

as k is no more than the number of bits in a computer word. We will endeavor to

avoid such gray areas in the RAM model, but we will treat computation of 2 k

as a

constant-time operation when k is a small enough positive integer.

In the RAM model, we do not attempt to model the memory hierarchy that is

common in contemporary computers. That is, we do not model caches or virtual

memory. Several computational models attempt to account for memory-hierarchy

effects, which are sometimes signiﬁcant in real programs on real machines. A

handful of problems in this book examine memory-hierarchy effects, but for the

most part, the analyses in this book will not consider them. Models that include

the memory hierarchy are quite a bit more complex than the RAM model, and so

they can be difﬁcult to work with. Moreover, RAM-model analyses are usually

excellent predictors of performance on actual machines.

Analyzing even a simple algorithm in the RAM model can be a challenge. The

mathematical tools required may include combinatorics, probability theory, alge-

braic dexterity, and the ability to identify the most signiﬁcant terms in a formula.

Because the behavior of an algorithm may be different for each possible input, we

need a means for summarizing that behavior in simple, easily understood formulas.

Even though we typically select only one machine model to analyze a given al-

gorithm, we still face many choices in deciding how to express our analysis. We

would like a way that is simple to write and manipulate, shows the important char-

acteristics of an algorithm’s resource requirements, and suppresses tedious details.

Analysis of insertion sort

The time taken by the INSERTION-SORT procedure depends on the input: sorting a

thousand numbers takes longer than sorting three numbers. Moreover, INSERTION-

SORT can take different amounts of time to sort two input sequences of the same

size depending on how nearly sorted they already are. In general, the time taken

by an algorithm grows with the size of the input, so it is traditional to describe the

running time of a program as a function of the size of its input. To do so, we need

to deﬁne the terms “running time” and “size of input” more carefully.

2.2 Analyzing algorithms 25

The best notion for input size depends on the problem being studied. For many

problems, such as sorting or computing discrete Fourier transforms, the most nat-

ural measure is the number of items in the input—for example, the array size n

for sorting. For many other problems, such as multiplying two integers, the best

measure of input size is the total number of bits needed to represent the input in

ordinary binary notation. Sometimes, it is more appropriate to describe the size of

the input with two numbers rather than one. For instance, if the input to an algo-

rithm is a graph, the input size can be described by the numbers of vertices and

edges in the graph. We shall indicate which input size measure is being used with

each problem we study.

The running time of an algorithm on a particular input is the number of primitive

operations or “steps” executed. It is convenient to deﬁne the notion of step so

that it is as machine-independent as possible. For the moment, let us adopt the

following view. A constant amount of time is required to execute each line of our

pseudocode. One line may take a different amount of time than another line, but

we shall assume that each execution of the ith line takes time c i , where c i is a

constant. This viewpoint is in keeping with the RAM model, and it also reﬂects

how the pseudocode would be implemented on most actual computers. 5

In the following discussion, our expression for the running time of INSERTION-

SORT will evolve from a messy formula that uses all the statement costs c i to a

much simpler notation that is more concise and more easily manipulated. This

simpler notation will also make it easy to determine whether one algorithm is more

efﬁcient than another.

We start by presenting the INSERTION-SORT procedure with the time “cost”

of each statement and the number of times each statement is executed. For each

j D 2; 3; : : : ; n, where n D A:length, we let t j denote the number of times the

while loop test in line 5 is executed for that value of j . When a for or while loop

exits in the usual way (i.e., due to the test in the loop header), the test is executed

one time more than the loop body. We assume that comments are not executable

statements, and so they take no time.

5

There are some subtleties here. Computational steps that we specify in English are often variants

of a procedure that requires more than just a constant amount of time. For example, later in this

book we might say “sort the points by x-coordinate,” which, as we shall see, takes more than a

constant amount of time. Also, note that a statement that calls a subroutine takes constant time,

though the subroutine, once invoked, may take more. That is, we separate the process of calling the

subroutine—passing parameters to it, etc.—from the process of executing the subroutine.

26 Chapter 2 Getting Started

INSERTION-SORT.A/ cost times

1 for j D 2 to A:length c 1 n

2 key D AŒj c 2 n 1

3 // Insert AŒj into the sorted

sequence AŒ1 : : j 1. 0 n 1

4 i D j 1 c 4 n 1

5 while i > 0 and AŒi > key c 5

P n

j D2

t j

6 AŒi C 1 D AŒi c 6

P n

j D2

.t j 1/

7 i D i 1 c 7

P n

j D2

.t j 1/

8 AŒi C 1 D key c 8 n 1

The running time of the algorithm is the sum of running times for each state-

ment executed; a statement that takes c i steps to execute and executes n times will

contribute c i n to the total running time. 6 To compute T .n/, the running time of

INSERTION-SORT on an input of n values, we sum the products of the cost and

times columns, obtaining

T .n/ D c 1 n C c 2 .n 1/ C c 4 .n 1/ C c 5

n X

j D2

t j C c 6

n X

j D2

.t j 1/

C c 7

n X

j D2

.t j 1/ C c 8 .n 1/ :

Even for inputs of a given size, an algorithm’s running time may depend on

which input of that size is given. For example, in INSERTION-SORT, the best

case occurs if the array is already sorted. For each j D 2; 3; : : : ; n, we then ﬁnd

that AŒi key in line 5 when i has its initial value of j 1. Thus t j D 1 for

j D 2; 3; : : : ; n, and the best-case running time is

T .n/ D c 1 n C c 2 .n 1/ C c 4 .n 1/ C c 5 .n 1/ C c 8 .n 1/

D .c 1 C c 2 C c 4 C c 5 C c 8 /n .c 2 C c 4 C c 5 C c 8 / :

We can express this running time as an C b for constants a and b that depend on

the statement costs c i ; it is thus a linear function of n.

If the array is in reverse sorted order—that is, in decreasing order—the worst

case results. We must compare each element AŒj with each element in the entire

sorted subarray AŒ1 : : j 1, and so t j D j for j D 2; 3; : : : ; n. Noting that

6

This characteristic does not necessarily hold for a resource such as memory. A statement that

references m words of memory and is executed n times does not necessarily reference mn distinct

words of memory.

2.2 Analyzing algorithms 27

n X

j D2

j D

n.n C 1/

2

1

and

n X

j D2

.j 1/ D

n.n 1/

2

(see Appendix A for a review of how to solve these summations), we ﬁnd that in

the worst case, the running time of INSERTION-SORT is

T .n/ D c 1 n C c 2 .n 1/ C c 4 .n 1/ C c 5

n.n C 1/

2

1

C c 6

n.n 1/

2

C c 7

n.n 1/

2

C c 8 .n 1/

D

c 5

2

C

c 6

2

C

c 7

2

n

2

C

c 1 C c 2 C c 4 C

c 5

2

c 6

2

c 7

2

C c 8

n

.c 2 C c 4 C c 5 C c 8 / :

We can express this worst-case running time as an 2 C bn C c for constants a, b,

and c that again depend on the statement costs c i ; it is thus a quadratic function

of n.

Typically, as in insertion sort, the running time of an algorithm is ﬁxed for a

given input, although in later chapters we shall see some interesting “randomized”

algorithms whose behavior can vary even for a ﬁxed input.

Worst-case and average-case analysis

In our analysis of insertion sort, we looked at both the best case, in which the input

array was already sorted, and the worst case, in which the input array was reverse

sorted. For the remainder of this book, though, we shall usually concentrate on

ﬁnding only the worst-case running time, that is, the longest running time for any

input of size n. We give three reasons for this orientation.

The worst-case running time of an algorithm gives us an upper bound on the

running time for any input. Knowing it provides a guarantee that the algorithm

will never take any longer. We need not make some educated guess about the

running time and hope that it never gets much worse.

For some algorithms, the worst case occurs fairly often. For example, in search-

ing a database for a particular piece of information, the searching algorithm’s

worst case will often occur when the information is not present in the database.

In some applications, searches for absent information may be frequent.

28 Chapter 2 Getting Started

The “average case” is often roughly as bad as the worst case. Suppose that we

randomly choose n numbers and apply insertion sort. How long does it take to

determine where in subarray AŒ1 : : j 1 to insert element AŒj ? On average,

half the elements in AŒ1 : : j 1 are less than AŒj , and half the elements are

greater. On average, therefore, we check half of the subarray AŒ1 : : j 1, and

so t j is about j=2. The resulting average-case running time turns out to be a

quadratic function of the input size, just like the worst-case running time.

In some particular cases, we shall be interested in the average-case running time

of an algorithm; we shall see the technique of probabilistic analysis applied to

various algorithms throughout this book. The scope of average-case analysis is

limited, because it may not be apparent what constitutes an “average” input for

a particular problem. Often, we shall assume that all inputs of a given size are

equally likely. In practice, this assumption may be violated, but we can sometimes

use a randomized algorithm, which makes random choices, to allow a probabilistic

analysis and yield an expected running time. We explore randomized algorithms

more in Chapter 5 and in several other subsequent chapters.

Order of growth

We used some simplifying abstractions to ease our analysis of the INSERTION-

SORT procedure. First, we ignored the actual cost of each statement, using the

constants c i to represent these costs. Then, we observed that even these constants

give us more detail than we really need: we expressed the worst-case running time

as an 2 C bn C c for some constants a, b, and c that depend on the statement

costs c i . We thus ignored not only the actual statement costs, but also the abstract

costs c i .

We shall now make one more simplifying abstraction: it is the rate of growth,

or order of growth, of the running time that really interests us. We therefore con-

sider only the leading term of a formula (e.g., an 2

), since the lower-order terms are

relatively insigniﬁcant for large values of n. We also ignore the leading term’s con-

stant coefﬁcient, since constant factors are less signiﬁcant than the rate of growth

in determining computational efﬁciency for large inputs. For insertion sort, when

we ignore the lower-order terms and the leading term’s constant coefﬁcient, we are

left with the factor of n 2

from the leading term. We write that insertion sort has a

worst-case running time of ‚.n 2 / (pronounced “theta of n-squared”). We shall use

‚-notation informally in this chapter, and we will deﬁne it precisely in Chapter 3.

We usually consider one algorithm to be more efﬁcient than another if its worst-

case running time has a lower order of growth. Due to constant factors and lower-

order terms, an algorithm whose running time has a higher order of growth might

take less time for small inputs than an algorithm whose running time has a lower

2.3 Designing algorithms 29

order of growth. But for large enough inputs, a ‚.n 2 / algorithm, for example, will

run more quickly in the worst case than a ‚.n 3 / algorithm.

Exercises

2.2-1

Express the function n 3 =1000 100n 2 100n C 3 in terms of ‚-notation.

2.2-2

Consider sorting n numbers stored in array A by ﬁrst ﬁnding the smallest element

of A and exchanging it with the element in AŒ1. Then ﬁnd the second smallest

element of A, and exchange it with AŒ2. Continue in this manner for the ﬁrst n1

elements of A. Write pseudocode for this algorithm, which is known as selection

sort. What loop invariant does this algorithm maintain? Why does it need to run

for only the ﬁrst n 1 elements, rather than for all n elements? Give the best-case

and worst-case running times of selection sort in ‚-notation.

2.2-3

Consider linear search again (see Exercise 2.1-3). How many elements of the in-

put sequence need to be checked on the average, assuming that the element being

searched for is equally likely to be any element in the array? How about in the

worst case? What are the average-case and worst-case running times of linear

search in ‚-notation? Justify your answers.

2.2-4

How can we modify almost any algorithm to have a good best-case running time?

2.3 Designing algorithms

We can choose from a wide range of algorithm design techniques. For insertion

sort, we used an incremental approach: having sorted the subarray AŒ1 : : j 1,

we inserted the single element AŒj into its proper place, yielding the sorted

subarray AŒ1 : : j .

In this section, we examine an alternative design approach, known as “divide-

and-conquer,” which we shall explore in more detail in Chapter 4. We’ll use divide-

and-conquer to design a sorting algorithm whose worst-case running time is much

less than that of insertion sort. One advantage of divide-and-conquer algorithms is

that their running times are often easily determined using techniques that we will

see in Chapter 4.

30 Chapter 2 Getting Started

2.3.1 The divide-and-conquer approach

Many useful algorithms are recursive in structure: to solve a given problem, they

call themselves recursively one or more times to deal with closely related sub-

problems. These algorithms typically follow a divide-and-conquer approach: they

break the problem into several subproblems that are similar to the original prob-

lem but smaller in size, solve the subproblems recursively, and then combine these

solutions to create a solution to the original problem.

The divide-and-conquer paradigm involves three steps at each level of the recur-

sion:

Divide the problem into a number of subproblems that are smaller instances of the

same problem.

Conquer the subproblems by solving them recursively. If the subproblem sizes are

small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original prob-

lem.

The merge sort algorithm closely follows the divide-and-conquer paradigm. In-

tuitively, it operates as follows.

Divide: Divide the n-element sequence to be sorted into two subsequences of n=2

elements each.

Conquer: Sort the two subsequences recursively using merge sort.

Combine: Merge the two sorted subsequences to produce the sorted answer.

The recursion “bottoms out” when the sequence to be sorted has length 1, in which

case there is no work to be done, since every sequence of length 1 is already in

sorted order.

The key operation of the merge sort algorithm is the merging of two sorted

sequences in the “combine” step. We merge by calling an auxiliary procedure

MERGE.A; p; q; r/, where A is an array and p, q, and r are indices into the array

such that p q < r. The procedure assumes that the subarrays AŒp : : q and

AŒq C 1 : : r are in sorted order. It merges them to form a single sorted subarray

that replaces the current subarray AŒp : : r.

Our MERGE procedure takes time ‚.n/, where n D r p C 1 is the total

number of elements being merged, and it works as follows. Returning to our card-

playing motif, suppose we have two piles of cards face up on a table. Each pile is

sorted, with the smallest cards on top. We wish to merge the two piles into a single

sorted output pile, which is to be face down on the table. Our basic step consists

of choosing the smaller of the two cards on top of the face-up piles, removing it

from its pile (which exposes a new top card), and placing this card face down onto

2.3 Designing algorithms 31

the output pile. We repeat this step until one input pile is empty, at which time

we just take the remaining input pile and place it face down onto the output pile.

Computationally, each basic step takes constant time, since we are comparing just

the two top cards. Since we perform at most n basic steps, merging takes ‚.n/

time.

The following pseudocode implements the above idea, but with an additional

twist that avoids having to check whether either pile is empty in each basic step.

We place on the bottom of each pile a sentinel card, which contains a special value

that we use to simplify our code. Here, we use 1 as the sentinel value, so that

whenever a card with 1 is exposed, it cannot be the smaller card unless both piles

have their sentinel cards exposed. But once that happens, all the nonsentinel cards

have already been placed onto the output pile. Since we know in advance that

exactly r p C 1 cards will be placed onto the output pile, we can stop once we

have performed that many basic steps.

MERGE.A; p; q; r/

1 n 1 D q p C 1

2 n 2 D r q

3 let LŒ1 : : n 1 C 1 and RŒ1 : : n 2 C 1 be new arrays

4 for i D 1 to n 1

5 LŒi D AŒp C i 1

6 for j D 1 to n 2

7 RŒj D AŒq C j

8 LŒn 1 C 1 D 1

9 RŒn 2 C 1 D 1

10 i D 1

11 j D 1

12 for k D p to r

13 if LŒi RŒj

14 AŒk D LŒi

15 i D i C 1

16 else AŒk D RŒj

17 j D j C 1

In detail, the MERGE procedure works as follows. Line 1 computes the length n 1

of the subarray AŒp : : q, and line 2 computes the length n 2 of the subarray

AŒq C 1 : : r. We create arrays L and R (“left” and “right”), of lengths n 1 C 1

and n 2 C 1, respectively, in line 3; the extra position in each array will hold the

sentinel. The for loop of lines 4–5 copies the subarray AŒp : : q into LŒ1 : : n 1 ,

and the for loop of lines 6–7 copies the subarray AŒq C 1 : : r into RŒ1 : : n 2 .

Lines 8–9 put the sentinels at the ends of the arrays L and R. Lines 10–17, illus-

32 Chapter 2 Getting Started

A

L R

1 2 3 4 1 2 3 4

i j

k

(a)

2 4 5 7 1 2 3 6

A

L R

1 2 3 4 1 2 3 4

i j

k

(b)

2 4 5 7

1

2 3 6 1

2 4 5 7 1 2 3 6 4 5 7 1 2 3 6

A

L R

9 10 11 12 13 14 15 16

1 2 3 4 1 2 3 4

i j

k

(c)

2 4 5 7

1

2 3 6 1

5 7 1 2 3 6 2 A

L R

1 2 3 4 1 2 3 4

i j

k

(d)

2 4 5 7

1

2 3 6 1

7 1 2 3 6 2 2

5

∞

5

∞

5

∞

5

∞

5

∞

5

∞

5

∞

5

∞

9 10 11 12 13 14 15 16

9 10 11 12 13 14 15 16

9 10 11 12 13 14 15 16 8

…

17

…

8

…

17

…

8

…

17

…

8

…

17

…

Figure 2.3 The operation of lines 10–17 in the call MERGE.A; 9; 12; 16/, when the subarray

AŒ9 : : 16 contains the sequence h2; 4; 5; 7; 1; 2; 3; 6i. After copying and inserting sentinels, the

array L contains h2; 4; 5; 7; 1i, and the array R contains h1; 2; 3; 6; 1i. Lightly shaded positions

in A contain their ﬁnal values, and lightly shaded positions in L and R contain values that have yet

to be copied back into A. Taken together, the lightly shaded positions always comprise the values

originally in AŒ9 : : 16, along with the two sentinels. Heavily shaded positions in A contain values

that will be copied over, and heavily shaded positions in L and R contain values that have already

been copied back into A. (a)–(h) The arrays A, L, and R, and their respective indices k, i, and j

prior to each iteration of the loop of lines 12–17.

trated in Figure 2.3, perform the r p C1 basic steps by maintaining the following

loop invariant:

At the start of each iteration of the for loop of lines 12–17, the subarray

AŒp : : k 1 contains the k p smallest elements of LŒ1 : : n 1 C 1 and

RŒ1 : : n 2 C 1, in sorted order. Moreover, LŒi and RŒj are the smallest

elements of their arrays that have not been copied back into A.

We must show that this loop invariant holds prior to the ﬁrst iteration of the for

loop of lines 12–17, that each iteration of the loop maintains the invariant, and

that the invariant provides a useful property to show correctness when the loop

terminates.

Initialization: Prior to the ﬁrst iteration of the loop, we have k D p, so that the

subarray AŒp : : k 1 is empty. This empty subarray contains the k p D 0

smallest elements of L and R, and since i D j D 1, both LŒi and RŒj are the

smallest elements of their arrays that have not been copied back into A.

2.3 Designing algorithms 33

A

L R

1 2 3 4 1 2 3 4

i j

k

(e)

2 4 5 7

1

2 3 6 1

1 2 3 6 2 2 3 A

L R

1 2 3 4 1 2 3 4

i j

k

(f)

2 4 5 7

1

2 3 6 1

2 3 6 2 2 3 4

A

L R

1 2 3 4 1 2 3 4

i j

k

(g)

2 4 5 7

1

2 3 6 1

3 6 2 2 3 4 5 A

L R

1 2 3 4 1 2 3 4

i j

k

(h)

2 4 5 7

1

2 3 6 1

6 2 2 3 4 5

5

∞

5

∞

5

∞

5

∞

5

∞

5

∞

5

∞

5

∞

6

A

L R

1 2 3 4 1 2 3 4

i j

k

(i)

2 4 5 7

1

2 3 6 1

7 2 2 3 4 5

5

∞

5

∞

6

9 10 11 12 13 14 15 16

9 10 11 12 13 14 15 16

9 10 11 12 13 14 15 16

9 10 11 12 13 14 15 16

9 10 11 12 13 14 15 16

8

…

17

…

8

…

17

…

8

…

17

…

8

…

17

…

8

…

17

…

Figure 2.3, continued (i) The arrays and indices at termination. At this point, the subarray in

AŒ9 : : 16 is sorted, and the two sentinels in L and R are the only two elements in these arrays that

have not been copied into A.

Maintenance: To see that each iteration maintains the loop invariant, let us ﬁrst

suppose that LŒi RŒj . Then LŒi is the smallest element not yet copied

back into A. Because AŒp : : k 1 contains the k p smallest elements, after

line 14 copies LŒi into AŒk, the subarray AŒp : : k will contain the k p C 1

smallest elements. Incrementing k (in the for loop update) and i (in line 15)

reestablishes the loop invariant for the next iteration. If instead LŒi > RŒj ,

then lines 16–17 perform the appropriate action to maintain the loop invariant.

Termination: At termination, k D r C 1. By the loop invariant, the subarray

AŒp : : k 1, which is AŒp : : r, contains the k p D r p C 1 smallest

elements of LŒ1 : : n 1 C 1 and RŒ1 : : n 2 C 1, in sorted order. The arrays L

and R together contain n 1 C n 2 C 2 D r p C 3 elements. All but the two

largest have been copied back into A, and these two largest elements are the

sentinels.

34 Chapter 2 Getting Started

To see that the MERGE procedure runs in ‚.n/ time, where n D r p C 1,

observe that each of lines 1–3 and 8–11 takes constant time, the for loops of

lines 4–7 take ‚.n 1 C n 2 / D ‚.n/ time, 7 and there are n iterations of the for

loop of lines 12–17, each of which takes constant time.

We can now use the MERGE procedure as a subroutine in the merge sort al-

gorithm. The procedure MERGE-SORT.A; p; r/ sorts the elements in the subar-

ray AŒp : : r. If p r, the subarray has at most one element and is therefore

already sorted. Otherwise, the divide step simply computes an index q that par-

titions AŒp : : r into two subarrays: AŒp : : q, containing dn=2e elements, and

AŒq C 1 : : r, containing bn=2c elements. 8

MERGE-SORT.A; p; r/

1 if p < r

2 q D b.p C r/=2c

3 MERGE-SORT.A; p; q/

4 MERGE-SORT.A; q C 1; r/

5 MERGE.A; p; q; r/

To sort the entire sequence A D hAŒ1; AŒ2; : : : ; AŒni, we make the initial call

MERGE-SORT.A; 1; A:length/, where once again A:length D n. Figure 2.4 il-

lustrates the operation of the procedure bottom-up when n is a power of 2. The

algorithm consists of merging pairs of 1-item sequences to form sorted sequences

of length 2, merging pairs of sequences of length 2 to form sorted sequences of

length 4, and so on, until two sequences of length n=2 are merged to form the ﬁnal

sorted sequence of length n.

2.3.2 Analyzing divide-and-conquer algorithms

When an algorithm contains a recursive call to itself, we can often describe its

running time by a recurrence equation or recurrence, which describes the overall

running time on a problem of size n in terms of the running time on smaller inputs.

We can then use mathematical tools to solve the recurrence and provide bounds on

the performance of the algorithm.

7

We shall see in Chapter 3 how to formally interpret equations containing ‚-notation.

8

The expression dxe denotes the least integer greater than or equal to x, and bxc denotes the greatest

integer less than or equal to x. These notations are deﬁned in Chapter 3. The easiest way to verify

that setting q to b.p C r/=2c yields subarrays AŒp : : q and AŒq C 1 : : r of sizes dn=2e and bn=2c,

respectively, is to examine the four cases that arise depending on whether each of p and r is odd or

even.

2.3 Designing algorithms 35

5 2 4 7 1 3 2 6

2 5 4 7 1 3 2 6

2 4 5 7 1 2 3 6

1 2 2 3 4 5 6 7

merge

merge

merge

sorted sequence

initial sequence

merge merge merge merge

Figure 2.4 The operation of merge sort on the array A D h5; 2; 4; 7; 1; 3; 2; 6i. The lengths of the

sorted sequences being merged increase as the algorithm progresses from bottom to top.

A recurrence for the running time of a divide-and-conquer algorithm falls out

from the three steps of the basic paradigm. As before, we let T .n/ be the running

time on a problem of size n. If the problem size is small enough, say n c

for some constant c, the straightforward solution takes constant time, which we

write as ‚.1/. Suppose that our division of the problem yields a subproblems,

each of which is 1=b the size of the original. (For merge sort, both a and b are 2,

but we shall see many divide-and-conquer algorithms in which a ¤ b.) It takes

time T .n=b/ to solve one subproblem of size n=b, and so it takes time aT .n=b/

to solve a of them. If we take D.n/ time to divide the problem into subproblems

and C.n/ time to combine the solutions to the subproblems into the solution to the

original problem, we get the recurrence

T .n/ D

(

‚.1/ if n c ;

aT .n=b/ C D.n/ C C.n/ otherwise :

In Chapter 4, we shall see how to solve common recurrences of this form.

Analysis of merge sort

Although the pseudocode for MERGE-SORT works correctly when the number of

elements is not even, our recurrence-based analysis is simpliﬁed if we assume that

36 Chapter 2 Getting Started

the original problem size is a power of 2. Each divide step then yields two subse-

quences of size exactly n=2. In Chapter 4, we shall see that this assumption does

not affect the order of growth of the solution to the recurrence.

We reason as follows to set up the recurrence for T .n/, the worst-case running

time of merge sort on n numbers. Merge sort on just one element takes constant

time. When we have n > 1 elements, we break down the running time as follows.

Divide: The divide step just computes the middle of the subarray, which takes

constant time. Thus, D.n/ D ‚.1/.

Conquer: We recursively solve two subproblems, each of size n=2, which con-

tributes 2T .n=2/ to the running time.

Combine: We have already noted that the MERGE procedure on an n-element

subarray takes time ‚.n/, and so C.n/ D ‚.n/.

When we add the functions D.n/ and C.n/ for the merge sort analysis, we are

adding a function that is ‚.n/ and a function that is ‚.1/. This sum is a linear

function of n, that is, ‚.n/. Adding it to the 2T .n=2/ term from the “conquer”

step gives the recurrence for the worst-case running time T .n/ of merge sort:

T .n/ D

(

‚.1/ if n D 1 ;

2T .n=2/ C ‚.n/ if n > 1 :

(2.1)

In Chapter 4, we shall see the “master theorem,” which we can use to show

that T .n/ is ‚.n lg n/, where lg n stands for log

2

n. Because the logarithm func-

tion grows more slowly than any linear function, for large enough inputs, merge

sort, with its ‚.n lg n/ running time, outperforms insertion sort, whose running

time is ‚.n 2 /, in the worst case.

We do not need the master theorem to intuitively understand why the solution to

the recurrence (2.1) is T .n/ D ‚.n lg n/. Let us rewrite recurrence (2.1) as

T .n/ D

(

c if n D 1 ;

2T .n=2/ C cn if n > 1 ;

(2.2)

where the constant c represents the time required to solve problems of size 1 as

well as the time per array element of the divide and combine steps. 9

9

It is unlikely that the same constant exactly represents both the time to solve problems of size 1

and the time per array element of the divide and combine steps. We can get around this problem by

letting c be the larger of these times and understanding that our recurrence gives an upper bound on

the running time, or by letting c be the lesser of these times and understanding that our recurrence

gives a lower bound on the running time. Both bounds are on the order of n lg n and, taken together,

give a ‚.n lg n/ running time.

2.3 Designing algorithms 37

Figure 2.5 shows how we can solve recurrence (2.2). For convenience, we as-

sume that n is an exact power of 2. Part (a) of the ﬁgure shows T .n/, which we

expand in part (b) into an equivalent tree representing the recurrence. The cn term

is the root (the cost incurred at the top level of recursion), and the two subtrees of

the root are the two smaller recurrences T .n=2/. Part (c) shows this process carried

one step further by expanding T .n=2/. The cost incurred at each of the two sub-

nodes at the second level of recursion is cn=2. We continue expanding each node

in the tree by breaking it into its constituent parts as determined by the recurrence,

until the problem sizes get down to 1, each with a cost of c. Part (d) shows the

resulting recursion tree.

Next, we add the costs across each level of the tree. The top level has total

cost cn, the next level down has total cost c.n=2/ C c.n=2/ D cn, the level after

that has total cost c.n=4/Cc.n=4/Cc.n=4/Cc.n=4/ D cn, and so on. In general,

the level i below the top has 2 i

nodes, each contributing a cost of c.n=2 i /, so that

the ith level below the top has total cost 2 i c.n=2 i / D cn. The bottom level has n

nodes, each contributing a cost of c, for a total cost of cn.

The total number of levels of the recursion tree in Figure 2.5 is lg n C 1, where

n is the number of leaves, corresponding to the input size. An informal inductive

argument justiﬁes this claim. The base case occurs when n D 1, in which case the

tree has only one level. Since lg 1 D 0, we have that lg n C 1 gives the correct

number of levels. Now assume as an inductive hypothesis that the number of levels

of a recursion tree with 2 i

leaves is lg 2 i C 1 D i C 1 (since for any value of i,

we have that lg 2 i D i). Because we are assuming that the input size is a power

of 2, the next input size to consider is 2 iC1

. A tree with n D 2 iC1

leaves has

one more level than a tree with 2 i

leaves, and so the total number of levels is

.i C 1/ C 1 D lg 2 iC1 C 1.

To compute the total cost represented by the recurrence (2.2), we simply add up

the costs of all the levels. The recursion tree has lg n C 1 levels, each costing cn,

for a total cost of cn.lg n C 1/ D cn lg n C cn. Ignoring the low-order term and

the constant c gives the desired result of ‚.n lg n/.

Exercises

2.3-1

Using Figure 2.4 as a model, illustrate the operation of merge sort on the array

A D h3; 41; 52; 26; 38; 57; 9; 49i.

2.3-2

Rewrite the MERGE procedure so that it does not use sentinels, instead stopping

once either array L or R has had all its elements copied back to A and then copying

the remainder of the other array back into A.

38 Chapter 2 Getting Started

cn

cn

…

Total: cn lg n + cn

cn

lg n

cn

n

c c c c c c c

…

(d)

(c)

cn

T(n/2) T(n/2)

(b)

T(n)

(a)

cn

cn/2

T(n/4) T(n/4)

cn/2

T(n/4) T(n/4)

cn

cn/2

cn/4 cn/4

cn/2

cn/4 cn/4

Figure 2.5 How to construct a recursion tree for the recurrence T .n/ D 2T .n=2/ C cn.

Part (a) shows T .n/, which progressively expands in (b)–(d) to form the recursion tree. The fully

expanded tree in part (d) has lg n C 1 levels (i.e., it has height lg n, as indicated), and each level

contributes a total cost of cn. The total cost, therefore, is cn lg n C cn, which is ‚.n lg n/.

Problems for Chapter 2 39

2.3-3

Use mathematical induction to show that when n is an exact power of 2, the solu-

tion of the recurrence

T .n/ D

(

2 if n D 2 ;

2T .n=2/ C n if n D 2 k

, for k > 1

is T .n/ D n lg n.

2.3-4

We can express insertion sort as a recursive procedure as follows. In order to sort

AŒ1 : : n, we recursively sort AŒ1 : : n 1 and then insert AŒn into the sorted array

AŒ1 : : n 1. Write a recurrence for the running time of this recursive version of

insertion sort.

2.3-5

Referring back to the searching problem (see Exercise 2.1-3), observe that if the

sequence A is sorted, we can check the midpoint of the sequence against and

eliminate half of the sequence from further consideration. The binary search al-

gorithm repeats this procedure, halving the size of the remaining portion of the

sequence each time. Write pseudocode, either iterative or recursive, for binary

search. Argue that the worst-case running time of binary search is ‚.lg n/.

2.3-6

Observe that the while loop of lines 5–7 of the INSERTION-SORT procedure in

Section 2.1 uses a linear search to scan (backward) through the sorted subarray

AŒ1 : : j 1. Can we use a binary search (see Exercise 2.3-5) instead to improve

the overall worst-case running time of insertion sort to ‚.n lg n/?

2.3-7 ?

Describe a ‚.n lg n/-time algorithm that, given a set S of n integers and another

integer x, determines whether or not there exist two elements in S whose sum is

exactly x.

Problems

2-1 Insertion sort on small arrays in merge sort

Although merge sort runs in ‚.n lg n/ worst-case time and insertion sort runs

in ‚.n 2 / worst-case time, the constant factors in insertion sort can make it faster

in practice for small problem sizes on many machines. Thus, it makes sense to

coarsen the leaves of the recursion by using insertion sort within merge sort when

40 Chapter 2 Getting Started

subproblems become sufﬁciently small. Consider a modiﬁcation to merge sort in

which n=k sublists of length k are sorted using insertion sort and then merged

using the standard merging mechanism, where k is a value to be determined.

a. Show that insertion sort can sort the n=k sublists, each of length k, in ‚.nk/

worst-case time.

b. Show how to merge the sublists in ‚.n lg.n=k// worst-case time.

c. Given that the modiﬁed algorithm runs in ‚.nk C n lg.n=k// worst-case time,

what is the largest value of k as a function of n for which the modiﬁed algorithm

has the same running time as standard merge sort, in terms of ‚-notation?

d. How should we choose k in practice?

2-2 Correctness of bubblesort

Bubblesort is a popular, but inefﬁcient, sorting algorithm. It works by repeatedly

swapping adjacent elements that are out of order.

BUBBLESORT.A/

1 for i D 1 to A:length 1

2 for j D A:length downto i C 1

3 if AŒj < AŒj 1

4 exchange AŒj with AŒj 1

a. Let A 0

denote the output of BUBBLESORT.A/. To prove that BUBBLESORT is

correct, we need to prove that it terminates and that

A

0

Œ1 A

0

Œ2 A

0

Œn ; (2.3)

where n D A:length. In order to show that BUBBLESORT actually sorts, what

else do we need to prove?

The next two parts will prove inequality (2.3).

b. State precisely a loop invariant for the for loop in lines 2–4, and prove that this

loop invariant holds. Your proof should use the structure of the loop invariant

proof presented in this chapter.

c. Using the termination condition of the loop invariant proved in part (b), state

a loop invariant for the for loop in lines 1–4 that will allow you to prove in-

equality (2.3). Your proof should use the structure of the loop invariant proof

presented in this chapter.

Problems for Chapter 2 41

d. What is the worst-case running time of bubblesort? How does it compare to the

running time of insertion sort?

2-3 Correctness of Horner’s rule

The following code fragment implements Horner’s rule for evaluating a polynomial

P.x/ D

n X

kD0

a k x

k

D a 0 C x.a 1 C x.a 2 C C x.a n1 C xa n / // ;

given the coefﬁcients a 0 ; a 1 ; : : : ; a n and a value for x:

1 y D 0

2 for i D n downto 0

3 y D a i C x y

a. In terms of ‚-notation, what is the running time of this code fragment for

Horner’s rule?

b. Write pseudocode to implement the naive polynomial-evaluation algorithm that

computes each term of the polynomial from scratch. What is the running time

of this algorithm? How does it compare to Horner’s rule?

c. Consider the following loop invariant:

At the start of each iteration of the for loop of lines 2–3,

y D

n.iC1/ X

kD0

a kCiC1 x

k

:

Interpret a summation with no terms as equaling 0. Following the structure of

the loop invariant proof presented in this chapter, use this loop invariant to show

that, at termination, y D

P n

kD0

a k x k

.

d. Conclude by arguing that the given code fragment correctly evaluates a poly-

nomial characterized by the coefﬁcients a 0 ; a 1 ; : : : ; a n .

2-4 Inversions

Let AŒ1 : : n be an array of n distinct numbers. If i < j and AŒi > AŒj , then the

pair .i; j / is called an inversion of A.

a. List the ﬁve inversions of the array h2; 3; 8; 6; 1i.

42 Chapter 2 Getting Started

b. What array with elements from the set f1; 2; : : : ; ng has the most inversions?

How many does it have?

c. What is the relationship between the running time of insertion sort and the

number of inversions in the input array? Justify your answer.

d. Give an algorithm that determines the number of inversions in any permutation

on n elements in ‚.n lg n/ worst-case time. (Hint: Modify merge sort.)

Chapter notes

In 1968, Knuth published the ﬁrst of three volumes with the general title The Art of

Computer Programming [209, 210, 211]. The ﬁrst volume ushered in the modern

study of computer algorithms with a focus on the analysis of running time, and the

full series remains an engaging and worthwhile reference for many of the topics

presented here. According to Knuth, the word “algorithm” is derived from the

name “al-Khowˆarizmˆı,” a ninth-century Persian mathematician.

Aho, Hopcroft, and Ullman [5] advocated the asymptotic analysis of algo-

rithms—using notations that Chapter 3 introduces, including ‚-notation—as a

means of comparing relative performance. They also popularized the use of re-

currence relations to describe the running times of recursive algorithms.

Knuth [211] provides an encyclopedic treatment of many sorting algorithms. His

comparison of sorting algorithms (page 381) includes exact step-counting analyses,

like the one we performed here for insertion sort. Knuth’s discussion of insertion

sort encompasses several variations of the algorithm. The most important of these

is Shell’s sort, introduced by D. L. Shell, which uses insertion sort on periodic

subsequences of the input to produce a faster sorting algorithm.

Merge sort is also described by Knuth. He mentions that a mechanical colla-

tor capable of merging two decks of punched cards in a single pass was invented

in 1938. J. von Neumann, one of the pioneers of computer science, apparently

wrote a program for merge sort on the EDVAC computer in 1945.

The early history of proving programs correct is described by Gries [153], who

credits P. Naur with the ﬁrst article in this ﬁeld. Gries attributes loop invariants to

R. W. Floyd. The textbook by Mitchell [256] describes more recent progress in

proving programs correct.

3 Growth of Functions

The order of growth of the running time of an algorithm, deﬁned in Chapter 2,

gives a simple characterization of the algorithm’s efﬁciency and also allows us to

compare the relative performance of alternative algorithms. Once the input size n

becomes large enough, merge sort, with its ‚.n lg n/ worst-case running time,

beats insertion sort, whose worst-case running time is ‚.n 2 /. Although we can

sometimes determine the exact running time of an algorithm, as we did for insertion

sort in Chapter 2, the extra precision is not usually worth the effort of computing

it. For large enough inputs, the multiplicative constants and lower-order terms of

an exact running time are dominated by the effects of the input size itself.

When we look at input sizes large enough to make only the order of growth of

the running time relevant, we are studying the asymptotic efﬁciency of algorithms.

That is, we are concerned with how the running time of an algorithm increases with

the size of the input in the limit, as the size of the input increases without bound.

Usually, an algorithm that is asymptotically more efﬁcient will be the best choice

for all but very small inputs.

This chapter gives several standard methods for simplifying the asymptotic anal-

ysis of algorithms. The next section begins by deﬁning several types of “asymp-

totic notation,” of which we have already seen an example in ‚-notation. We then

present several notational conventions used throughout this book, and ﬁnally we

review the behavior of functions that commonly arise in the analysis of algorithms.

3.1 Asymptotic notation

The notations we use to describe the asymptotic running time of an algorithm

are deﬁned in terms of functions whose domains are the set of natural numbers

N D f0; 1; 2; : : :g. Such notations are convenient for describing the worst-case

running-time function T .n/, which usually is deﬁned only on integer input sizes.

We sometimes ﬁnd it convenient, however, to abuse asymptotic notation in a va-

44 Chapter 3 Growth of Functions

riety of ways. For example, we might extend the notation to the domain of real

numbers or, alternatively, restrict it to a subset of the natural numbers. We should

make sure, however, to understand the precise meaning of the notation so that when

we abuse, we do not misuse it. This section deﬁnes the basic asymptotic notations

and also introduces some common abuses.

Asymptotic notation, functions, and running times

We will use asymptotic notation primarily to describe the running times of algo-

rithms, as when we wrote that insertion sort’s worst-case running time is ‚.n 2 /.

Asymptotic notation actually applies to functions, however. Recall that we charac-

terized insertion sort’s worst-case running time as an 2 CbnCc, for some constants

a, b, and c. By writing that insertion sort’s running time is ‚.n 2 /, we abstracted

away some details of this function. Because asymptotic notation applies to func-

tions, what we were writing as ‚.n 2 / was the function an 2 C bn C c, which in

that case happened to characterize the worst-case running time of insertion sort.

In this book, the functions to which we apply asymptotic notation will usually

characterize the running times of algorithms. But asymptotic notation can apply to

functions that characterize some other aspect of algorithms (the amount of space

they use, for example), or even to functions that have nothing whatsoever to do

with algorithms.

Even when we use asymptotic notation to apply to the running time of an al-

gorithm, we need to understand which running time we mean. Sometimes we are

interested in the worst-case running time. Often, however, we wish to characterize

the running time no matter what the input. In other words, we often wish to make

a blanket statement that covers all inputs, not just the worst case. We shall see

asymptotic notations that are well suited to characterizing running times no matter

what the input.

‚-notation

In Chapter 2, we found that the worst-case running time of insertion sort is

T .n/ D ‚.n 2 /. Let us deﬁne what this notation means. For a given function g.n/,

we denote by ‚.g.n// the set of functions

‚.g.n// D ff .n/ W there exist positive constants c 1 , c 2 , and n 0 such that

0 c 1 g.n/ f .n/ c 2 g.n/ for all n n 0 g : 1

1

Within set notation, a colon means “such that.”

3.1 Asymptotic notation 45

(b) (c) (a)

n n n

n 0 n 0 n 0

f .n/ D ‚.g.n// f .n/ D O.g.n// f .n/ D .g.n//

f .n/

f .n/

f .n/

cg.n/

cg.n/

c 1 g.n/

c 2 g.n/

Figure 3.1 Graphic examples of the ‚, O, and notations. In each part, the value of n0 shown

is the minimum possible value; any greater value would also work. (a) ‚-notation bounds a func-

tion to within constant factors. We write f .n/ D ‚.g.n// if there exist positive constants n0, c1,

and c2 such that at and to the right of n0, the value of f .n/ always lies between c1g.n/ and c2g.n/

inclusive. (b) O-notation gives an upper bound for a function to within a constant factor. We write

f .n/ D O.g.n// if there are positive constants n0 and c such that at and to the right of n0, the value

of f .n/ always lies on or below cg.n/. (c) -notation gives a lower bound for a function to within

a constant factor. We write f .n/ D .g.n// if there are positive constants n0 and c such that at and

to the right of n0, the value of f .n/ always lies on or above cg.n/.

A function f .n/ belongs to the set ‚.g.n// if there exist positive constants c 1

and c 2 such that it can be “sandwiched” between c 1 g.n/ and c 2 g.n/, for sufﬁ-

ciently large n. Because ‚.g.n// is a set, we could write “f .n/ 2 ‚.g.n//”

to indicate that f .n/ is a member of ‚.g.n//. Instead, we will usually write

“f .n/ D ‚.g.n//” to express the same notion. You might be confused because

we abuse equality in this way, but we shall see later in this section that doing so

has its advantages.

Figure 3.1(a) gives an intuitive picture of functions f .n/ and g.n/, where

f .n/ D ‚.g.n//. For all values of n at and to the right of n 0 , the value of f .n/

lies at or above c 1 g.n/ and at or below c 2 g.n/. In other words, for all n n 0 , the

function f .n/ is equal to g.n/ to within a constant factor. We say that g.n/ is an

asymptotically tight bound for f .n/.

The deﬁnition of ‚.g.n// requires that every member f .n/ 2 ‚.g.n// be

asymptotically nonnegative, that is, that f .n/ be nonnegative whenever n is suf-

ﬁciently large. (An asymptotically positive function is one that is positive for all

sufﬁciently large n.) Consequently, the function g.n/ itself must be asymptotically

nonnegative, or else the set ‚.g.n// is empty. We shall therefore assume that every

function used within ‚-notation is asymptotically nonnegative. This assumption

holds for the other asymptotic notations deﬁned in this chapter as well.

46 Chapter 3 Growth of Functions

In Chapter 2, we introduced an informal notion of ‚-notation that amounted

to throwing away lower-order terms and ignoring the leading coefﬁcient of the

highest-order term. Let us brieﬂy justify this intuition by using the formal deﬁ-

nition to show that

1

2

n 2 3n D ‚.n 2 /. To do so, we must determine positive

constants c 1 , c 2 , and n 0 such that

c 1 n

2

1

2

n

2

3n c 2 n

2

for all n n 0 . Dividing by n 2

yields

c 1

1

2

3

n

c 2 :

We can make the right-hand inequality hold for any value of n 1 by choosing any

constant c 2 1=2. Likewise, we can make the left-hand inequality hold for any

value of n 7 by choosing any constant c 1 1=14. Thus, by choosing c 1 D 1=14,

c 2 D 1=2, and n 0 D 7, we can verify that

1

2

n 2 3n D ‚.n 2 /. Certainly, other

choices for the constants exist, but the important thing is that some choice exists.

Note that these constants depend on the function

1

2

n 2 3n; a different function

belonging to ‚.n 2 / would usually require different constants.

We can also use the formal deﬁnition to verify that 6n 3 ¤ ‚.n 2 /. Suppose

for the purpose of contradiction that c 2 and n 0 exist such that 6n 3 c 2 n 2

for

all n n 0 . But then dividing by n 2

yields n c 2 =6, which cannot possibly hold

for arbitrarily large n, since c 2 is constant.

Intuitively, the lower-order terms of an asymptotically positive function can be

ignored in determining asymptotically tight bounds because they are insigniﬁcant

for large n. When n is large, even a tiny fraction of the highest-order term suf-

ﬁces to dominate the lower-order terms. Thus, setting c 1 to a value that is slightly

smaller than the coefﬁcient of the highest-order term and setting c 2 to a value that

is slightly larger permits the inequalities in the deﬁnition of ‚-notation to be sat-

isﬁed. The coefﬁcient of the highest-order term can likewise be ignored, since it

only changes c 1 and c 2 by a constant factor equal to the coefﬁcient.

As an example, consider any quadratic function f .n/ D an 2 C bn C c, where

a, b, and c are constants and a > 0. Throwing away the lower-order terms and

ignoring the constant yields f .n/ D ‚.n 2 /. Formally, to show the same thing, we

take the constants c 1 D a=4, c 2 D 7a=4, and n 0 D 2 max.jbj =a;

p

jcj =a/. You

may verify that 0 c 1 n 2 an 2 C bn C c c 2 n 2

for all n n 0 . In general,

for any polynomial p.n/ D

P d

iD0

a i n i

, where the a i are constants and a d > 0, we

have p.n/ D ‚.n d / (see Problem 3-1).

Since any constant is a degree-0 polynomial, we can express any constant func-

tion as ‚.n 0 /, or ‚.1/. This latter notation is a minor abuse, however, because the

3.1 Asymptotic notation 47

expression does not indicate what variable is tending to inﬁnity. 2 We shall often

use the notation ‚.1/ to mean either a constant or a constant function with respect

to some variable.

O-notation

The ‚-notation asymptotically bounds a function from above and below. When

we have only an asymptotic upper bound, we use O-notation. For a given func-

tion g.n/, we denote by O.g.n// (pronounced “big-oh of g of n” or sometimes

just “oh of g of n”) the set of functions

O.g.n// D ff .n/ W there exist positive constants c and n 0 such that

0 f .n/ cg.n/ for all n n 0 g :

We use O-notation to give an upper bound on a function, to within a constant

factor. Figure 3.1(b) shows the intuition behind O-notation. For all values n at and

to the right of n 0 , the value of the function f .n/ is on or below cg.n/.

We write f .n/ D O.g.n// to indicate that a function f .n/ is a member of the

set O.g.n//. Note that f .n/ D ‚.g.n// implies f .n/ D O.g.n//, since ‚-

notation is a stronger notion than O-notation. Written set-theoretically, we have

‚.g.n// O.g.n//. Thus, our proof that any quadratic function an 2 C bn C c,

where a > 0, is in ‚.n 2 / also shows that any such quadratic function is in O.n 2 /.

What may be more surprising is that when a > 0, any linear function an C b is

in O.n 2 /, which is easily veriﬁed by taking c D a C jbj and n 0 D max.1; b=a/.

If you have seen O-notation before, you might ﬁnd it strange that we should

write, for example, n D O.n 2 /. In the literature, we sometimes ﬁnd O-notation

informally describing asymptotically tight bounds, that is, what we have deﬁned

using ‚-notation. In this book, however, when we write f .n/ D O.g.n//, we

are merely claiming that some constant multiple of g.n/ is an asymptotic upper

bound on f .n/, with no claim about how tight an upper bound it is. Distinguish-

ing asymptotic upper bounds from asymptotically tight bounds is standard in the

algorithms literature.

Using O-notation, we can often describe the running time of an algorithm

merely by inspecting the algorithm’s overall structure. For example, the doubly

nested loop structure of the insertion sort algorithm from Chapter 2 immediately

yields an O.n 2 / upper bound on the worst-case running time: the cost of each it-

eration of the inner loop is bounded from above by O.1/ (constant), the indices i

2

The real problem is that our ordinary notation for functions does not distinguish functions from

values. In -calculus, the parameters to a function are clearly speciﬁed: the function n

2

could be

written as n:n

2

, or even r:r

2

. Adopting a more rigorous notation, however, would complicate

algebraic manipulations, and so we choose to tolerate the abuse.

48 Chapter 3 Growth of Functions

and j are both at most n, and the inner loop is executed at most once for each of

the n 2

pairs of values for i and j .

Since O-notation describes an upper bound, when we use it to bound the worst-

case running time of an algorithm, we have a bound on the running time of the algo-

rithm on every input—the blanket statement we discussed earlier. Thus, the O.n 2 /

bound on worst-case running time of insertion sort also applies to its running time

on every input. The ‚.n 2 / bound on the worst-case running time of insertion sort,

however, does not imply a ‚.n 2 / bound on the running time of insertion sort on

every input. For example, we saw in Chapter 2 that when the input is already

sorted, insertion sort runs in ‚.n/ time.

Technically, it is an abuse to say that the running time of insertion sort is O.n 2 /,

since for a given n, the actual running time varies, depending on the particular

input of size n. When we say “the running time is O.n 2 /,” we mean that there is a

function f .n/ that is O.n 2 / such that for any value of n, no matter what particular

input of size n is chosen, the running time on that input is bounded from above by

the value f .n/. Equivalently, we mean that the worst-case running time is O.n 2 /.

-notation

Just as O-notation provides an asymptotic upper bound on a function, -notation

provides an asymptotic lower bound. For a given function g.n/, we denote

by .g.n// (pronounced “big-omega of g of n” or sometimes just “omega of g

of n”) the set of functions

.g.n// D ff .n/ W there exist positive constants c and n 0 such that

0 cg.n/ f .n/ for all n n 0 g :

Figure 3.1(c) shows the intuition behind -notation. For all values n at or to the

right of n 0 , the value of f .n/ is on or above cg.n/.

From the deﬁnitions of the asymptotic notations we have seen thus far, it is easy

to prove the following important theorem (see Exercise 3.1-5).

Theorem 3.1

For any two functions f .n/ and g.n/, we have f .n/ D ‚.g.n// if and only if

f .n/ D O.g.n// and f .n/ D .g.n//.

As an example of the application of this theorem, our proof that an 2 C bn C c D

‚.n 2 / for any constants a, b, and c, where a > 0, immediately implies that

an 2 C bn C c D .n 2 / and an 2 CbnCc D O.n 2 /. In practice, rather than using

Theorem 3.1 to obtain asymptotic upper and lower bounds from asymptotically

tight bounds, as we did for this example, we usually use it to prove asymptotically

tight bounds from asymptotic upper and lower bounds.

3.1 Asymptotic notation 49

When we say that the running time (no modiﬁer) of an algorithm is .g.n//,

we mean that no matter what particular input of size n is chosen for each value

of n, the running time on that input is at least a constant times g.n/, for sufﬁciently

large n. Equivalently, we are giving a lower bound on the best-case running time

of an algorithm. For example, the best-case running time of insertion sort is .n/,

which implies that the running time of insertion sort is .n/.

The running time of insertion sort therefore belongs to both .n/ and O.n 2 /,

since it falls anywhere between a linear function of n and a quadratic function of n.

Moreover, these bounds are asymptotically as tight as possible: for instance, the

running time of insertion sort is not .n 2 /, since there exists an input for which

insertion sort runs in ‚.n/ time (e.g., when the input is already sorted). It is not

contradictory, however, to say that the worst-case running time of insertion sort

is .n 2 /, since there exists an input that causes the algorithm to take .n 2 / time.

Asymptotic notation in equations and inequalities

We have already seen how asymptotic notation can be used within mathematical

formulas. For example, in introducing O-notation, we wrote “n D O.n 2 /.” We

might also write 2n 2 C3nC1 D 2n 2 C‚.n/. How do we interpret such formulas?

When the asymptotic notation stands alone (that is, not within a larger formula)

on the right-hand side of an equation (or inequality), as in n D O.n 2 /, we have

already deﬁned the equal sign to mean set membership: n 2 O.n 2 /. In general,

however, when asymptotic notation appears in a formula, we interpret it as stand-

ing for some anonymous function that we do not care to name. For example, the

formula 2n 2 C 3n C 1 D 2n 2 C ‚.n/ means that 2n 2 C 3n C 1 D 2n 2 C f .n/,

where f .n/ is some function in the set ‚.n/. In this case, we let f .n/ D 3n C 1,

which indeed is in ‚.n/.

Using asymptotic notation in this manner can help eliminate inessential detail

and clutter in an equation. For example, in Chapter 2 we expressed the worst-case

running time of merge sort as the recurrence

T .n/ D 2T .n=2/ C ‚.n/ :

If we are interested only in the asymptotic behavior of T .n/, there is no point in

specifying all the lower-order terms exactly; they are all understood to be included

in the anonymous function denoted by the term ‚.n/.

The number of anonymous functions in an expression is understood to be equal

to the number of times the asymptotic notation appears. For example, in the ex-

pression

n X

iD1

O.i/ ;

50 Chapter 3 Growth of Functions

there is only a single anonymous function (a function of i). This expression is thus

not the same as O.1/ C O.2/ C C O.n/, which doesn’t really have a clean

interpretation.

In some cases, asymptotic notation appears on the left-hand side of an equation,

as in

2n

2

C ‚.n/ D ‚.n

2

/ :

We interpret such equations using the following rule: No matter how the anony-

mous functions are chosen on the left of the equal sign, there is a way to choose

the anonymous functions on the right of the equal sign to make the equation valid.

Thus, our example means that for any function f .n/ 2 ‚.n/, there is some func-

tion g.n/ 2 ‚.n 2 / such that 2n 2 C f .n/ D g.n/ for all n. In other words, the

right-hand side of an equation provides a coarser level of detail than the left-hand

side.

We can chain together a number of such relationships, as in

2n

2

C 3n C 1 D 2n

2

C ‚.n/

D ‚.n

2

/ :

We can interpret each equation separately by the rules above. The ﬁrst equa-

tion says that there is some function f .n/ 2 ‚.n/ such that 2n 2 C 3n C 1 D

2n 2 C f .n/ for all n. The second equation says that for any function g.n/ 2 ‚.n/

(such as the f .n/ just mentioned), there is some function h.n/ 2 ‚.n 2 / such

that 2n 2 C g.n/ D h.n/ for all n. Note that this interpretation implies that

2n 2 C 3n C 1 D ‚.n 2 /, which is what the chaining of equations intuitively gives

us.

o-notation

The asymptotic upper bound provided by O-notation may or may not be asymp-

totically tight. The bound 2n 2 D O.n 2 / is asymptotically tight, but the bound

2n D O.n 2 / is not. We use o-notation to denote an upper bound that is not asymp-

totically tight. We formally deﬁne o.g.n// (“little-oh of g of n”) as the set

o.g.n// D ff .n/ W for any positive constant c > 0, there exists a constant

n 0 > 0 such that 0 f .n/ < cg.n/ for all n n 0 g :

For example, 2n D o.n 2 /, but 2n 2 ¤ o.n 2 /.

The deﬁnitions of O-notation and o-notation are similar. The main difference

is that in f .n/ D O.g.n//, the bound 0 f .n/ cg.n/ holds for some con-

stant c > 0, but in f .n/ D o.g.n//, the bound 0 f .n/ < cg.n/ holds for all

constants c > 0. Intuitively, in o-notation, the function f .n/ becomes insigniﬁcant

relative to g.n/ as n approaches inﬁnity; that is,

3.1 Asymptotic notation 51

lim

n!1

f .n/

g.n/

D 0 : (3.1)

Some authors use this limit as a deﬁnition of the o-notation; the deﬁnition in this

book also restricts the anonymous functions to be asymptotically nonnegative.

!-notation

By analogy, !-notation is to -notation as o-notation is to O-notation. We use

!-notation to denote a lower bound that is not asymptotically tight. One way to

deﬁne it is by

f .n/ 2 !.g.n// if and only if g.n/ 2 o.f .n// :

Formally, however, we deﬁne !.g.n// (“little-omega of g of n”) as the set

!.g.n// D ff .n/ W for any positive constant c > 0, there exists a constant

n 0 > 0 such that 0 cg.n/ < f .n/ for all n n 0 g :

For example, n 2 =2 D !.n/, but n 2 =2 ¤ !.n 2 /. The relation f .n/ D !.g.n//

implies that

lim

n!1

f .n/

g.n/

D 1 ;

if the limit exists. That is, f .n/ becomes arbitrarily large relative to g.n/ as n

approaches inﬁnity.

Comparing functions

Many of the relational properties of real numbers apply to asymptotic comparisons

as well. For the following, assume that f .n/ and g.n/ are asymptotically positive.

Transitivity:

f .n/ D ‚.g.n// and g.n/ D ‚.h.n// imply f .n/ D ‚.h.n// ;

f .n/ D O.g.n// and g.n/ D O.h.n// imply f .n/ D O.h.n// ;

f .n/ D .g.n// and g.n/ D .h.n// imply f .n/ D .h.n// ;

f .n/ D o.g.n// and g.n/ D o.h.n// imply f .n/ D o.h.n// ;

f .n/ D !.g.n// and g.n/ D !.h.n// imply f .n/ D !.h.n// :

Reﬂexivity:

f .n/ D ‚.f .n// ;

f .n/ D O.f .n// ;

f .n/ D .f .n// :

52 Chapter 3 Growth of Functions

Symmetry:

f .n/ D ‚.g.n// if and only if g.n/ D ‚.f .n// :

Transpose symmetry:

f .n/ D O.g.n// if and only if g.n/ D .f .n// ;

f .n/ D o.g.n// if and only if g.n/ D !.f .n// :

Because these properties hold for asymptotic notations, we can draw an analogy

between the asymptotic comparison of two functions f and g and the comparison

of two real numbers a and b:

f .n/ D O.g.n// is like a b ;

f .n/ D .g.n// is like a b ;

f .n/ D ‚.g.n// is like a D b ;

f .n/ D o.g.n// is like a < b ;

f .n/ D !.g.n// is like a > b :

We say that f .n/ is asymptotically smaller than g.n/ if f .n/ D o.g.n//, and f .n/

is asymptotically larger than g.n/ if f .n/ D !.g.n//.

One property of real numbers, however, does not carry over to asymptotic nota-

tion:

Trichotomy: For any two real numbers a and b, exactly one of the following must

hold: a < b, a D b, or a > b.

Although any two real numbers can be compared, not all functions are asymptot-

ically comparable. That is, for two functions f .n/ and g.n/, it may be the case

that neither f .n/ D O.g.n// nor f .n/ D .g.n// holds. For example, we cannot

compare the functions n and n 1Csin n

using asymptotic notation, since the value of

the exponent in n 1Csin n

oscillates between 0 and 2, taking on all values in between.

Exercises

3.1-1

Let f .n/ and g.n/ be asymptotically nonnegative functions. Using the basic deﬁ-

nition of ‚-notation, prove that max.f .n/; g.n// D ‚.f .n/ C g.n//.

3.1-2

Show that for any real constants a and b, where b > 0,

.n C a/

b

D ‚.n

b

/ : (3.2)

3.2 Standard notations and common functions 53

3.1-3

Explain why the statement, “The running time of algorithm A is at least O.n 2 /,” is

meaningless.

3.1-4

Is 2 nC1 D O.2 n /? Is 2 2n D O.2 n /?

3.1-5

Prove Theorem 3.1.

3.1-6

Prove that the running time of an algorithm is ‚.g.n// if and only if its worst-case

running time is O.g.n// and its best-case running time is .g.n//.

3.1-7

Prove that o.g.n// \ !.g.n// is the empty set.

3.1-8

We can extend our notation to the case of two parameters n and m that can go to

inﬁnity independently at different rates. For a given function g.n; m/, we denote

by O.g.n; m// the set of functions

O.g.n; m// D ff .n; m/ W there exist positive constants c, n 0 , and m 0

such that 0 f .n; m/ cg.n; m/

for all n n 0 or m m 0 g :

Give corresponding deﬁnitions for .g.n; m// and ‚.g.n; m//.

3.2 Standard notations and common functions

This section reviews some standard mathematical functions and notations and ex-

plores the relationships among them. It also illustrates the use of the asymptotic

notations.

Monotonicity

A function f .n/ is monotonically increasing if m n implies f .m/ f .n/.

Similarly, it is monotonically decreasing if m n implies f .m/ f .n/. A

function f .n/ is strictly increasing if m < n implies f .m/ < f .n/ and strictly

decreasing if m < n implies f .m/ > f .n/.

54 Chapter 3 Growth of Functions

Floors and ceilings

For any real number x, we denote the greatest integer less than or equal to x by bxc

(read “the ﬂoor of x”) and the least integer greater than or equal to x by dxe (read

“the ceiling of x”). For all real x,

x 1 < bxc x dxe < x C 1 : (3.3)

For any integer n,

dn=2e C bn=2c D n ;

and for any real number x 0 and integers a; b > 0,

dx=ae

b

D

l

x

ab

m

; (3.4)

bx=ac

b

D

j

x

ab

k

; (3.5)

l

a

b

m

a C .b 1/

b

; (3.6)

j

a

b

k

a .b 1/

b

: (3.7)

The ﬂoor function f .x/ D bxc is monotonically increasing, as is the ceiling func-

tion f .x/ D dxe.

Modular arithmetic

For any integer a and any positive integer n, the value a mod n is the remainder

(or residue) of the quotient a=n:

a mod n D a n ba=nc : (3.8)

It follows that

0 a mod n < n : (3.9)

Given a well-deﬁned notion of the remainder of one integer when divided by an-

other, it is convenient to provide special notation to indicate equality of remainders.

If .a mod n/ D .b mod n/, we write a b .mod n/ and say that a is equivalent

to b, modulo n. In other words, a b .mod n/ if a and b have the same remain-

der when divided by n. Equivalently, a b .mod n/ if and only if n is a divisor

of b a. We write a 6 b .mod n/ if a is not equivalent to b, modulo n.

3.2 Standard notations and common functions 55

Polynomials

Given a nonnegative integer d, a polynomial in n of degree d is a function p.n/

of the form

p.n/ D

d X

iD0

a i n

i

;

where the constants a 0 ; a 1 ; : : : ; a d are the coefﬁcients of the polynomial and

a d ¤ 0. A polynomial is asymptotically positive if and only if a d > 0. For an

asymptotically positive polynomial p.n/ of degree d, we have p.n/ D ‚.n d /. For

any real constant a 0, the function n a

is monotonically increasing, and for any

real constant a 0, the function n a

is monotonically decreasing. We say that a

function f .n/ is polynomially bounded if f .n/ D O.n k / for some constant k.

Exponentials

For all real a > 0, m, and n, we have the following identities:

a

0

D 1 ;

a

1

D a ;

a

1

D 1=a ;

.a

m

/

n

D a

mn

;

.a

m

/

n

D .a

n

/

m

;

a

m

a

n

D a

mCn

:

For all n and a 1, the function a n

is monotonically increasing in n. When

convenient, we shall assume 0 0 D 1.

We can relate the rates of growth of polynomials and exponentials by the fol-

lowing fact. For all real constants a and b such that a > 1,

lim

n!1

n b

a n

D 0 ; (3.10)

from which we can conclude that

n

b

D o.a

n

/ :

Thus, any exponential function with a base strictly greater than 1 grows faster than

any polynomial function.

Using e to denote 2:71828 : : :, the base of the natural logarithm function, we

have for all real x,

e

x

D 1 C x C

x 2

2Š

C

x 3

3Š

C D

1 X

iD0

x i

iŠ

; (3.11)

56 Chapter 3 Growth of Functions

where “Š” denotes the factorial function deﬁned later in this section. For all real x,

we have the inequality

e

x

1 C x ; (3.12)

where equality holds only when x D 0. When jxj 1, we have the approximation

1 C x e

x

1 C x C x

2

: (3.13)

When x ! 0, the approximation of e x

by 1 C x is quite good:

e

x

D 1 C x C ‚.x

2

/ :

(In this equation, the asymptotic notation is used to describe the limiting behavior

as x ! 0 rather than as x ! 1.) We have for all x,

lim

n!1

1 C

x

n

n

D e

x

: (3.14)

Logarithms

We shall use the following notations:

lg n D log 2 n (binary logarithm) ,

ln n D log e n (natural logarithm) ,

lg

k

n D .lg n/

k

(exponentiation) ,

lg lg n D lg.lg n/ (composition) .

An important notational convention we shall adopt is that logarithm functions will

apply only to the next term in the formula, so that lg n C k will mean .lg n/ C k

and not lg.n C k/. If we hold b > 1 constant, then for n > 0, the function log

b

n

is strictly increasing.

For all real a > 0, b > 0, c > 0, and n,

a D b log b a

;

log

c

.ab/ D log

c

a C log

c

b ;

log

b

a

n

D n log

b

a ;

log b a D

log c a

log

c

b

; (3.15)

log

b

.1=a/ D log

b

a ;

log

b

a D

1

log

a

b

;

a log b c

D c log b a

; (3.16)

where, in each equation above, logarithm bases are not 1.

3.2 Standard notations and common functions 57

By equation (3.15), changing the base of a logarithm from one constant to an-

other changes the value of the logarithm by only a constant factor, and so we shall

often use the notation “lg n” when we don’t care about constant factors, such as in

O-notation. Computer scientists ﬁnd 2 to be the most natural base for logarithms

because so many algorithms and data structures involve splitting a problem into

two parts.

There is a simple series expansion for ln.1 C x/ when jxj < 1:

ln.1 C x/ D x

x 2

2

C

x 3

3

x 4

4

C

x 5

5

:

We also have the following inequalities for x > 1:

x

1 C x

ln.1 C x/ x ; (3.17)

where equality holds only for x D 0.

We say that a function f .n/ is polylogarithmically bounded if f .n/ D O.lg

k

n/

for some constant k. We can relate the growth of polynomials and polylogarithms

by substituting lg n for n and 2 a

for a in equation (3.10), yielding

lim

n!1

lg

b

n

.2 a / lg n

D lim

n!1

lg

b

n

n a

D 0 :

From this limit, we can conclude that

lg

b

n D o.n

a

/

for any constant a > 0. Thus, any positive polynomial function grows faster than

any polylogarithmic function.

Factorials

The notation nŠ (read “n factorial”) is deﬁned for integers n 0 as

nŠ D

(

1 if n D 0 ;

n .n 1/Š if n > 0 :

Thus, nŠ D 1 2 3 n.

A weak upper bound on the factorial function is nŠ n n

, since each of the n

terms in the factorial product is at most n. Stirling’s approximation,

nŠ D

p

2n

n

e

n

1 C ‚

1

n

; (3.18)

58 Chapter 3 Growth of Functions

where e is the base of the natural logarithm, gives us a tighter upper bound, and a

lower bound as well. As Exercise 3.2-3 asks you to prove,

nŠ D o.n

n

/ ;

nŠ D !.2

n

/ ;

lg.nŠ/ D ‚.n lg n/ ; (3.19)

where Stirling’s approximation is helpful in proving equation (3.19). The following

equation also holds for all n 1:

nŠ D

p

2n

n

e

n

e

˛ n (3.20)

where

1

12n C 1

< ˛ n <

1

12n

: (3.21)

Functional iteration

We use the notation f .i/ .n/ to denote the function f .n/ iteratively applied i times

to an initial value of n. Formally, let f .n/ be a function over the reals. For non-

negative integers i, we recursively deﬁne

f

.i/

.n/ D

(

n if i D 0 ;

f .f .i1/ .n// if i > 0 :

For example, if f .n/ D 2n, then f .i/ .n/ D 2 i n.

The iterated logarithm function

We use the notation lg

n (read “log star of n”) to denote the iterated logarithm, de-

ﬁned as follows. Let lg

.i/

n be as deﬁned above, with f .n/ D lg n. Because the log-

arithm of a nonpositive number is undeﬁned, lg

.i/

n is deﬁned only if lg

.i1/

n > 0.

Be sure to distinguish lg

.i/

n (the logarithm function applied i times in succession,

starting with argument n) from lg

i

n (the logarithm of n raised to the ith power).

Then we deﬁne the iterated logarithm function as

lg

n D min

˚

i 0 W lg

.i/

n 1

:

The iterated logarithm is a very slowly growing function:

lg

2 D 1 ;

lg

4 D 2 ;

lg

16 D 3 ;

lg

65536 D 4 ;

lg

.2

65536

/ D 5 :

3.2 Standard notations and common functions 59

Since the number of atoms in the observable universe is estimated to be about 10 80

,

which is much less than 2 65536

, we rarely encounter an input size n such that

lg

n > 5.

Fibonacci numbers

We deﬁne the Fibonacci numbers by the following recurrence:

F 0 D 0 ;

F 1 D 1 ; (3.22)

F i D F i1 C F i2 for i 2 :

Thus, each Fibonacci number is the sum of the two previous ones, yielding the

sequence

0; 1; 1; 2; 3; 5; 8; 13; 21; 34; 55; : : : :

Fibonacci numbers are related to the golden ratio and to its conjugate y, which

are the two roots of the equation

x

2

D x C 1 (3.23)

and are given by the following formulas (see Exercise 3.2-6):

D

1 C

p

5

2

(3.24)

D 1:61803 : : : ;

y D

1

p

5

2

D :61803 : : : :

Speciﬁcally, we have

F i D

i y i

p

5

;

which we can prove by induction (Exercise 3.2-7). Since

ˇ

ˇ y

ˇ

ˇ < 1, we have

ˇ

ˇ y i

ˇ

ˇ

p

5

<

1

p

5

<

1

2

;

which implies that

60 Chapter 3 Growth of Functions

F i D

i

p

5

C

1

2

; (3.25)

which is to say that the ith Fibonacci number F i is equal to i =

p

5 rounded to the

nearest integer. Thus, Fibonacci numbers grow exponentially.

Exercises

3.2-1

Show that if f .n/ and g.n/ are monotonically increasing functions, then so are

the functions f .n/ C g.n/ and f .g.n//, and if f .n/ and g.n/ are in addition

nonnegative, then f .n/ g.n/ is monotonically increasing.

3.2-2

Prove equation (3.16).

3.2-3

Prove equation (3.19). Also prove that nŠ D !.2 n / and nŠ D o.n n /.

3.2-4 ?

Is the function dlg neŠ polynomially bounded? Is the function dlg lg neŠ polynomi-

ally bounded?

3.2-5 ?

Which is asymptotically larger: lg.lg

n/ or lg

.lg n/?

3.2-6

Show that the golden ratio and its conjugate y both satisfy the equation

x 2 D x C 1.

3.2-7

Prove by induction that the ith Fibonacci number satisﬁes the equality

F i D

i y i

p

5

;

where is the golden ratio and y is its conjugate.

3.2-8

Show that k ln k D ‚.n/ implies k D ‚.n= ln n/.

Problems for Chapter 3 61

Problems

3-1 Asymptotic behavior of polynomials

Let

p.n/ D

d X

iD0

a i n

i

;

where a d > 0, be a degree-d polynomial in n, and let k be a constant. Use the

deﬁnitions of the asymptotic notations to prove the following properties.

a. If k d, then p.n/ D O.n k /.

b. If k d, then p.n/ D .n k /.

c. If k D d, then p.n/ D ‚.n k /.

d. If k > d, then p.n/ D o.n k /.

e. If k < d, then p.n/ D !.n k /.

3-2 Relative asymptotic growths

Indicate, for each pair of expressions .A; B/ in the table below, whether A is O, o,

, !, or ‚ of B. Assume that k 1, > 0, and c > 1 are constants. Your answer

should be in the form of the table with “yes” or “no” written in each box.

A B O o ! ‚

a. lg

k

n n

b. n k c n

c.

p

n n sin n

d. 2 n 2 n=2

e. n lg c c lg n

f. lg.nŠ/ lg.n n /

3-3 Ordering by asymptotic growth rates

a. Rank the following functions by order of growth; that is, ﬁnd an arrangement

g 1 ; g 2 ; : : : ; g 30 of the functions satisfying g 1 D .g 2 /, g 2 D .g 3 /, . . . ,

g 29 D .g 30 /. Partition your list into equivalence classes such that functions

f .n/ and g.n/ are in the same class if and only if f .n/ D ‚.g.n//.

62 Chapter 3 Growth of Functions

lg.lg

n/ 2 lg

n .

p

2/ lg n n 2 nŠ .lg n/Š

.

3

2

/ n n 3

lg

2

n lg.nŠ/ 2 2

n

n 1= lg n

ln ln n lg

n n 2 n n lg lg n

ln n 1

2 lg n .lg n/ lg n e n 4 lg n .n C 1/Š

p

lg n

lg

.lg n/ 2

p

2 lg n n 2 n n lg n 2 2

nC1

b. Give an example of a single nonnegative function f .n/ such that for all func-

tions g i .n/ in part (a), f .n/ is neither O.g i .n// nor .g i .n//.

3-4 Asymptotic notation properties

Let f .n/ and g.n/ be asymptotically positive functions. Prove or disprove each of

the following conjectures.

a. f .n/ D O.g.n// implies g.n/ D O.f .n//.

b. f .n/ C g.n/ D ‚.min.f .n/; g.n///.

c. f .n/ D O.g.n// implies lg.f .n// D O.lg.g.n///, where lg.g.n// 1 and

f .n/ 1 for all sufﬁciently large n.

d. f .n/ D O.g.n// implies 2 f .n/ D O

2 g.n/

.

e. f .n/ D O ..f .n// 2 /.

f. f .n/ D O.g.n// implies g.n/ D .f .n//.

g. f .n/ D ‚.f .n=2//.

h. f .n/ C o.f .n// D ‚.f .n//.

3-5 Variations on O and ˝

Some authors deﬁne in a slightly different way than we do; let’s use

1

(read

“omega inﬁnity”) for this alternative deﬁnition. We say that f .n/ D

1

.g.n// if

there exists a positive constant c such that f .n/ cg.n/ 0 for inﬁnitely many

integers n.

a. Show that for any two functions f .n/ and g.n/ that are asymptotically nonneg-

ative, either f .n/ D O.g.n// or f .n/ D

1

.g.n// or both, whereas this is not

true if we use in place of

1

.

Problems for Chapter 3 63

b. Describe the potential advantages and disadvantages of using

1

instead of to

characterize the running times of programs.

Some authors also deﬁne O in a slightly different manner; let’s use O 0

for the

alternative deﬁnition. We say that f .n/ D O 0 .g.n// if and only if jf .n/j D

O.g.n//.

c. What happens to each direction of the “if and only if” in Theorem 3.1 if we

substitute O 0

for O but still use ?

Some authors deﬁne e O (read “soft-oh”) to mean O with logarithmic factors ig-

nored:

e O.g.n// D ff .n/ W there exist positive constants c, k, and n 0 such that

0 f .n/ cg.n/ lg

k

.n/ for all n n 0 g :

d. Deﬁne e and e ‚ in a similar manner. Prove the corresponding analog to Theo-

rem 3.1.

3-6 Iterated functions

We can apply the iteration operator

used in the lg

function to any monotonically

increasing function f .n/ over the reals. For a given constant c 2 R , we deﬁne the

iterated function f

c

by

f

c

.n/ D min

˚

i 0 W f

.i/

.n/ c

;

which need not be well deﬁned in all cases. In other words, the quantity f

c

.n/ is

the number of iterated applications of the function f required to reduce its argu-

ment down to c or less.

For each of the following functions f .n/ and constants c, give as tight a bound

as possible on f

c

.n/.

f .n/ c f

c

.n/

a. n 1 0

b. lg n 1

c. n=2 1

d. n=2 2

e.

p

n 2

f.

p

n 1

g. n 1=3 2

h. n= lg n 2

64 Chapter 3 Growth of Functions

Chapter notes

Knuth [209] traces the origin of the O-notation to a number-theory text by P. Bach-

mann in 1892. The o-notation was invented by E. Landau in 1909 for his discussion

of the distribution of prime numbers. The and ‚ notations were advocated by

Knuth [213] to correct the popular, but technically sloppy, practice in the literature

of using O-notation for both upper and lower bounds. Many people continue to

use the O-notation where the ‚-notation is more technically precise. Further dis-

cussion of the history and development of asymptotic notations appears in works

by Knuth [209, 213] and Brassard and Bratley [54].

Not all authors deﬁne the asymptotic notations in the same way, although the

various deﬁnitions agree in most common situations. Some of the alternative def-

initions encompass functions that are not asymptotically nonnegative, as long as

their absolute values are appropriately bounded.

Equation (3.20) is due to Robbins [297]. Other properties of elementary math-

ematical functions can be found in any good mathematical reference, such as

Abramowitz and Stegun [1] or Zwillinger [362], or in a calculus book, such as

Apostol [18] or Thomas et al. [334]. Knuth [209] and Graham, Knuth, and Patash-

nik [152] contain a wealth of material on discrete mathematics as used in computer

science.

4 Divide-and-Conquer

In Section 2.3.1, we saw how merge sort serves as an example of the divide-and-

conquer paradigm. Recall that in divide-and-conquer, we solve a problem recur-

sively, applying three steps at each level of the recursion:

Divide the problem into a number of subproblems that are smaller instances of the

same problem.

Conquer the subproblems by solving them recursively. If the subproblem sizes are

small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original prob-

lem.

When the subproblems are large enough to solve recursively, we call that the recur-

sive case. Once the subproblems become small enough that we no longer recurse,

we say that the recursion “bottoms out” and that we have gotten down to the base

case. Sometimes, in addition to subproblems that are smaller instances of the same

problem, we have to solve subproblems that are not quite the same as the original

problem. We consider solving such subproblems as part of the combine step.

In this chapter, we shall see more algorithms based on divide-and-conquer. The

ﬁrst one solves the maximum-subarray problem: it takes as input an array of num-

bers, and it determines the contiguous subarray whose values have the greatest sum.

Then we shall see two divide-and-conquer algorithms for multiplying n n matri-

ces. One runs in ‚.n 3 / time, which is no better than the straightforward method of

multiplying square matrices. But the other, Strassen’s algorithm, runs in O.n 2:81 /

time, which beats the straightforward method asymptotically.

Recurrences

Recurrences go hand in hand with the divide-and-conquer paradigm, because they

give us a natural way to characterize the running times of divide-and-conquer algo-

rithms. A recurrence is an equation or inequality that describes a function in terms

66 Chapter 4 Divide-and-Conquer

of its value on smaller inputs. For example, in Section 2.3.2 we described the

worst-case running time T .n/ of the MERGE-SORT procedure by the recurrence

T .n/ D

(

‚.1/ if n D 1 ;

2T .n=2/ C ‚.n/ if n > 1 ;

(4.1)

whose solution we claimed to be T .n/ D ‚.n lg n/.

Recurrences can take many forms. For example, a recursive algorithm might

divide subproblems into unequal sizes, such as a 2=3-to-1=3 split. If the divide and

combine steps take linear time, such an algorithm would give rise to the recurrence

T .n/ D T .2n=3/ C T .n=3/ C ‚.n/.

Subproblems are not necessarily constrained to being a constant fraction of

the original problem size. For example, a recursive version of linear search

(see Exercise 2.1-3) would create just one subproblem containing only one el-

ement fewer than the original problem. Each recursive call would take con-

stant time plus the time for the recursive calls it makes, yielding the recurrence

T .n/ D T .n 1/ C ‚.1/.

This chapter offers three methods for solving recurrences—that is, for obtaining

asymptotic “‚” or “O” bounds on the solution:

In the substitution method, we guess a bound and then use mathematical in-

duction to prove our guess correct.

The recursion-tree method converts the recurrence into a tree whose nodes

represent the costs incurred at various levels of the recursion. We use techniques

for bounding summations to solve the recurrence.

The master method provides bounds for recurrences of the form

T .n/ D aT .n=b/ C f .n/ ; (4.2)

where a 1, b > 1, and f .n/ is a given function. Such recurrences arise

frequently. A recurrence of the form in equation (4.2) characterizes a divide-

and-conquer algorithm that creates a subproblems, each of which is 1=b the

size of the original problem, and in which the divide and combine steps together

take f .n/ time.

To use the master method, you will need to memorize three cases, but once

you do that, you will easily be able to determine asymptotic bounds for many

simple recurrences. We will use the master method to determine the running

times of the divide-and-conquer algorithms for the maximum-subarray problem

and for matrix multiplication, as well as for other algorithms based on divide-

and-conquer elsewhere in this book.

Chapter 4 Divide-and-Conquer 67

Occasionally, we shall see recurrences that are not equalities but rather inequal-

ities, such as T .n/ 2T .n=2/ C ‚.n/. Because such a recurrence states only

an upper bound on T .n/, we will couch its solution using O-notation rather than

‚-notation. Similarly, if the inequality were reversed to T .n/ 2T .n=2/ C ‚.n/,

then because the recurrence gives only a lower bound on T .n/, we would use

-notation in its solution.

Technicalities in recurrences

In practice, we neglect certain technical details when we state and solve recur-

rences. For example, if we call MERGE-SORT on n elements when n is odd, we

end up with subproblems of size bn=2c and dn=2e. Neither size is actually n=2,

because n=2 is not an integer when n is odd. Technically, the recurrence describing

the worst-case running time of MERGE-SORT is really

T .n/ D

(

‚.1/ if n D 1 ;

T .dn=2e/ C T .bn=2c/ C ‚.n/ if n > 1 :

(4.3)

Boundary conditions represent another class of details that we typically ignore.

Since the running time of an algorithm on a constant-sized input is a constant,

the recurrences that arise from the running times of algorithms generally have

T .n/ D ‚.1/ for sufﬁciently small n. Consequently, for convenience, we shall

generally omit statements of the boundary conditions of recurrences and assume

that T .n/ is constant for small n. For example, we normally state recurrence (4.1)

as

T .n/ D 2T .n=2/ C ‚.n/ ; (4.4)

without explicitly giving values for small n. The reason is that although changing

the value of T .1/ changes the exact solution to the recurrence, the solution typi-

cally doesn’t change by more than a constant factor, and so the order of growth is

unchanged.

When we state and solve recurrences, we often omit ﬂoors, ceilings, and bound-

ary conditions. We forge ahead without these details and later determine whether

or not they matter. They usually do not, but you should know when they do. Ex-

perience helps, and so do some theorems stating that these details do not affect the

asymptotic bounds of many recurrences characterizing divide-and-conquer algo-

rithms (see Theorem 4.1). In this chapter, however, we shall address some of these

details and illustrate the ﬁne points of recurrence solution methods.

68 Chapter 4 Divide-and-Conquer

4.1 The maximum-subarray problem

Suppose that you been offered the opportunity to invest in the Volatile Chemical

Corporation. Like the chemicals the company produces, the stock price of the

Volatile Chemical Corporation is rather volatile. You are allowed to buy one unit

of stock only one time and then sell it at a later date, buying and selling after the

close of trading for the day. To compensate for this restriction, you are allowed to

learn what the price of the stock will be in the future. Your goal is to maximize

your proﬁt. Figure 4.1 shows the price of the stock over a 17-day period. You

may buy the stock at any one time, starting after day 0, when the price is $100

per share. Of course, you would want to “buy low, sell high”—buy at the lowest

possible price and later on sell at the highest possible price—to maximize your

proﬁt. Unfortunately, you might not be able to buy at the lowest price and then sell

at the highest price within a given period. In Figure 4.1, the lowest price occurs

after day 7, which occurs after the highest price, after day 1.

You might think that you can always maximize proﬁt by either buying at the

lowest price or selling at the highest price. For example, in Figure 4.1, we would

maximize proﬁt by buying at the lowest price, after day 7. If this strategy always

worked, then it would be easy to determine how to maximize proﬁt: ﬁnd the highest

and lowest prices, and then work left from the highest price to ﬁnd the lowest prior

price, work right from the lowest price to ﬁnd the highest later price, and take

the pair with the greater difference. Figure 4.2 shows a simple counterexample,

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

120

110

100

90

80

70

60

Day 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Price 100 113 110 85 105 102 86 63 81 101 94 106 101 79 94 90 97

Change 13 3 25 20 3 16 23 18 20 7 12 5 22 15 4 7

Figure 4.1 Information about the price of stock in the Volatile Chemical Corporation after the close

of trading over a period of 17 days. The horizontal axis of the chart indicates the day, and the vertical

axis shows the price. The bottom row of the table gives the change in price from the previous day.

4.1 The maximum-subarray problem 69

0 1 2 3 4

11

10

9

8

7

6

Day 0 1 2 3 4

Price 10 11 7 10 6

Change 1 4 3 4

Figure 4.2 An example showing that the maximum proﬁt does not always start at the lowest price

or end at the highest price. Again, the horizontal axis indicates the day, and the vertical axis shows

the price. Here, the maximum proﬁt of $3 per share would be earned by buying after day 2 and

selling after day 3. The price of $7 after day 2 is not the lowest price overall, and the price of $10

after day 3 is not the highest price overall.

demonstrating that the maximum proﬁt sometimes comes neither by buying at the

lowest price nor by selling at the highest price.

A brute-force solution

We can easily devise a brute-force solution to this problem: just try every possible

pair of buy and sell dates in which the buy date precedes the sell date. A period of n

days has

n

2

such pairs of dates. Since

n

2

is ‚.n 2 /, and the best we can hope for

is to evaluate each pair of dates in constant time, this approach would take .n 2 /

time. Can we do better?

A transformation

In order to design an algorithm with an o.n 2 / running time, we will look at the

input in a slightly different way. We want to ﬁnd a sequence of days over which

the net change from the ﬁrst day to the last is maximum. Instead of looking at the

daily prices, let us instead consider the daily change in price, where the change on

day i is the difference between the prices after day i 1 and after day i. The table

in Figure 4.1 shows these daily changes in the bottom row. If we treat this row as

an array A, shown in Figure 4.3, we now want to ﬁnd the nonempty, contiguous

subarray of A whose values have the largest sum. We call this contiguous subarray

the maximum subarray. For example, in the array of Figure 4.3, the maximum

subarray of AŒ1 : : 16 is AŒ8 : : 11, with the sum 43. Thus, you would want to buy

the stock just before day 8 (that is, after day 7) and sell it after day 11, earning a

proﬁt of $43 per share.

At ﬁrst glance, this transformation does not help. We still need to check   
 n1

2

D ‚.n 2 / subarrays for a period of n days. Exercise 4.1-2 asks you to show

70 Chapter 4 Divide-and-Conquer

13

1

–3

2

–25

3

20

4

–3

5

–16

6

–23

7 8 9 10

maximum subarray

11

18

12

20

13

–7

14

12

15

7

16

–5 –22 15 –4 A

Figure 4.3 The change in stock prices as a maximum-subarray problem. Here, the subar-

ray AŒ8 : : 11, with sum 43, has the greatest sum of any contiguous subarray of array A.

that although computing the cost of one subarray might take time proportional to

the length of the subarray, when computing all ‚.n 2 / subarray sums, we can orga-

nize the computation so that each subarray sum takes O.1/ time, given the values

of previously computed subarray sums, so that the brute-force solution takes ‚.n 2 /

time.

So let us seek a more efﬁcient solution to the maximum-subarray problem.

When doing so, we will usually speak of “a” maximum subarray rather than “the”

maximum subarray, since there could be more than one subarray that achieves the

maximum sum.

The maximum-subarray problem is interesting only when the array contains

some negative numbers. If all the array entries were nonnegative, then the

maximum-subarray problem would present no challenge, since the entire array

would give the greatest sum.

A solution using divide-and-conquer

Let’s think about how we might solve the maximum-subarray problem using

the divide-and-conquer technique. Suppose we want to ﬁnd a maximum subar-

ray of the subarray AŒlow : : high. Divide-and-conquer suggests that we divide

the subarray into two subarrays of as equal size as possible. That is, we ﬁnd

the midpoint, say mid, of the subarray, and consider the subarrays AŒlow : : mid

and AŒmid C 1 : : high. As Figure 4.4(a) shows, any contiguous subarray AŒi : : j

of AŒlow : : high must lie in exactly one of the following places:

entirely in the subarray AŒlow : : mid, so that low i j mid,

entirely in the subarray AŒmid C 1 : : high, so that mid < i j high, or

crossing the midpoint, so that low i mid < j high.

Therefore, a maximum subarray of AŒlow : : high must lie in exactly one of these

places. In fact, a maximum subarray of AŒlow : : high must have the greatest

sum over all subarrays entirely in AŒlow : : mid, entirely in AŒmid C 1 : : high,

or crossing the midpoint. We can ﬁnd maximum subarrays of AŒlow : : mid and

AŒmidC1 : : high recursively, because these two subproblems are smaller instances

of the problem of ﬁnding a maximum subarray. Thus, all that is left to do is ﬁnd a

4.1 The maximum-subarray problem 71

(a) (b)

low low mid mid high high

crosses the midpoint

mid C 1 mid C 1

entirely in AŒlow : : mid entirely in AŒmid C 1 : : high

i

j

AŒi : : mid

AŒmid C 1 : : j

Figure 4.4 (a) Possible locations of subarrays of AŒlow : : high: entirely in AŒlow : : mid, entirely

in AŒmid C 1 : : high, or crossing the midpoint mid. (b) Any subarray of AŒlow : : high crossing

the midpoint comprises two subarrays AŒi : : mid and AŒmid C 1 : : j , where low i mid and

mid < j high.

maximum subarray that crosses the midpoint, and take a subarray with the largest

sum of the three.

We can easily ﬁnd a maximum subarray crossing the midpoint in time linear

in the size of the subarray AŒlow : : high. This problem is not a smaller instance

of our original problem, because it has the added restriction that the subarray it

chooses must cross the midpoint. As Figure 4.4(b) shows, any subarray crossing

the midpoint is itself made of two subarrays AŒi : : mid and AŒmid C 1 : : j , where

low i mid and mid < j high. Therefore, we just need to ﬁnd maximum

subarrays of the form AŒi : : mid and AŒmid C 1 : : j and then combine them. The

procedure FIND-MAX-CROSSING-SUBARRAY takes as input the array A and the

indices low, mid, and high, and it returns a tuple containing the indices demarcating

a maximum subarray that crosses the midpoint, along with the sum of the values in

a maximum subarray.

FIND-MAX-CROSSING-SUBARRAY.A; low; mid; high/

1 left-sum D 1

2 sum D 0

3 for i D mid downto low

4 sum D sum C AŒi

5 if sum > left-sum

6 left-sum D sum

7 max-left D i

8 right-sum D 1

9 sum D 0

10 for j D mid C 1 to high

11 sum D sum C AŒj

12 if sum > right-sum

13 right-sum D sum

14 max-right D j

15 return .max-left; max-right; left-sum C right-sum/

72 Chapter 4 Divide-and-Conquer

This procedure works as follows. Lines 1–7 ﬁnd a maximum subarray of the

left half, AŒlow : : mid. Since this subarray must contain AŒmid, the for loop of

lines 3–7 starts the index i at mid and works down to low, so that every subarray

it considers is of the form AŒi : : mid. Lines 1–2 initialize the variables left-sum,

which holds the greatest sum found so far, and sum, holding the sum of the entries

in AŒi : : mid. Whenever we ﬁnd, in line 5, a subarray AŒi : : mid with a sum of

values greater than left-sum, we update left-sum to this subarray’s sum in line 6, and

in line 7 we update the variable max-left to record this index i. Lines 8–14 work

analogously for the right half, AŒmid C1 : : high. Here, the for loop of lines 10–14

starts the index j at midC1 and works up to high, so that every subarray it considers

is of the form AŒmid C 1 : : j . Finally, line 15 returns the indices max-left and

max-right that demarcate a maximum subarray crossing the midpoint, along with

the sum left-sumCright-sum of the values in the subarray AŒmax-left : : max-right.

If the subarray AŒlow : : high contains n entries (so that n D high low C 1),

we claim that the call FIND-MAX-CROSSING-SUBARRAY.A; low; mid; high/

takes ‚.n/ time. Since each iteration of each of the two for loops takes ‚.1/

time, we just need to count up how many iterations there are altogether. The for

loop of lines 3–7 makes mid low C 1 iterations, and the for loop of lines 10–14

makes high mid iterations, and so the total number of iterations is

.mid low C 1/ C .high mid/ D high low C 1

D n :

With a linear-time FIND-MAX-CROSSING-SUBARRAY procedure in hand, we

can write pseudocode for a divide-and-conquer algorithm to solve the maximum-

subarray problem:

FIND-MAXIMUM-SUBARRAY.A; low; high/

1 if high == low

2 return .low; high; AŒlow/ // base case: only one element

3 else mid D b.low C high/=2c

4 .left-low; left-high; left-sum/ D

FIND-MAXIMUM-SUBARRAY.A; low; mid/

5 .right-low; right-high; right-sum/ D

FIND-MAXIMUM-SUBARRAY.A; mid C 1; high/

6 .cross-low; cross-high; cross-sum/ D

FIND-MAX-CROSSING-SUBARRAY.A; low; mid; high/

7 if left-sum right-sum and left-sum cross-sum

8 return .left-low; left-high; left-sum/

9 elseif right-sum left-sum and right-sum cross-sum

10 return .right-low; right-high; right-sum/

11 else return .cross-low; cross-high; cross-sum/

4.1 The maximum-subarray problem 73

The initial call FIND-MAXIMUM-SUBARRAY.A; 1; A:length/ will ﬁnd a maxi-

mum subarray of AŒ1 : : n.

Similar to FIND-MAX-CROSSING-SUBARRAY, the recursive procedure FIND-

MAXIMUM-SUBARRAY returns a tuple containing the indices that demarcate a

maximum subarray, along with the sum of the values in a maximum subarray.

Line 1 tests for the base case, where the subarray has just one element. A subar-

ray with just one element has only one subarray—itself—and so line 2 returns a

tuple with the starting and ending indices of just the one element, along with its

value. Lines 3–11 handle the recursive case. Line 3 does the divide part, comput-

ing the index mid of the midpoint. Let’s refer to the subarray AŒlow : : mid as the

left subarray and to AŒmid C 1 : : high as the right subarray. Because we know

that the subarray AŒlow : : high contains at least two elements, each of the left and

right subarrays must have at least one element. Lines 4 and 5 conquer by recur-

sively ﬁnding maximum subarrays within the left and right subarrays, respectively.

Lines 6–11 form the combine part. Line 6 ﬁnds a maximum subarray that crosses

the midpoint. (Recall that because line 6 solves a subproblem that is not a smaller

instance of the original problem, we consider it to be in the combine part.) Line 7

tests whether the left subarray contains a subarray with the maximum sum, and

line 8 returns that maximum subarray. Otherwise, line 9 tests whether the right

subarray contains a subarray with the maximum sum, and line 10 returns that max-

imum subarray. If neither the left nor right subarrays contain a subarray achieving

the maximum sum, then a maximum subarray must cross the midpoint, and line 11

returns it.

Analyzing the divide-and-conquer algorithm

Next we set up a recurrence that describes the running time of the recursive FIND-

MAXIMUM-SUBARRAY procedure. As we did when we analyzed merge sort in

Section 2.3.2, we make the simplifying assumption that the original problem size

is a power of 2, so that all subproblem sizes are integers. We denote by T .n/ the

running time of FIND-MAXIMUM-SUBARRAY on a subarray of n elements. For

starters, line 1 takes constant time. The base case, when n D 1, is easy: line 2

takes constant time, and so

T .1/ D ‚.1/ : (4.5)

The recursive case occurs when n > 1. Lines 1 and 3 take constant time. Each

of the subproblems solved in lines 4 and 5 is on a subarray of n=2 elements (our

assumption that the original problem size is a power of 2 ensures that n=2 is an

integer), and so we spend T .n=2/ time solving each of them. Because we have

to solve two subproblems—for the left subarray and for the right subarray—the

contribution to the running time from lines 4 and 5 comes to 2T .n=2/. As we have

74 Chapter 4 Divide-and-Conquer

already seen, the call to FIND-MAX-CROSSING-SUBARRAY in line 6 takes ‚.n/

time. Lines 7–11 take only ‚.1/ time. For the recursive case, therefore, we have

T .n/ D ‚.1/ C 2T .n=2/ C ‚.n/ C ‚.1/

D 2T .n=2/ C ‚.n/ : (4.6)

Combining equations (4.5) and (4.6) gives us a recurrence for the running

time T .n/ of FIND-MAXIMUM-SUBARRAY:

T .n/ D

(

‚.1/ if n D 1 ;

2T .n=2/ C ‚.n/ if n > 1 :

(4.7)

This recurrence is the same as recurrence (4.1) for merge sort. As we shall

see from the master method in Section 4.5, this recurrence has the solution

T .n/ D ‚.n lg n/. You might also revisit the recursion tree in Figure 2.5 to un-

derstand why the solution should be T .n/ D ‚.n lg n/.

Thus, we see that the divide-and-conquer method yields an algorithm that is

asymptotically faster than the brute-force method. With merge sort and now the

maximum-subarray problem, we begin to get an idea of how powerful the divide-

and-conquer method can be. Sometimes it will yield the asymptotically fastest

algorithm for a problem, and other times we can do even better. As Exercise 4.1-5

shows, there is in fact a linear-time algorithm for the maximum-subarray problem,

and it does not use divide-and-conquer.

Exercises

4.1-1

What does FIND-MAXIMUM-SUBARRAY return when all elements of A are nega-

tive?

4.1-2

Write pseudocode for the brute-force method of solving the maximum-subarray

problem. Your procedure should run in ‚.n 2 / time.

4.1-3

Implement both the brute-force and recursive algorithms for the maximum-

subarray problem on your own computer. What problem size n 0 gives the crossover

point at which the recursive algorithm beats the brute-force algorithm? Then,

change the base case of the recursive algorithm to use the brute-force algorithm

whenever the problem size is less than n 0 . Does that change the crossover point?

4.1-4

Suppose we change the deﬁnition of the maximum-subarray problem to allow the

result to be an empty subarray, where the sum of the values of an empty subar-

4.2 Strassen’s algorithm for matrix multiplication 75

ray is 0. How would you change any of the algorithms that do not allow empty

subarrays to permit an empty subarray to be the result?

4.1-5

Use the following ideas to develop a nonrecursive, linear-time algorithm for the

maximum-subarray problem. Start at the left end of the array, and progress toward

the right, keeping track of the maximum subarray seen so far. Knowing a maximum

subarray of AŒ1 : : j , extend the answer to ﬁnd a maximum subarray ending at in-

dex j C1 by using the following observation: a maximum subarray of AŒ1 : : j C 1

is either a maximum subarray of AŒ1 : : j or a subarray AŒi : : j C 1, for some

1 i j C 1. Determine a maximum subarray of the form AŒi : : j C 1 in

constant time based on knowing a maximum subarray ending at index j .

4.2 Strassen’s algorithm for matrix multiplication

If you have seen matrices before, then you probably know how to multiply them.

(Otherwise, you should read Section D.1 in Appendix D.) If A D .a ij / and

B D .b ij / are square n n matrices, then in the product C D A B, we deﬁne the

entry c ij , for i; j D 1; 2; : : : ; n, by

c ij D

n X

kD1

a ik b kj : (4.8)

We must compute n 2

matrix entries, and each is the sum of n values. The following

procedure takes n n matrices A and B and multiplies them, returning their n n

product C . We assume that each matrix has an attribute rows, giving the number

of rows in the matrix.

SQUARE-MATRIX-MULTIPLY.A; B/

1 n D A:rows

2 let C be a new n n matrix

3 for i D 1 to n

4 for j D 1 to n

5 c ij D 0

6 for k D 1 to n

7 c ij D c ij C a ik b kj

8 return C

The SQUARE-MATRIX-MULTIPLY procedure works as follows. The for loop

of lines 3–7 computes the entries of each row i, and within a given row i, the

76 Chapter 4 Divide-and-Conquer

for loop of lines 4–7 computes each of the entries c ij , for each column j . Line 5

initializes c ij to 0 as we start computing the sum given in equation (4.8), and each

iteration of the for loop of lines 6–7 adds in one more term of equation (4.8).

Because each of the triply-nested for loops runs exactly n iterations, and each

execution of line 7 takes constant time, the SQUARE-MATRIX-MULTIPLY proce-

dure takes ‚.n 3 / time.

You might at ﬁrst think that any matrix multiplication algorithm must take .n 3 /

time, since the natural deﬁnition of matrix multiplication requires that many mul-

tiplications. You would be incorrect, however: we have a way to multiply matrices

in o.n 3 / time. In this section, we shall see Strassen’s remarkable recursive algo-

rithm for multiplying n n matrices. It runs in ‚.n lg 7 / time, which we shall show

in Section 4.5. Since lg 7 lies between 2:80 and 2:81, Strassen’s algorithm runs in

O.n 2:81 / time, which is asymptotically better than the simple SQUARE-MATRIX-

MULTIPLY procedure.

A simple divide-and-conquer algorithm

To keep things simple, when we use a divide-and-conquer algorithm to compute

the matrix product C D A B, we assume that n is an exact power of 2 in each of

the n n matrices. We make this assumption because in each divide step, we will

divide n n matrices into four n=2 n=2 matrices, and by assuming that n is an

exact power of 2, we are guaranteed that as long as n 2, the dimension n=2 is an

integer.

Suppose that we partition each of A, B, and C into four n=2 n=2 matrices

A D

A 11 A 12

A 21 A 22

; B D

B 11 B 12

B 21 B 22

; C D

C 11 C 12

C 21 C 22

; (4.9)

so that we rewrite the equation C D A B as

C 11 C 12

C 21 C 22

D

A 11 A 12

A 21 A 22

B 11 B 12

B 21 B 22

: (4.10)

Equation (4.10) corresponds to the four equations

C 11 D A 11 B 11 C A 12 B 21 ; (4.11)

C 12 D A 11 B 12 C A 12 B 22 ; (4.12)

C 21 D A 21 B 11 C A 22 B 21 ; (4.13)

C 22 D A 21 B 12 C A 22 B 22 : (4.14)

Each of these four equations speciﬁes two multiplications of n=2 n=2 matrices

and the addition of their n=2 n=2 products. We can use these equations to create

a straightforward, recursive, divide-and-conquer algorithm:

4.2 Strassen’s algorithm for matrix multiplication 77

SQUARE-MATRIX-MULTIPLY-RECURSIVE.A; B/

1 n D A:rows

2 let C be a new n n matrix

3 if n == 1

4 c 11 D a 11 b 11

5 else partition A, B, and C as in equations (4.9)

6 C 11 D SQUARE-MATRIX-MULTIPLY-RECURSIVE.A 11 ; B 11 /

C SQUARE-MATRIX-MULTIPLY-RECURSIVE.A 12 ; B 21 /

7 C 12 D SQUARE-MATRIX-MULTIPLY-RECURSIVE.A 11 ; B 12 /

C SQUARE-MATRIX-MULTIPLY-RECURSIVE.A 12 ; B 22 /

8 C 21 D SQUARE-MATRIX-MULTIPLY-RECURSIVE.A 21 ; B 11 /

C SQUARE-MATRIX-MULTIPLY-RECURSIVE.A 22 ; B 21 /

9 C 22 D SQUARE-MATRIX-MULTIPLY-RECURSIVE.A 21 ; B 12 /

C SQUARE-MATRIX-MULTIPLY-RECURSIVE.A 22 ; B 22 /

10 return C

This pseudocode glosses over one subtle but important implementation detail.

How do we partition the matrices in line 5? If we were to create 12 new n=2 n=2

matrices, we would spend ‚.n 2 / time copying entries. In fact, we can partition

the matrices without copying entries. The trick is to use index calculations. We

identify a submatrix by a range of row indices and a range of column indices of

the original matrix. We end up representing a submatrix a little differently from

how we represent the original matrix, which is the subtlety we are glossing over.

The advantage is that, since we can specify submatrices by index calculations,

executing line 5 takes only ‚.1/ time (although we shall see that it makes no

difference asymptotically to the overall running time whether we copy or partition

in place).

Now, we derive a recurrence to characterize the running time of SQUARE-

MATRIX-MULTIPLY-RECURSIVE. Let T .n/ be the time to multiply two n n

matrices using this procedure. In the base case, when n D 1, we perform just the

one scalar multiplication in line 4, and so

T .1/ D ‚.1/ : (4.15)

The recursive case occurs when n > 1. As discussed, partitioning the matrices in

line 5 takes ‚.1/ time, using index calculations. In lines 6–9, we recursively call

SQUARE-MATRIX-MULTIPLY-RECURSIVE a total of eight times. Because each

recursive call multiplies two n=2 n=2 matrices, thereby contributing T .n=2/ to

the overall running time, the time taken by all eight recursive calls is 8T .n=2/. We

also must account for the four matrix additions in lines 6–9. Each of these matrices

contains n 2 =4 entries, and so each of the four matrix additions takes ‚.n 2 / time.

Since the number of matrix additions is a constant, the total time spent adding ma-

78 Chapter 4 Divide-and-Conquer

trices in lines 6–9 is ‚.n 2 /. (Again, we use index calculations to place the results

of the matrix additions into the correct positions of matrix C , with an overhead

of ‚.1/ time per entry.) The total time for the recursive case, therefore, is the sum

of the partitioning time, the time for all the recursive calls, and the time to add the

matrices resulting from the recursive calls:

T .n/ D ‚.1/ C 8T .n=2/ C ‚.n

2

/

D 8T .n=2/ C ‚.n

2

/ : (4.16)

Notice that if we implemented partitioning by copying matrices, which would cost

‚.n 2 / time, the recurrence would not change, and hence the overall running time

would increase by only a constant factor.

Combining equations (4.15) and (4.16) gives us the recurrence for the running

time of SQUARE-MATRIX-MULTIPLY-RECURSIVE:

T .n/ D

(

‚.1/ if n D 1 ;

8T .n=2/ C ‚.n 2 / if n > 1 :

(4.17)

As we shall see from the master method in Section 4.5, recurrence (4.17) has the

solution T .n/ D ‚.n 3 /. Thus, this simple divide-and-conquer approach is no

faster than the straightforward SQUARE-MATRIX-MULTIPLY procedure.

Before we continue on to examining Strassen’s algorithm, let us review where

the components of equation (4.16) came from. Partitioning each n n matrix by

index calculation takes ‚.1/ time, but we have two matrices to partition. Although

you could say that partitioning the two matrices takes ‚.2/ time, the constant of 2

is subsumed by the ‚-notation. Adding two matrices, each with, say, k entries,

takes ‚.k/ time. Since the matrices we add each have n 2 =4 entries, you could

say that adding each pair takes ‚.n 2 =4/ time. Again, however, the ‚-notation

subsumes the constant factor of 1=4, and we say that adding two n 2 =4 n 2 =4

matrices takes ‚.n 2 / time. We have four such matrix additions, and once again,

instead of saying that they take ‚.4n 2 / time, we say that they take ‚.n 2 / time.

(Of course, you might observe that we could say that the four matrix additions

take ‚.4n 2 =4/ time, and that 4n 2 =4 D n 2

, but the point here is that ‚-notation

subsumes constant factors, whatever they are.) Thus, we end up with two terms

of ‚.n 2 /, which we can combine into one.

When we account for the eight recursive calls, however, we cannot just sub-

sume the constant factor of 8. In other words, we must say that together they take

8T .n=2/ time, rather than just T .n=2/ time. You can get a feel for why by looking

back at the recursion tree in Figure 2.5, for recurrence (2.1) (which is identical to

recurrence (4.7)), with the recursive case T .n/ D 2T .n=2/C‚.n/. The factor of 2

determined how many children each tree node had, which in turn determined how

many terms contributed to the sum at each level of the tree. If we were to ignore

4.2 Strassen’s algorithm for matrix multiplication 79

the factor of 8 in equation (4.16) or the factor of 2 in recurrence (4.1), the recursion

tree would just be linear, rather than “bushy,” and each level would contribute only

one term to the sum.

Bear in mind, therefore, that although asymptotic notation subsumes constant

multiplicative factors, recursive notation such as T .n=2/ does not.

Strassen’s method

The key to Strassen’s method is to make the recursion tree slightly less bushy. That

is, instead of performing eight recursive multiplications of n=2 n=2 matrices,

it performs only seven. The cost of eliminating one matrix multiplication will be

several new additions of n=2 n=2 matrices, but still only a constant number of

additions. As before, the constant number of matrix additions will be subsumed

by ‚-notation when we set up the recurrence equation to characterize the running

time.

Strassen’s method is not at all obvious. (This might be the biggest understate-

ment in this book.) It has four steps:

1. Divide the input matrices A and B and output matrix C into n=2 n=2 subma-

trices, as in equation (4.9). This step takes ‚.1/ time by index calculation, just

as in SQUARE-MATRIX-MULTIPLY-RECURSIVE.

2. Create 10 matrices S 1 ; S 2 ; : : : ; S 10 , each of which is n=2 n=2 and is the sum

or difference of two matrices created in step 1. We can create all 10 matrices in

‚.n 2 / time.

3. Using the submatrices created in step 1 and the 10 matrices created in step 2,

recursively compute seven matrix products P 1 ; P 2 ; : : : ; P 7 . Each matrix P i is

n=2 n=2.

4. Compute the desired submatrices C 11 ; C 12 ; C 21 ; C 22 of the result matrix C by

adding and subtracting various combinations of the P i matrices. We can com-

pute all four submatrices in ‚.n 2 / time.

We shall see the details of steps 2–4 in a moment, but we already have enough

information to set up a recurrence for the running time of Strassen’s method. Let us

assume that once the matrix size n gets down to 1, we perform a simple scalar mul-

tiplication, just as in line 4 of SQUARE-MATRIX-MULTIPLY-RECURSIVE. When

n > 1, steps 1, 2, and 4 take a total of ‚.n 2 / time, and step 3 requires us to per-

form seven multiplications of n=2 n=2 matrices. Hence, we obtain the following

recurrence for the running time T .n/ of Strassen’s algorithm:

T .n/ D

(

‚.1/ if n D 1 ;

7T .n=2/ C ‚.n 2 / if n > 1 :

(4.18)

80 Chapter 4 Divide-and-Conquer

We have traded off one matrix multiplication for a constant number of matrix ad-

ditions. Once we understand recurrences and their solutions, we shall see that this

tradeoff actually leads to a lower asymptotic running time. By the master method

in Section 4.5, recurrence (4.18) has the solution T .n/ D ‚.n lg 7 /.

We now proceed to describe the details. In step 2, we create the following 10

matrices:

S 1 D B 12 B 22 ;

S 2 D A 11 C A 12 ;

S 3 D A 21 C A 22 ;

S 4 D B 21 B 11 ;

S 5 D A 11 C A 22 ;

S 6 D B 11 C B 22 ;

S 7 D A 12 A 22 ;

S 8 D B 21 C B 22 ;

S 9 D A 11 A 21 ;

S 10 D B 11 C B 12 :

Since we must add or subtract n=2 n=2 matrices 10 times, this step does indeed

take ‚.n 2 / time.

In step 3, we recursively multiply n=2 n=2 matrices seven times to compute the

following n=2 n=2 matrices, each of which is the sum or difference of products

of A and B submatrices:

P 1 D A 11 S 1 D A 11 B 12 A 11 B 22 ;

P 2 D S 2 B 22 D A 11 B 22 C A 12 B 22 ;

P 3 D S 3 B 11 D A 21 B 11 C A 22 B 11 ;

P 4 D A 22 S 4 D A 22 B 21 A 22 B 11 ;

P 5 D S 5 S 6 D A 11 B 11 C A 11 B 22 C A 22 B 11 C A 22 B 22 ;

P 6 D S 7 S 8 D A 12 B 21 C A 12 B 22 A 22 B 21 A 22 B 22 ;

P 7 D S 9 S 10 D A 11 B 11 C A 11 B 12 A 21 B 11 A 21 B 12 :

Note that the only multiplications we need to perform are those in the middle col-

umn of the above equations. The right-hand column just shows what these products

equal in terms of the original submatrices created in step 1.

Step 4 adds and subtracts the P i matrices created in step 3 to construct the four

n=2 n=2 submatrices of the product C . We start with

C 11 D P 5 C P 4 P 2 C P 6 :

4.2 Strassen’s algorithm for matrix multiplication 81

Expanding out the right-hand side, with the expansion of each P i on its own line

and vertically aligning terms that cancel out, we see that C 11 equals

A 11 B 11 CA 11 B 22 CA 22 B 11 C A 22 B 22

A 22 B 11 CA 22 B 21

A 11 B 22 A 12 B 22

A 22 B 22 A 22 B 21 CA 12 B 22 C A 12 B 21

A 11 B 11 C A 12 B 21 ;

which corresponds to equation (4.11).

Similarly, we set

C 12 D P 1 C P 2 ;

and so C 12 equals

A 11 B 12 A 11 B 22

C A 11 B 22 C A 12 B 22

A 11 B 12 C A 12 B 22 ;

corresponding to equation (4.12).

Setting

C 21 D P 3 C P 4

makes C 21 equal

A 21 B 11 C A 22 B 11

A 22 B 11 C A 22 B 21

A 21 B 11 C A 22 B 21 ;

corresponding to equation (4.13).

Finally, we set

C 22 D P 5 C P 1 P 3 P 7 ;

so that C 22 equals

A 11 B 11 C A 11 B 22 CA 22 B 11 C A 22 B 22

A 11 B 22 C A 11 B 12

A 22 B 11 A 21 B 11

A 11 B 11 A 11 B 12 CA 21 B 11 C A 21 B 12

A 22 B 22 C A 21 B 12 ;

82 Chapter 4 Divide-and-Conquer

which corresponds to equation (4.14). Altogether, we add or subtract n=2 n=2

matrices eight times in step 4, and so this step indeed takes ‚.n 2 / time.

Thus, we see that Strassen’s algorithm, comprising steps 1–4, produces the cor-

rect matrix product and that recurrence (4.18) characterizes its running time. Since

we shall see in Section 4.5 that this recurrence has the solution T .n/ D ‚.n lg 7 /,

Strassen’s method is asymptotically faster than the straightforward SQUARE-

MATRIX-MULTIPLY procedure. The notes at the end of this chapter discuss some

of the practical aspects of Strassen’s algorithm.

Exercises

Note: Although Exercises 4.2-3, 4.2-4, and 4.2-5 are about variants on Strassen’s

algorithm, you should read Section 4.5 before trying to solve them.

4.2-1

Use Strassen’s algorithm to compute the matrix product

1 3

7 5

6 8

4 2

:

Show your work.

4.2-2

Write pseudocode for Strassen’s algorithm.

4.2-3

How would you modify Strassen’s algorithm to multiply n n matrices in which n

is not an exact power of 2? Show that the resulting algorithm runs in time ‚.n lg 7 /.

4.2-4

What is the largest k such that if you can multiply 3 3 matrices using k multi-

plications (not assuming commutativity of multiplication), then you can multiply

n n matrices in time o.n lg 7 /? What would the running time of this algorithm be?

4.2-5

V. Pan has discovered a way of multiplying 68 68 matrices using 132,464 mul-

tiplications, a way of multiplying 70 70 matrices using 143,640 multiplications,

and a way of multiplying 72 72 matrices using 155,424 multiplications. Which

method yields the best asymptotic running time when used in a divide-and-conquer

matrix-multiplication algorithm? How does it compare to Strassen’s algorithm?

4.3 The substitution method for solving recurrences 83

4.2-6

How quickly can you multiply a kn n matrix by an n kn matrix, using Strassen’s

algorithm as a subroutine? Answer the same question with the order of the input

matrices reversed.

4.2-7

Show how to multiply the complex numbers a C bi and c C di using only three

multiplications of real numbers. The algorithm should take a, b, c, and d as input

and produce the real component ac bd and the imaginary component ad C bc

separately.

4.3 The substitution method for solving recurrences

Now that we have seen how recurrences characterize the running times of divide-

and-conquer algorithms, we will learn how to solve recurrences. We start in this

section with the “substitution” method.

The substitution method for solving recurrences comprises two steps:

1. Guess the form of the solution.

2. Use mathematical induction to ﬁnd the constants and show that the solution

works.

We substitute the guessed solution for the function when applying the inductive

hypothesis to smaller values; hence the name “substitution method.” This method

is powerful, but we must be able to guess the form of the answer in order to apply it.

We can use the substitution method to establish either upper or lower bounds on

a recurrence. As an example, let us determine an upper bound on the recurrence

T .n/ D 2T .bn=2c/ C n ; (4.19)

which is similar to recurrences (4.3) and (4.4). We guess that the solution is

T .n/ D O.n lg n/. The substitution method requires us to prove that T .n/

cn lg n for an appropriate choice of the constant c > 0. We start by assuming

that this bound holds for all positive m < n, in particular for m D bn=2c, yielding

T .bn=2c/ c bn=2c lg.bn=2c/. Substituting into the recurrence yields

T .n/ 2.c bn=2c lg.bn=2c// C n

cn lg.n=2/ C n

D cn lg n cn lg 2 C n

D cn lg n cn C n

cn lg n ;

84 Chapter 4 Divide-and-Conquer

where the last step holds as long as c 1.

Mathematical induction now requires us to show that our solution holds for the

boundary conditions. Typically, we do so by showing that the boundary condi-

tions are suitable as base cases for the inductive proof. For the recurrence (4.19),

we must show that we can choose the constant c large enough so that the bound

T .n/ cn lg n works for the boundary conditions as well. This requirement

can sometimes lead to problems. Let us assume, for the sake of argument, that

T .1/ D 1 is the sole boundary condition of the recurrence. Then for n D 1, the

bound T .n/ cn lg n yields T .1/ c1 lg 1 D 0, which is at odds with T .1/ D 1.

Consequently, the base case of our inductive proof fails to hold.

We can overcome this obstacle in proving an inductive hypothesis for a spe-

ciﬁc boundary condition with only a little more effort. In the recurrence (4.19),

for example, we take advantage of asymptotic notation requiring us only to prove

T .n/ cn lg n for n n 0 , where n 0 is a constant that we get to choose. We

keep the troublesome boundary condition T .1/ D 1, but remove it from consid-

eration in the inductive proof. We do so by ﬁrst observing that for n > 3, the

recurrence does not depend directly on T .1/. Thus, we can replace T .1/ by T .2/

and T .3/ as the base cases in the inductive proof, letting n 0 D 2. Note that we

make a distinction between the base case of the recurrence (n D 1) and the base

cases of the inductive proof (n D 2 and n D 3). With T .1/ D 1, we derive from

the recurrence that T .2/ D 4 and T .3/ D 5. Now we can complete the inductive

proof that T .n/ cn lg n for some constant c 1 by choosing c large enough

so that T .2/ c2 lg 2 and T .3/ c3 lg 3. As it turns out, any choice of c 2

sufﬁces for the base cases of n D 2 and n D 3 to hold. For most of the recurrences

we shall examine, it is straightforward to extend boundary conditions to make the

inductive assumption work for small n, and we shall not always explicitly work out

the details.

Making a good guess

Unfortunately, there is no general way to guess the correct solutions to recurrences.

Guessing a solution takes experience and, occasionally, creativity. Fortunately,

though, you can use some heuristics to help you become a good guesser. You

can also use recursion trees, which we shall see in Section 4.4, to generate good

guesses.

If a recurrence is similar to one you have seen before, then guessing a similar

solution is reasonable. As an example, consider the recurrence

T .n/ D 2T .bn=2c C 17/ C n ;

which looks difﬁcult because of the added “17” in the argument to T on the right-

hand side. Intuitively, however, this additional term cannot substantially affect the

4.3 The substitution method for solving recurrences 85

solution to the recurrence. When n is large, the difference between bn=2c and

bn=2c C 17 is not that large: both cut n nearly evenly in half. Consequently, we

make the guess that T .n/ D O.n lg n/, which you can verify as correct by using

the substitution method (see Exercise 4.3-6).

Another way to make a good guess is to prove loose upper and lower bounds on

the recurrence and then reduce the range of uncertainty. For example, we might

start with a lower bound of T .n/ D .n/ for the recurrence (4.19), since we

have the term n in the recurrence, and we can prove an initial upper bound of

T .n/ D O.n 2 /. Then, we can gradually lower the upper bound and raise the

lower bound until we converge on the correct, asymptotically tight solution of

T .n/ D ‚.n lg n/.

Subtleties

Sometimes you might correctly guess an asymptotic bound on the solution of a

recurrence, but somehow the math fails to work out in the induction. The problem

frequently turns out to be that the inductive assumption is not strong enough to

prove the detailed bound. If you revise the guess by subtracting a lower-order term

when you hit such a snag, the math often goes through.

Consider the recurrence

T .n/ D T .bn=2c/ C T .dn=2e/ C 1 :

We guess that the solution is T .n/ D O.n/, and we try to show that T .n/ cn for

an appropriate choice of the constant c. Substituting our guess in the recurrence,

we obtain

T .n/ c bn=2c C c dn=2e C 1

D cn C 1 ;

which does not imply T .n/ cn for any choice of c. We might be tempted to try

a larger guess, say T .n/ D O.n 2 /. Although we can make this larger guess work,

our original guess of T .n/ D O.n/ is correct. In order to show that it is correct,

however, we must make a stronger inductive hypothesis.

Intuitively, our guess is nearly right: we are off only by the constant 1, a

lower-order term. Nevertheless, mathematical induction does not work unless we

prove the exact form of the inductive hypothesis. We overcome our difﬁculty

by subtracting a lower-order term from our previous guess. Our new guess is

T .n/ cn d, where d 0 is a constant. We now have

T .n/ .c bn=2c d/ C .c dn=2e d/ C 1

D cn 2d C 1

cn d ;

86 Chapter 4 Divide-and-Conquer

as long as d 1. As before, we must choose the constant c large enough to handle

the boundary conditions.

You might ﬁnd the idea of subtracting a lower-order term counterintuitive. Af-

ter all, if the math does not work out, we should increase our guess, right?

Not necessarily! When proving an upper bound by induction, it may actually be

more difﬁcult to prove that a weaker upper bound holds, because in order to prove

the weaker bound, we must use the same weaker bound inductively in the proof.

In our current example, when the recurrence has more than one recursive term, we

get to subtract out the lower-order term of the proposed bound once per recursive

term. In the above example, we subtracted out the constant d twice, once for the

T .bn=2c/ term and once for the T .dn=2e/ term. We ended up with the inequality

T .n/ cn 2d C 1, and it was easy to ﬁnd values of d to make cn 2d C 1 be

less than or equal to cn d.

Avoiding pitfalls

It is easy to err in the use of asymptotic notation. For example, in the recur-

rence (4.19) we can falsely “prove” T .n/ D O.n/ by guessing T .n/ cn and

then arguing

T .n/ 2.c bn=2c/ C n

cn C n

D O.n/ ;

wrong!!

since c is a constant. The error is that we have not proved the exact form of the

inductive hypothesis, that is, that T .n/ cn. We therefore will explicitly prove

that T .n/ cn when we want to show that T .n/ D O.n/.

Changing variables

Sometimes, a little algebraic manipulation can make an unknown recurrence simi-

lar to one you have seen before. As an example, consider the recurrence

T .n/ D 2T

p

n

˘

C lg n ;

which looks difﬁcult. We can simplify this recurrence, though, with a change of

variables. For convenience, we shall not worry about rounding off values, such

as

p

n, to be integers. Renaming m D lg n yields

T .2

m

/ D 2T .2

m=2

/ C m :

We can now rename S.m/ D T .2 m / to produce the new recurrence

S.m/ D 2S.m=2/ C m ;

4.3 The substitution method for solving recurrences 87

which is very much like recurrence (4.19). Indeed, this new recurrence has the

same solution: S.m/ D O.m lg m/. Changing back from S.m/ to T .n/, we obtain

T .n/ D T .2

m

/ D S.m/ D O.m lg m/ D O.lg n lg lg n/ :

Exercises

4.3-1

Show that the solution of T .n/ D T .n 1/ C n is O.n 2 /.

4.3-2

Show that the solution of T .n/ D T .dn=2e/ C 1 is O.lg n/.

4.3-3

We saw that the solution of T .n/ D 2T .bn=2c/Cn is O.n lg n/. Show that the so-

lution of this recurrence is also .n lg n/. Conclude that the solution is ‚.n lg n/.

4.3-4

Show that by making a different inductive hypothesis, we can overcome the difﬁ-

culty with the boundary condition T .1/ D 1 for recurrence (4.19) without adjusting

the boundary conditions for the inductive proof.

4.3-5

Show that ‚.n lg n/ is the solution to the “exact” recurrence (4.3) for merge sort.

4.3-6

Show that the solution to T .n/ D 2T .bn=2c C 17/ C n is O.n lg n/.

4.3-7

Using the master method in Section 4.5, you can show that the solution to the

recurrence T .n/ D 4T .n=3/ C n is T .n/ D ‚.n log 3 4 /. Show that a substitution

proof with the assumption T .n/ cn log 3 4

fails. Then show how to subtract off a

lower-order term to make a substitution proof work.

4.3-8

Using the master method in Section 4.5, you can show that the solution to the

recurrence T .n/ D 4T .n=2/ C n 2

is T .n/ D ‚.n 2 /. Show that a substitution

proof with the assumption T .n/ cn 2

fails. Then show how to subtract off a

lower-order term to make a substitution proof work.

88 Chapter 4 Divide-and-Conquer

4.3-9

Solve the recurrence T .n/ D 3T .

p

n/ C log n by making a change of variables.

Your solution should be asymptotically tight. Do not worry about whether values

are integral.

4.4 The recursion-tree method for solving recurrences

Although you can use the substitution method to provide a succinct proof that

a solution to a recurrence is correct, you might have trouble coming up with a

good guess. Drawing out a recursion tree, as we did in our analysis of the merge

sort recurrence in Section 2.3.2, serves as a straightforward way to devise a good

guess. In a recursion tree, each node represents the cost of a single subproblem

somewhere in the set of recursive function invocations. We sum the costs within

each level of the tree to obtain a set of per-level costs, and then we sum all the

per-level costs to determine the total cost of all levels of the recursion.

A recursion tree is best used to generate a good guess, which you can then verify

by the substitution method. When using a recursion tree to generate a good guess,

you can often tolerate a small amount of “sloppiness,” since you will be verifying

your guess later on. If you are very careful when drawing out a recursion tree and

summing the costs, however, you can use a recursion tree as a direct proof of a

solution to a recurrence. In this section, we will use recursion trees to generate

good guesses, and in Section 4.6, we will use recursion trees directly to prove the

theorem that forms the basis of the master method.

For example, let us see how a recursion tree would provide a good guess for

the recurrence T .n/ D 3T .bn=4c/ C ‚.n 2 /. We start by focusing on ﬁnding an

upper bound for the solution. Because we know that ﬂoors and ceilings usually do

not matter when solving recurrences (here’s an example of sloppiness that we can

tolerate), we create a recursion tree for the recurrence T .n/ D 3T .n=4/ C cn 2

,

having written out the implied constant coefﬁcient c > 0.

Figure 4.5 shows how we derive the recursion tree for T .n/ D 3T .n=4/ C cn 2

.

For convenience, we assume that n is an exact power of 4 (another example of

tolerable sloppiness) so that all subproblem sizes are integers. Part (a) of the ﬁgure

shows T .n/, which we expand in part (b) into an equivalent tree representing the

recurrence. The cn 2

term at the root represents the cost at the top level of recursion,

and the three subtrees of the root represent the costs incurred by the subproblems

of size n=4. Part (c) shows this process carried one step further by expanding each

node with cost T .n=4/ from part (b). The cost for each of the three children of the

root is c.n=4/ 2

. We continue expanding each node in the tree by breaking it into

its constituent parts as determined by the recurrence.

4.4 The recursion-tree method for solving recurrences 89

…

…

(d)

(c) (b) (a)

T .n/ cn 2 cn 2

cn 2

T

n

4

T

n

4

T

n

4

T

n

16

T

n

16

T

n

16

T

n

16

T

n

16

T

n

16

T

n

16

T

n

16

T

n

16

cn 2

c

n

4

2

c

n

4

2

c

n

4

2

c

n

4

2

c

n

4

2

c

n

4

2

c

n

16

2

c

n

16

2

c

n

16

2

c

n

16

2

c

n

16

2

c

n

16

2

c

n

16

2

c

n

16

2

c

n

16

2

3

16

cn 2

3

16

2

cn 2

log 4 n

n log 4 3

T .1/ T .1/ T .1/ T .1/ T .1/ T .1/ T .1/ T .1/ T .1/ T .1/ T .1/ T .1/ T .1/ ‚.n log 4 3 /

Total: O.n 2 /

Figure 4.5 Constructing a recursion tree for the recurrence T .n/ D 3T .n=4/ C cn

2

. Part (a)

shows T .n/, which progressively expands in (b)–(d) to form the recursion tree. The fully expanded

tree in part (d) has height log4

n (it has log4

n C 1 levels).

90 Chapter 4 Divide-and-Conquer

Because subproblem sizes decrease by a factor of 4 each time we go down one

level, we eventually must reach a boundary condition. How far from the root do

we reach one? The subproblem size for a node at depth i is n=4 i

. Thus, the

subproblem size hits n D 1 when n=4 i D 1 or, equivalently, when i D log 4 n.

Thus, the tree has log 4 n C 1 levels (at depths 0; 1; 2; : : : ; log 4 n).

Next we determine the cost at each level of the tree. Each level has three times

more nodes than the level above, and so the number of nodes at depth i is 3 i

.

Because subproblem sizes reduce by a factor of 4 for each level we go down

from the root, each node at depth i, for i D 0; 1; 2; : : : ; log

4

n 1, has a cost

of c.n=4 i / 2

. Multiplying, we see that the total cost over all nodes at depth i, for

i D 0; 1; 2; : : : ; log

4

n 1, is 3 i c.n=4 i / 2 D .3=16/ i cn 2

. The bottom level, at

depth log

4

n, has 3 log 4 n D n log 4 3

nodes, each contributing cost T .1/, for a total

cost of n log 4 3 T .1/, which is ‚.n log 4 3 /, since we assume that T .1/ is a constant.

Now we add up the costs over all levels to determine the cost for the entire tree:

T .n/ D cn

2

C

3

16

cn

2

C

3

16

2

cn

2

C C

3

16

log 4 n1

cn

2

C ‚.n log 4 3

/

D

log 4 n1 X

iD0

3

16

i

cn

2

C ‚.n log 4 3

/

D

.3=16/ log 4 n 1

.3=16/ 1

cn

2

C ‚.n log 4 3

/ (by equation (A.5)) :

This last formula looks somewhat messy until we realize that we can again take

advantage of small amounts of sloppiness and use an inﬁnite decreasing geometric

series as an upper bound. Backing up one step and applying equation (A.6), we

have

T .n/ D

log 4 n1 X

iD0

3

16

i

cn

2

C ‚.n log 4 3

/

<

1 X

iD0

3

16

i

cn

2

C ‚.n log 4 3

/

D

1

1 .3=16/

cn

2

C ‚.n log 4 3

/

D

16

13

cn

2

C ‚.n log 4 3

/

D O.n

2

/ :

Thus, we have derived a guess of T .n/ D O.n 2 / for our original recurrence

T .n/ D 3T .bn=4c/ C ‚.n 2 /. In this example, the coefﬁcients of cn 2

form a

decreasing geometric series and, by equation (A.6), the sum of these coefﬁcients

4.4 The recursion-tree method for solving recurrences 91

…

…

cn

cn

cn

cn

c

n

3

c

2n

3

c

n

9

c

2n

9

c

2n

9

c

4n

9

log 3=2 n

Total: O.n lg n/

Figure 4.6 A recursion tree for the recurrence T .n/ D T .n=3/ C T .2n=3/ C cn.

is bounded from above by the constant 16=13. Since the root’s contribution to the

total cost is cn 2

, the root contributes a constant fraction of the total cost. In other

words, the cost of the root dominates the total cost of the tree.

In fact, if O.n 2 / is indeed an upper bound for the recurrence (as we shall verify in

a moment), then it must be a tight bound. Why? The ﬁrst recursive call contributes

a cost of ‚.n 2 /, and so .n 2 / must be a lower bound for the recurrence.

Now we can use the substitution method to verify that our guess was cor-

rect, that is, T .n/ D O.n 2 / is an upper bound for the recurrence T .n/ D

3T .bn=4c/ C ‚.n 2 /. We want to show that T .n/ dn 2

for some constant d > 0.

Using the same constant c > 0 as before, we have

T .n/ 3T .bn=4c/ C cn

2

3d bn=4c

2

C cn

2

3d.n=4/

2

C cn

2

D

3

16

dn

2

C cn

2

dn

2

;

where the last step holds as long as d .16=13/c.

In another, more intricate, example, Figure 4.6 shows the recursion tree for

T .n/ D T .n=3/ C T .2n=3/ C O.n/ :

(Again, we omit ﬂoor and ceiling functions for simplicity.) As before, we let c

represent the constant factor in the O.n/ term. When we add the values across the

levels of the recursion tree shown in the ﬁgure, we get a value of cn for every level.

92 Chapter 4 Divide-and-Conquer

The longest simple path from the root to a leaf is n ! .2=3/n ! .2=3/ 2 n !

! 1. Since .2=3/ k n D 1 when k D log

3=2

n, the height of the tree is log

3=2

n.

Intuitively, we expect the solution to the recurrence to be at most the number

of levels times the cost of each level, or O.cn log 3=2 n/ D O.n lg n/. Figure 4.6

shows only the top levels of the recursion tree, however, and not every level in the

tree contributes a cost of cn. Consider the cost of the leaves. If this recursion tree

were a complete binary tree of height log 3=2 n, there would be 2 log 3=2 n D n log 3=2 2

leaves. Since the cost of each leaf is a constant, the total cost of all leaves would

then be ‚.n log 3=2 2 / which, since log

3=2

2 is a constant strictly greater than 1,

is !.n lg n/. This recursion tree is not a complete binary tree, however, and so

it has fewer than n log 3=2 2

leaves. Moreover, as we go down from the root, more

and more internal nodes are absent. Consequently, levels toward the bottom of the

recursion tree contribute less than cn to the total cost. We could work out an accu-

rate accounting of all costs, but remember that we are just trying to come up with a

guess to use in the substitution method. Let us tolerate the sloppiness and attempt

to show that a guess of O.n lg n/ for the upper bound is correct.

Indeed, we can use the substitution method to verify that O.n lg n/ is an upper

bound for the solution to the recurrence. We show that T .n/ dn lg n, where d is

a suitable positive constant. We have

T .n/ T .n=3/ C T .2n=3/ C cn

d.n=3/ lg.n=3/ C d.2n=3/ lg.2n=3/ C cn

D .d.n=3/ lg n d.n=3/ lg 3/

C .d.2n=3/ lg n d.2n=3/ lg.3=2// C cn

D dn lg n d..n=3/ lg 3 C .2n=3/ lg.3=2// C cn

D dn lg n d..n=3/ lg 3 C .2n=3/ lg 3 .2n=3/ lg 2/ C cn

D dn lg n dn.lg 3 2=3/ C cn

dn lg n ;

as long as d c=.lg 3 .2=3//. Thus, we did not need to perform a more accurate

accounting of costs in the recursion tree.

Exercises

4.4-1

Use a recursion tree to determine a good asymptotic upper bound on the recurrence

T .n/ D 3T .bn=2c/ C n. Use the substitution method to verify your answer.

4.4-2

Use a recursion tree to determine a good asymptotic upper bound on the recurrence

T .n/ D T .n=2/ C n 2

. Use the substitution method to verify your answer.

4.5 The master method for solving recurrences 93

4.4-3

Use a recursion tree to determine a good asymptotic upper bound on the recurrence

T .n/ D 4T .n=2 C 2/ C n. Use the substitution method to verify your answer.

4.4-4

Use a recursion tree to determine a good asymptotic upper bound on the recurrence

T .n/ D 2T .n 1/ C 1. Use the substitution method to verify your answer.

4.4-5

Use a recursion tree to determine a good asymptotic upper bound on the recurrence

T .n/ D T .n1/CT .n=2/Cn. Use the substitution method to verify your answer.

4.4-6

Argue that the solution to the recurrence T .n/ D T .n=3/CT .2n=3/Ccn, where c

is a constant, is .n lg n/ by appealing to a recursion tree.

4.4-7

Draw the recursion tree for T .n/ D 4T .bn=2c/ C cn, where c is a constant, and

provide a tight asymptotic bound on its solution. Verify your bound by the substi-

tution method.

4.4-8

Use a recursion tree to give an asymptotically tight solution to the recurrence

T .n/ D T .n a/ C T .a/ C cn, where a 1 and c > 0 are constants.

4.4-9

Use a recursion tree to give an asymptotically tight solution to the recurrence

T .n/ D T .˛n/ C T ..1 ˛/n/ C cn, where ˛ is a constant in the range 0 < ˛ < 1

and c > 0 is also a constant.

4.5 The master method for solving recurrences

The master method provides a “cookbook” method for solving recurrences of the

form

T .n/ D aT .n=b/ C f .n/ ; (4.20)

where a 1 and b > 1 are constants and f .n/ is an asymptotically positive

function. To use the master method, you will need to memorize three cases, but

then you will be able to solve many recurrences quite easily, often without pencil

and paper.

94 Chapter 4 Divide-and-Conquer

The recurrence (4.20) describes the running time of an algorithm that divides a

problem of size n into a subproblems, each of size n=b, where a and b are positive

constants. The a subproblems are solved recursively, each in time T .n=b/. The

function f .n/ encompasses the cost of dividing the problem and combining the

results of the subproblems. For example, the recurrence arising from Strassen’s

algorithm has a D 7, b D 2, and f .n/ D ‚.n 2 /.

As a matter of technical correctness, the recurrence is not actually well deﬁned,

because n=b might not be an integer. Replacing each of the a terms T .n=b/ with

either T .bn=bc/ or T .dn=be/ will not affect the asymptotic behavior of the recur-

rence, however. (We will prove this assertion in the next section.) We normally

ﬁnd it convenient, therefore, to omit the ﬂoor and ceiling functions when writing

divide-and-conquer recurrences of this form.

The master theorem

The master method depends on the following theorem.

Theorem 4.1 (Master theorem)

Let a 1 and b > 1 be constants, let f .n/ be a function, and let T .n/ be deﬁned

on the nonnegative integers by the recurrence

T .n/ D aT .n=b/ C f .n/ ;

where we interpret n=b to mean either bn=bc or dn=be. Then T .n/ has the follow-

ing asymptotic bounds:

1. If f .n/ D O.n log b a / for some constant > 0, then T .n/ D ‚.n log b a /.

2. If f .n/ D ‚.n log b a /, then T .n/ D ‚.n log b a

lg n/.

3. If f .n/ D .n log b aC / for some constant > 0, and if af .n=b/ cf .n/ for

some constant c < 1 and all sufﬁciently large n, then T .n/ D ‚.f .n//.

Before applying the master theorem to some examples, let’s spend a moment

trying to understand what it says. In each of the three cases, we compare the

function f .n/ with the function n log b a

. Intuitively, the larger of the two functions

determines the solution to the recurrence. If, as in case 1, the function n log b a

is the

larger, then the solution is T .n/ D ‚.n log b a /. If, as in case 3, the function f .n/

is the larger, then the solution is T .n/ D ‚.f .n//. If, as in case 2, the two func-

tions are the same size, we multiply by a logarithmic factor, and the solution is

T .n/ D ‚.n log b a

lg n/ D ‚.f .n/ lg n/.

Beyond this intuition, you need to be aware of some technicalities. In the ﬁrst

case, not only must f .n/ be smaller than n log b a

, it must be polynomially smaller.

4.5 The master method for solving recurrences 95

That is, f .n/ must be asymptotically smaller than n log b a

by a factor of n

for some

constant > 0. In the third case, not only must f .n/ be larger than n log b a

, it also

must be polynomially larger and in addition satisfy the “regularity” condition that

af .n=b/ cf .n/. This condition is satisﬁed by most of the polynomially bounded

functions that we shall encounter.

Note that the three cases do not cover all the possibilities for f .n/. There is

a gap between cases 1 and 2 when f .n/ is smaller than n log b a

but not polynomi-

ally smaller. Similarly, there is a gap between cases 2 and 3 when f .n/ is larger

than n log b a

but not polynomially larger. If the function f .n/ falls into one of these

gaps, or if the regularity condition in case 3 fails to hold, you cannot use the master

method to solve the recurrence.

Using the master method

To use the master method, we simply determine which case (if any) of the master

theorem applies and write down the answer.

As a ﬁrst example, consider

T .n/ D 9T .n=3/ C n :

For this recurrence, we have a D 9, b D 3, f .n/ D n, and thus we have that

n log b a D n log 3 9 D ‚.n 2

). Since f .n/ D O.n log 3 9 /, where D 1, we can apply

case 1 of the master theorem and conclude that the solution is T .n/ D ‚.n 2 /.

Now consider

T .n/ D T .2n=3/ C 1;

in which a D 1, b D 3=2, f .n/ D 1, and n log b a D n log 3=2 1 D n 0 D 1. Case 2

applies, since f .n/ D ‚.n log b a / D ‚.1/, and thus the solution to the recurrence

is T .n/ D ‚.lg n/.

For the recurrence

T .n/ D 3T .n=4/ C n lg n ;

we have a D 3, b D 4, f .n/ D n lg n, and n log b a D n log 4 3 D O.n 0:793 /.

Since f .n/ D .n log 4 3C /, where 0:2, case 3 applies if we can show that

the regularity condition holds for f .n/. For sufﬁciently large n, we have that

af .n=b/ D 3.n=4/ lg.n=4/ .3=4/n lg n D cf .n/ for c D 3=4. Consequently,

by case 3, the solution to the recurrence is T .n/ D ‚.n lg n/.

The master method does not apply to the recurrence

T .n/ D 2T .n=2/ C n lg n ;

even though it appears to have the proper form: a D 2, b D 2, f .n/ D n lg n,

and n log b a D n. You might mistakenly think that case 3 should apply, since

96 Chapter 4 Divide-and-Conquer

f .n/ D n lg n is asymptotically larger than n log b a D n. The problem is that it

is not polynomially larger. The ratio f .n/=n log b a D .n lg n/=n D lg n is asymp-

totically less than n

for any positive constant . Consequently, the recurrence falls

into the gap between case 2 and case 3. (See Exercise 4.6-2 for a solution.)

Let’s use the master method to solve the recurrences we saw in Sections 4.1

and 4.2. Recurrence (4.7),

T .n/ D 2T .n=2/ C ‚.n/ ;

characterizes the running times of the divide-and-conquer algorithm for both the

maximum-subarray problem and merge sort. (As is our practice, we omit stating

the base case in the recurrence.) Here, we have a D 2, b D 2, f .n/ D ‚.n/, and

thus we have that n log b a D n log 2 2 D n. Case 2 applies, since f .n/ D ‚.n/, and so

we have the solution T .n/ D ‚.n lg n/.

Recurrence (4.17),

T .n/ D 8T .n=2/ C ‚.n

2

/ ;

describes the running time of the ﬁrst divide-and-conquer algorithm that we saw

for matrix multiplication. Now we have a D 8, b D 2, and f .n/ D ‚.n 2 /,

and so n log b a D n log 2 8 D n 3

. Since n 3

is polynomially larger than f .n/ (that is,

f .n/ D O.n 3 / for D 1), case 1 applies, and T .n/ D ‚.n 3 /.

Finally, consider recurrence (4.18),

T .n/ D 7T .n=2/ C ‚.n

2

/ ;

which describes the running time of Strassen’s algorithm. Here, we have a D 7,

b D 2, f .n/ D ‚.n 2 /, and thus n log b a D n log 2 7

. Rewriting log

2

7 as lg 7 and

recalling that 2:80 < lg 7 < 2:81, we see that f .n/ D O.n lg 7 / for D 0:8.

Again, case 1 applies, and we have the solution T .n/ D ‚.n lg 7 /.

Exercises

4.5-1

Use the master method to give tight asymptotic bounds for the following recur-

rences.

a. T .n/ D 2T .n=4/ C 1.

b. T .n/ D 2T .n=4/ C

p

n.

c. T .n/ D 2T .n=4/ C n.

d. T .n/ D 2T .n=4/ C n 2

.

4.6 Proof of the master theorem 97

4.5-2

Professor Caesar wishes to develop a matrix-multiplication algorithm that is

asymptotically faster than Strassen’s algorithm. His algorithm will use the divide-

and-conquer method, dividing each matrix into pieces of size n=4 n=4, and the

divide and combine steps together will take ‚.n 2 / time. He needs to determine

how many subproblems his algorithm has to create in order to beat Strassen’s algo-

rithm. If his algorithm creates a subproblems, then the recurrence for the running

time T .n/ becomes T .n/ D aT .n=4/ C ‚.n 2 /. What is the largest integer value

of a for which Professor Caesar’s algorithm would be asymptotically faster than

Strassen’s algorithm?

4.5-3

Use the master method to show that the solution to the binary-search recurrence

T .n/ D T .n=2/ C ‚.1/ is T .n/ D ‚.lg n/. (See Exercise 2.3-5 for a description

of binary search.)

4.5-4

Can the master method be applied to the recurrence T .n/ D 4T .n=2/ C n 2

lg n?

Why or why not? Give an asymptotic upper bound for this recurrence.

4.5-5 ?

Consider the regularity condition af .n=b/ cf .n/ for some constant c < 1,

which is part of case 3 of the master theorem. Give an example of constants a 1

and b > 1 and a function f .n/ that satisﬁes all the conditions in case 3 of the

master theorem except the regularity condition.

? 4.6 Proof of the master theorem

This section contains a proof of the master theorem (Theorem 4.1). You do not

need to understand the proof in order to apply the master theorem.

The proof appears in two parts. The ﬁrst part analyzes the master recur-

rence (4.20), under the simplifying assumption that T .n/ is deﬁned only on ex-

act powers of b > 1, that is, for n D 1; b; b 2 ; : : :. This part gives all the intuition

needed to understand why the master theorem is true. The second part shows how

to extend the analysis to all positive integers n; it applies mathematical technique

to the problem of handling ﬂoors and ceilings.

In this section, we shall sometimes abuse our asymptotic notation slightly by

using it to describe the behavior of functions that are deﬁned only over exact

powers of b. Recall that the deﬁnitions of asymptotic notations require that

98 Chapter 4 Divide-and-Conquer

bounds be proved for all sufﬁciently large numbers, not just those that are pow-

ers of b. Since we could make new asymptotic notations that apply only to the set

fb i W i D 0; 1; 2; : : :g, instead of to the nonnegative numbers, this abuse is minor.

Nevertheless, we must always be on guard when we use asymptotic notation over

a limited domain lest we draw improper conclusions. For example, proving that

T .n/ D O.n/ when n is an exact power of 2 does not guarantee that T .n/ D O.n/.

The function T .n/ could be deﬁned as

T .n/ D

(

n if n D 1; 2; 4; 8; : : : ;

n 2

otherwise ;

in which case the best upper bound that applies to all values of n is T .n/ D O.n 2 /.

Because of this sort of drastic consequence, we shall never use asymptotic notation

over a limited domain without making it absolutely clear from the context that we

are doing so.

4.6.1 The proof for exact powers

The ﬁrst part of the proof of the master theorem analyzes the recurrence (4.20)

T .n/ D aT .n=b/ C f .n/ ;

for the master method, under the assumption that n is an exact power of b > 1,

where b need not be an integer. We break the analysis into three lemmas. The ﬁrst

reduces the problem of solving the master recurrence to the problem of evaluating

an expression that contains a summation. The second determines bounds on this

summation. The third lemma puts the ﬁrst two together to prove a version of the

master theorem for the case in which n is an exact power of b.

Lemma 4.2

Let a 1 and b > 1 be constants, and let f .n/ be a nonnegative function deﬁned

on exact powers of b. Deﬁne T .n/ on exact powers of b by the recurrence

T .n/ D

(

‚.1/ if n D 1 ;

aT .n=b/ C f .n/ if n D b i ;

where i is a positive integer. Then

T .n/ D ‚.n log b a

/ C

log b n1 X

j D0

a

j

f .n=b

j

/ : (4.21)

Proof We use the recursion tree in Figure 4.7. The root of the tree has cost f .n/,

and it has a children, each with cost f .n=b/. (It is convenient to think of a as being

4.6 Proof of the master theorem 99

…

…

…

… … …

…

… … …

…

… … …

…

f .n/ f .n/

a a a

a

a a a

a

a a a

a

a

f .n=b/ f .n=b/ f .n=b/

f .n=b 2 / f .n=b 2 / f .n=b 2 / f .n=b 2 / f .n=b 2 / f .n=b 2 / f .n=b 2 / f .n=b 2 / f .n=b 2 /

af .n=b/

a 2 f .n=b 2 /

log

b

n

n log b a

‚.1/ ‚.1/ ‚.1/ ‚.1/ ‚.1/ ‚.1/ ‚.1/ ‚.1/ ‚.1/ ‚.1/ ‚.1/ ‚.1/ ‚.1/ ‚.n log b a /

Total: ‚.n log b a

/ C

log b n1 X

j D0

a

j

f .n=b

j

/

Figure 4.7 The recursion tree generated by T .n/ D aT .n=b/ C f .n/. The tree is a complete a-ary

tree with n

log b a

leaves and height log

b

n. The cost of the nodes at each depth is shown at the right,

and their sum is given in equation (4.21).

an integer, especially when visualizing the recursion tree, but the mathematics does

not require it.) Each of these children has a children, making a 2

nodes at depth 2,

and each of the a children has cost f .n=b 2 /. In general, there are a j

nodes at

depth j , and each has cost f .n=b j /. The cost of each leaf is T .1/ D ‚.1/, and

each leaf is at depth log

b

n, since n=b log b n D 1. There are a log b n D n log b a

leaves

in the tree.

We can obtain equation (4.21) by summing the costs of the nodes at each depth

in the tree, as shown in the ﬁgure. The cost for all internal nodes at depth j is

a j f .n=b j /, and so the total cost of all internal nodes is

log b n1 X

j D0

a

j

f .n=b

j

/ :

In the underlying divide-and-conquer algorithm, this sum represents the costs of

dividing problems into subproblems and then recombining the subproblems. The

100 Chapter 4 Divide-and-Conquer

cost of all the leaves, which is the cost of doing all n log b a

subproblems of size 1,

is ‚.n log b a /.

In terms of the recursion tree, the three cases of the master theorem correspond

to cases in which the total cost of the tree is (1) dominated by the costs in the

leaves, (2) evenly distributed among the levels of the tree, or (3) dominated by the

cost of the root.

The summation in equation (4.21) describes the cost of the dividing and com-

bining steps in the underlying divide-and-conquer algorithm. The next lemma pro-

vides asymptotic bounds on the summation’s growth.

Lemma 4.3

Let a 1 and b > 1 be constants, and let f .n/ be a nonnegative function deﬁned

on exact powers of b. A function g.n/ deﬁned over exact powers of b by

g.n/ D

log b n1 X

j D0

a

j

f .n=b

j

/ (4.22)

has the following asymptotic bounds for exact powers of b:

1. If f .n/ D O.n log b a / for some constant > 0, then g.n/ D O.n log b a /.

2. If f .n/ D ‚.n log b a /, then g.n/ D ‚.n log b a

lg n/.

3. If af .n=b/ cf .n/ for some constant c < 1 and for all sufﬁciently large n,

then g.n/ D ‚.f .n//.

Proof For case 1, we have f .n/ D O.n log b a /, which implies that f .n=b j / D

O..n=b j / log b a /. Substituting into equation (4.22) yields

g.n/ D O

log b n1 X

j D0

a

j

n

b j

log b a

!

: (4.23)

We bound the summation within the O-notation by factoring out terms and simpli-

fying, which leaves an increasing geometric series:

log b n1 X

j D0

a

j

n

b j

log b a

D n log b a

log b n1 X

j D0

ab

b log b a

j

D n log b a

log b n1 X

j D0

.b

/

j

D n log b a

b log b n 1

b 1

4.6 Proof of the master theorem 101

D n log b a

n 1

b 1

:

Since b and are constants, we can rewrite the last expression as n log b a O.n / D

O.n log b a /. Substituting this expression for the summation in equation (4.23) yields

g.n/ D O.n log b a

/ ;

thereby proving case 1.

Because case 2 assumes that f .n/ D ‚.n log b a /, we have that f .n=b j / D

‚..n=b j / log b a /. Substituting into equation (4.22) yields

g.n/ D ‚

log b n1 X

j D0

a

j

n

b j

log b a

!

: (4.24)

We bound the summation within the ‚-notation as in case 1, but this time we do not

obtain a geometric series. Instead, we discover that every term of the summation

is the same:

log b n1 X

j D0

a

j

n

b j

log b a

D n log b a

log b n1 X

j D0

a

b log b a

j

D n log b a

log b n1 X

j D0

1

D n log b a

log b n :

Substituting this expression for the summation in equation (4.24) yields

g.n/ D ‚.n log b a

log

b

n/

D ‚.n log b a

lg n/ ;

proving case 2.

We prove case 3 similarly. Since f .n/ appears in the deﬁnition (4.22) of g.n/

and all terms of g.n/ are nonnegative, we can conclude that g.n/ D .f .n// for

exact powers of b. We assume in the statement of the lemma that af .n=b/ cf .n/

for some constant c < 1 and all sufﬁciently large n. We rewrite this assumption

as f .n=b/ .c=a/f .n/ and iterate j times, yielding f .n=b j / .c=a/ j f .n/ or,

equivalently, a j f .n=b j / c j f .n/, where we assume that the values we iterate

on are sufﬁciently large. Since the last, and smallest, such value is n=b j 1

, it is

enough to assume that n=b j 1

is sufﬁciently large.

Substituting into equation (4.22) and simplifying yields a geometric series, but

unlike the series in case 1, this one has decreasing terms. We use an O.1/ term to

102 Chapter 4 Divide-and-Conquer

capture the terms that are not covered by our assumption that n is sufﬁciently large:

g.n/ D

log b n1 X

j D0

a

j

f .n=b

j

/

log b n1 X

j D0

c

j

f .n/ C O.1/

f .n/

1 X

j D0

c

j

C O.1/

D f .n/

1

1 c

C O.1/

D O.f .n// ;

since c is a constant. Thus, we can conclude that g.n/ D ‚.f .n// for exact powers

of b. With case 3 proved, the proof of the lemma is complete.

We can now prove a version of the master theorem for the case in which n is an

exact power of b.

Lemma 4.4

Let a 1 and b > 1 be constants, and let f .n/ be a nonnegative function deﬁned

on exact powers of b. Deﬁne T .n/ on exact powers of b by the recurrence

T .n/ D

(

‚.1/ if n D 1 ;

aT .n=b/ C f .n/ if n D b i ;

where i is a positive integer. Then T .n/ has the following asymptotic bounds for

exact powers of b:

1. If f .n/ D O.n log b a / for some constant > 0, then T .n/ D ‚.n log b a /.

2. If f .n/ D ‚.n log b a /, then T .n/ D ‚.n log b a

lg n/.

3. If f .n/ D .n log b aC / for some constant > 0, and if af .n=b/ cf .n/ for

some constant c < 1 and all sufﬁciently large n, then T .n/ D ‚.f .n//.

Proof We use the bounds in Lemma 4.3 to evaluate the summation (4.21) from

Lemma 4.2. For case 1, we have

T .n/ D ‚.n log b a

/ C O.n log b a

/

D ‚.n log b a

/ ;

4.6 Proof of the master theorem 103

and for case 2,

T .n/ D ‚.n log b a

/ C ‚.n log b a

lg n/

D ‚.n log b a

lg n/ :

For case 3,

T .n/ D ‚.n log b a

/ C ‚.f .n//

D ‚.f .n// ;

because f .n/ D .n log b aC /.

4.6.2 Floors and ceilings

To complete the proof of the master theorem, we must now extend our analysis to

the situation in which ﬂoors and ceilings appear in the master recurrence, so that

the recurrence is deﬁned for all integers, not for just exact powers of b. Obtaining

a lower bound on

T .n/ D aT .dn=be/ C f .n/ (4.25)

and an upper bound on

T .n/ D aT .bn=bc/ C f .n/ (4.26)

is routine, since we can push through the bound dn=be n=b in the ﬁrst case to

yield the desired result, and we can push through the bound bn=bc n=b in the

second case. We use much the same technique to lower-bound the recurrence (4.26)

as to upper-bound the recurrence (4.25), and so we shall present only this latter

bound.

We modify the recursion tree of Figure 4.7 to produce the recursion tree in Fig-

ure 4.8. As we go down in the recursion tree, we obtain a sequence of recursive

invocations on the arguments

n ;

dn=be ;

ddn=be =be ;

dddn=be =be =be ;

:

:

:

Let us denote the j th element in the sequence by n j , where

n j D

(

n if j D 0 ;

dn j 1 =be if j > 0 :

(4.27)

104 Chapter 4 Divide-and-Conquer

…

…

…

… … …

…

… … …

…

… … …

…

f .n/ f .n/

a a a

a

a a a

a

a a a

a

a

f .n 1 / f .n 1 / f .n 1 /

f .n 2 / f .n 2 / f .n 2 / f .n 2 / f .n 2 / f .n 2 / f .n 2 / f .n 2 / f .n 2 /

af .n 1 /

a 2 f .n 2 /

blog

b

nc

‚.n log b a /

‚.1/ ‚.1/ ‚.1/ ‚.1/ ‚.1/ ‚.1/ ‚.1/ ‚.1/ ‚.1/ ‚.1/ ‚.1/ ‚.1/ ‚.1/ ‚.n log b a /

Total: ‚.n log b a

/ C

blog b nc1 X

j D0

a

j

f .n j /

Figure 4.8 The recursion tree generated by T .n/ D aT .dn=be/Cf .n/. The recursive argument nj

is given by equation (4.27).

Our ﬁrst goal is to determine the depth k such that n k is a constant. Using the

inequality dxe x C 1, we obtain

n 0 n ;

n 1

n

b

C 1 ;

n 2

n

b 2

C

1

b

C 1 ;

n 3

n

b 3

C

1

b 2

C

1

b

C 1 ;

:

:

:

In general, we have

4.6 Proof of the master theorem 105

n j

n

b j

C

j 1 X

iD0

1

b i

<

n

b j

C

1 X

iD0

1

b i

D

n

b j

C

b

b 1

:

Letting j D blog

b

nc, we obtain

n blog b nc <

n

b blog b nc

C

b

b 1

<

n

b log b n1

C

b

b 1

D

n

n=b

C

b

b 1

D b C

b

b 1

D O.1/ ;

and thus we see that at depth blog

b

nc, the problem size is at most a constant.

From Figure 4.8, we see that

T .n/ D ‚.n log b a

/ C

blog b nc1 X

j D0

a

j

f .n j / ; (4.28)

which is much the same as equation (4.21), except that n is an arbitrary integer and

not restricted to be an exact power of b.

We can now evaluate the summation

g.n/ D

blog b nc1 X

j D0

a

j

f .n j / (4.29)

from equation (4.28) in a manner analogous to the proof of Lemma 4.3. Beginning

with case 3, if af .dn=be/ cf .n/ for n > bCb=.b1/, where c < 1 is a constant,

then it follows that a j f .n j / c j f .n/. Therefore, we can evaluate the sum in

equation (4.29) just as in Lemma 4.3. For case 2, we have f .n/ D ‚.n log b a /. If we

can show that f .n j / D O.n log b a =a j / D O..n=b j / log b a /, then the proof for case 2

of Lemma 4.3 will go through. Observe that j blog

b

nc implies b j =n 1. The

bound f .n/ D O.n log b a / implies that there exists a constant c > 0 such that for all

sufﬁciently large n j ,

106 Chapter 4 Divide-and-Conquer

f .n j / c

n

b j

C

b

b 1

log b a

D c

n

b j

1 C

b j

n

b

b 1

log b a

D c

n log b a

a j

1 C

b j

n

b

b 1

log b a

c

n log b a

a j

1 C

b

b 1

log b a

D O

n log b a

a j

;

since c.1 C b=.b 1// log b a

is a constant. Thus, we have proved case 2. The proof

of case 1 is almost identical. The key is to prove the bound f .n j / D O.n log b a /,

which is similar to the corresponding proof of case 2, though the algebra is more

intricate.

We have now proved the upper bounds in the master theorem for all integers n.

The proof of the lower bounds is similar.

Exercises

4.6-1 ?

Give a simple and exact expression for n j in equation (4.27) for the case in which b

is a positive integer instead of an arbitrary real number.

4.6-2 ?

Show that if f .n/ D ‚.n log b a

lg

k

n/, where k 0, then the master recurrence has

solution T .n/ D ‚.n log b a

lg

kC1

n/. For simplicity, conﬁne your analysis to exact

powers of b.

4.6-3 ?

Show that case 3 of the master theorem is overstated, in the sense that the regularity

condition af .n=b/ cf .n/ for some constant c < 1 implies that there exists a

constant > 0 such that f .n/ D .n log b aC /.

Problems for Chapter 4 107

Problems

4-1 Recurrence examples

Give asymptotic upper and lower bounds for T .n/ in each of the following recur-

rences. Assume that T .n/ is constant for n 2. Make your bounds as tight as

possible, and justify your answers.

a. T .n/ D 2T .n=2/ C n 4

.

b. T .n/ D T .7n=10/ C n.

c. T .n/ D 16T .n=4/ C n 2

.

d. T .n/ D 7T .n=3/ C n 2

.

e. T .n/ D 7T .n=2/ C n 2

.

f. T .n/ D 2T .n=4/ C

p

n.

g. T .n/ D T .n 2/ C n 2

.

4-2 Parameter-passing costs

Throughout this book, we assume that parameter passing during procedure calls

takes constant time, even if an N -element array is being passed. This assumption

is valid in most systems because a pointer to the array is passed, not the array itself.

This problem examines the implications of three parameter-passing strategies:

1. An array is passed by pointer. Time D ‚.1/.

2. An array is passed by copying. Time D ‚.N /, where N is the size of the array.

3. An array is passed by copying only the subrange that might be accessed by the

called procedure. Time D ‚.q p C 1/ if the subarray AŒp : : q is passed.

a. Consider the recursive binary search algorithm for ﬁnding a number in a sorted

array (see Exercise 2.3-5). Give recurrences for the worst-case running times

of binary search when arrays are passed using each of the three methods above,

and give good upper bounds on the solutions of the recurrences. Let N be the

size of the original problem and n be the size of a subproblem.

b. Redo part (a) for the MERGE-SORT algorithm from Section 2.3.1.

108 Chapter 4 Divide-and-Conquer

4-3 More recurrence examples

Give asymptotic upper and lower bounds for T .n/ in each of the following recur-

rences. Assume that T .n/ is constant for sufﬁciently small n. Make your bounds

as tight as possible, and justify your answers.

a. T .n/ D 4T .n=3/ C n lg n.

b. T .n/ D 3T .n=3/ C n= lg n.

c. T .n/ D 4T .n=2/ C n 2

p

n.

d. T .n/ D 3T .n=3 2/ C n=2.

e. T .n/ D 2T .n=2/ C n= lg n.

f. T .n/ D T .n=2/ C T .n=4/ C T .n=8/ C n.

g. T .n/ D T .n 1/ C 1=n.

h. T .n/ D T .n 1/ C lg n.

i. T .n/ D T .n 2/ C 1= lg n.

j. T .n/ D

p

nT .

p

n/ C n.

4-4 Fibonacci numbers

This problem develops properties of the Fibonacci numbers, which are deﬁned

by recurrence (3.22). We shall use the technique of generating functions to solve

the Fibonacci recurrence. Deﬁne the generating function (or formal power se-

ries) F as

F .´/ D

1 X

iD0

F i ´

i

D 0 C ´ C ´

2

C 2´

3

C 3´

4

C 5´

5

C 8´

6

C 13´

7

C 21´

8

C ;

where F i is the ith Fibonacci number.

a. Show that F .´/ D ´ C ´ F .´/ C ´ 2 F .´/.

Problems for Chapter 4 109

b. Show that

F .´/ D

´

1 ´ ´ 2

D

´

.1 ´/.1 y´/

D

1

p

5

1

1 ´

1

1 y´

;

where

D

1 C

p

5

2

D 1:61803 : : :

and

y D

1

p

5

2

D 0:61803 : : : :

c. Show that

F .´/ D

1 X

iD0

1

p

5

.

i

y

i

/´

i

:

d. Use part (c) to prove that F i D i =

p

5 for i > 0, rounded to the nearest integer.

(Hint: Observe that

ˇ

ˇ y

ˇ

ˇ < 1.)

4-5 Chip testing

Professor Diogenes has n supposedly identical integrated-circuit chips that in prin-

ciple are capable of testing each other. The professor’s test jig accommodates two

chips at a time. When the jig is loaded, each chip tests the other and reports whether

it is good or bad. A good chip always reports accurately whether the other chip is

good or bad, but the professor cannot trust the answer of a bad chip. Thus, the four

possible outcomes of a test are as follows:

Chip A says Chip B says Conclusion

B is good A is good both are good, or both are bad

B is good A is bad at least one is bad

B is bad A is good at least one is bad

B is bad A is bad at least one is bad

a. Show that if more than n=2 chips are bad, the professor cannot necessarily de-

termine which chips are good using any strategy based on this kind of pairwise

test. Assume that the bad chips can conspire to fool the professor.

110 Chapter 4 Divide-and-Conquer

b. Consider the problem of ﬁnding a single good chip from among n chips, as-

suming that more than n=2 of the chips are good. Show that bn=2c pairwise

tests are sufﬁcient to reduce the problem to one of nearly half the size.

c. Show that the good chips can be identiﬁed with ‚.n/ pairwise tests, assuming

that more than n=2 of the chips are good. Give and solve the recurrence that

describes the number of tests.

4-6 Monge arrays

An m n array A of real numbers is a Monge array if for all i, j , k, and l such

that 1 i < k m and 1 j < l n, we have

AŒi; j C AŒk; l AŒi; l C AŒk; j :

In other words, whenever we pick two rows and two columns of a Monge array and

consider the four elements at the intersections of the rows and the columns, the sum

of the upper-left and lower-right elements is less than or equal to the sum of the

lower-left and upper-right elements. For example, the following array is Monge:

10 17 13 28 23

17 22 16 29 23

24 28 22 34 24

11 13 6 17 7

45 44 32 37 23

36 33 19 21 6

75 66 51 53 34

a. Prove that an array is Monge if and only if for all i D 1; 2; :::; m 1 and

j D 1; 2; :::; n 1, we have

AŒi; j C AŒi C 1; j C 1 AŒi; j C 1 C AŒi C 1; j :

(Hint: For the “if” part, use induction separately on rows and columns.)

b. The following array is not Monge. Change one element in order to make it

Monge. (Hint: Use part (a).)

37 23 22 32

21 6 7 10

53 34 30 31

32 13 9 6

43 21 15 8

Notes for Chapter 4 111

c. Let f .i/ be the index of the column containing the leftmost minimum element

of row i. Prove that f .1/ f .2/ f .m/ for any m n Monge array.

d. Here is a description of a divide-and-conquer algorithm that computes the left-

most minimum element in each row of an m n Monge array A:

Construct a submatrix A 0

of A consisting of the even-numbered rows of A.

Recursively determine the leftmost minimum for each row of A 0

. Then

compute the leftmost minimum in the odd-numbered rows of A.

Explain how to compute the leftmost minimum in the odd-numbered rows of A

(given that the leftmost minimum of the even-numbered rows is known) in

O.m C n/ time.

e. Write the recurrence describing the running time of the algorithm described in

part (d). Show that its solution is O.m C n log m/.

Chapter notes

Divide-and-conquer as a technique for designing algorithms dates back to at least

1962 in an article by Karatsuba and Ofman [194]. It might have been used well be-

fore then, however; according to Heideman, Johnson, and Burrus [163], C. F. Gauss

devised the ﬁrst fast Fourier transform algorithm in 1805, and Gauss’s formulation

breaks the problem into smaller subproblems whose solutions are combined.

The maximum-subarray problem in Section 4.1 is a minor variation on a problem

studied by Bentley [43, Chapter 7].

Strassen’s algorithm [325] caused much excitement when it was published

in 1969. Before then, few imagined the possibility of an algorithm asymptotically

faster than the basic SQUARE-MATRIX-MULTIPLY procedure. The asymptotic

upper bound for matrix multiplication has been improved since then. The most

asymptotically efﬁcient algorithm for multiplying n n matrices to date, due to

Coppersmith and Winograd [78], has a running time of O.n 2:376 /. The best lower

bound known is just the obvious .n 2 / bound (obvious because we must ﬁll in n 2

elements of the product matrix).

From a practical point of view, Strassen’s algorithm is often not the method of

choice for matrix multiplication, for four reasons:

1. The constant factor hidden in the ‚.n lg 7 / running time of Strassen’s algo-

rithm is larger than the constant factor in the ‚.n 3 /-time SQUARE-MATRIX-

MULTIPLY procedure.

2. When the matrices are sparse, methods tailored for sparse matrices are faster.

112 Chapter 4 Divide-and-Conquer

3. Strassen’s algorithm is not quite as numerically stable as SQUARE-MATRIX-

MULTIPLY. In other words, because of the limited precision of computer arith-

metic on noninteger values, larger errors accumulate in Strassen’s algorithm

than in SQUARE-MATRIX-MULTIPLY.

4. The submatrices formed at the levels of recursion consume space.

The latter two reasons were mitigated around 1990. Higham [167] demonstrated

that the difference in numerical stability had been overemphasized; although

Strassen’s algorithm is too numerically unstable for some applications, it is within

acceptable limits for others. Bailey, Lee, and Simon [32] discuss techniques for

reducing the memory requirements for Strassen’s algorithm.

In practice, fast matrix-multiplication implementations for dense matrices use

Strassen’s algorithm for matrix sizes above a “crossover point,” and they switch

to a simpler method once the subproblem size reduces to below the crossover

point. The exact value of the crossover point is highly system dependent. Analyses

that count operations but ignore effects from caches and pipelining have produced

crossover points as low as n D 8 (by Higham [167]) or n D 12 (by Huss-Lederman

et al. [186]). D’Alberto and Nicolau [81] developed an adaptive scheme, which

determines the crossover point by benchmarking when their software package is

installed. They found crossover points on various systems ranging from n D 400

to n D 2150, and they could not ﬁnd a crossover point on a couple of systems.

Recurrences were studied as early as 1202 by L. Fibonacci, for whom the Fi-

bonacci numbers are named. A. De Moivre introduced the method of generating

functions (see Problem 4-4) for solving recurrences. The master method is adapted

from Bentley, Haken, and Saxe [44], which provides the extended method justiﬁed

by Exercise 4.6-2. Knuth [209] and Liu [237] show how to solve linear recurrences

using the method of generating functions. Purdom and Brown [287] and Graham,

Knuth, and Patashnik [152] contain extended discussions of recurrence solving.

Several researchers, including Akra and Bazzi [13], Roura [299], Verma [346],

and Yap [360], have given methods for solving more general divide-and-conquer

recurrences than are solved by the master method. We describe the result of Akra

and Bazzi here, as modiﬁed by Leighton [228]. The Akra-Bazzi method works for

recurrences of the form

T .x/ D

(

‚.1/ if 1 x x 0 ;

P k

iD1

a i T .b i x/ C f .x/ if x > x 0 ;

(4.30)

where

x 1 is a real number,

x 0 is a constant such that x 0 1=b i and x 0 1=.1 b i / for i D 1; 2; : : : ; k,

a i is a positive constant for i D 1; 2; : : : ; k,

Notes for Chapter 4 113

b i is a constant in the range 0 < b i < 1 for i D 1; 2; : : : ; k,

k 1 is an integer constant, and

f .x/ is a nonnegative function that satisﬁes the polynomial-growth condi-

tion: there exist positive constants c 1 and c 2 such that for all x 1, for

i D 1; 2; : : : ; k, and for all u such that b i x u x, we have c 1 f .x/

f .u/ c 2 f .x/. (If jf 0 .x/j is upper-bounded by some polynomial in x, then

f .x/ satisﬁes the polynomial-growth condition. For example, f .x/ D x ˛

lg

ˇ

x

satisﬁes this condition for any real constants ˛ and ˇ.)

Although the master method does not apply to a recurrence such as T .n/ D

T .bn=3c/ C T .b2n=3c/ C O.n/, the Akra-Bazzi method does. To solve the re-

currence (4.30), we ﬁrst ﬁnd the unique real number p such that

P k

iD1

a i b

p

i

D 1.

(Such a p always exists.) The solution to the recurrence is then

T .n/ D ‚

x

p

1 C

Z x

1

f .u/

u pC1

du

:

The Akra-Bazzi method can be somewhat difﬁcult to use, but it serves in solving

recurrences that model division of the problem into substantially unequally sized

subproblems. The master method is simpler to use, but it applies only when sub-

problem sizes are equal.

5 Probabilistic Analysis and Randomized

Algorithms

This chapter introduces probabilistic analysis and randomized algorithms. If you

are unfamiliar with the basics of probability theory, you should read Appendix C,

which reviews this material. We shall revisit probabilistic analysis and randomized

algorithms several times throughout this book.

5.1 The hiring problem

Suppose that you need to hire a new ofﬁce assistant. Your previous attempts at

hiring have been unsuccessful, and you decide to use an employment agency. The

employment agency sends you one candidate each day. You interview that person

and then decide either to hire that person or not. You must pay the employment

agency a small fee to interview an applicant. To actually hire an applicant is more

costly, however, since you must ﬁre your current ofﬁce assistant and pay a substan-

tial hiring fee to the employment agency. You are committed to having, at all times,

the best possible person for the job. Therefore, you decide that, after interviewing

each applicant, if that applicant is better qualiﬁed than the current ofﬁce assistant,

you will ﬁre the current ofﬁce assistant and hire the new applicant. You are willing

to pay the resulting price of this strategy, but you wish to estimate what that price

will be.

The procedure HIRE-ASSISTANT, given below, expresses this strategy for hiring

in pseudocode. It assumes that the candidates for the ofﬁce assistant job are num-

bered 1 through n. The procedure assumes that you are able to, after interviewing

candidate i, determine whether candidate i is the best candidate you have seen so

far. To initialize, the procedure creates a dummy candidate, numbered 0, who is

less qualiﬁed than each of the other candidates.

5.1 The hiring problem 115

HIRE-ASSISTANT.n/

1 best D 0 // candidate 0 is a least-qualiﬁed dummy candidate

2 for i D 1 to n

3 interview candidate i

4 if candidate i is better than candidate best

5 best D i

6 hire candidate i

The cost model for this problem differs from the model described in Chapter 2.

We focus not on the running time of HIRE-ASSISTANT, but instead on the costs

incurred by interviewing and hiring. On the surface, analyzing the cost of this algo-

rithm may seem very different from analyzing the running time of, say, merge sort.

The analytical techniques used, however, are identical whether we are analyzing

cost or running time. In either case, we are counting the number of times certain

basic operations are executed.

Interviewing has a low cost, say c i , whereas hiring is expensive, costing c h . Let-

ting m be the number of people hired, the total cost associated with this algorithm

is O.c i n C c h m/. No matter how many people we hire, we always interview n

candidates and thus always incur the cost c i n associated with interviewing. We

therefore concentrate on analyzing c h m, the hiring cost. This quantity varies with

each run of the algorithm.

This scenario serves as a model for a common computational paradigm. We of-

ten need to ﬁnd the maximum or minimum value in a sequence by examining each

element of the sequence and maintaining a current “winner.” The hiring problem

models how often we update our notion of which element is currently winning.

Worst-case analysis

In the worst case, we actually hire every candidate that we interview. This situation

occurs if the candidates come in strictly increasing order of quality, in which case

we hire n times, for a total hiring cost of O.c h n/.

Of course, the candidates do not always come in increasing order of quality. In

fact, we have no idea about the order in which they arrive, nor do we have any

control over this order. Therefore, it is natural to ask what we expect to happen in

a typical or average case.

Probabilistic analysis

Probabilistic analysis is the use of probability in the analysis of problems. Most

commonly, we use probabilistic analysis to analyze the running time of an algo-

rithm. Sometimes we use it to analyze other quantities, such as the hiring cost

116 Chapter 5 Probabilistic Analysis and Randomized Algorithms

in procedure HIRE-ASSISTANT. In order to perform a probabilistic analysis, we

must use knowledge of, or make assumptions about, the distribution of the inputs.

Then we analyze our algorithm, computing an average-case running time, where

we take the average over the distribution of the possible inputs. Thus we are, in

effect, averaging the running time over all possible inputs. When reporting such a

running time, we will refer to it as the average-case running time.

We must be very careful in deciding on the distribution of inputs. For some

problems, we may reasonably assume something about the set of all possible in-

puts, and then we can use probabilistic analysis as a technique for designing an

efﬁcient algorithm and as a means for gaining insight into a problem. For other

problems, we cannot describe a reasonable input distribution, and in these cases

we cannot use probabilistic analysis.

For the hiring problem, we can assume that the applicants come in a random

order. What does that mean for this problem? We assume that we can compare

any two candidates and decide which one is better qualiﬁed; that is, there is a

total order on the candidates. (See Appendix B for the deﬁnition of a total or-

der.) Thus, we can rank each candidate with a unique number from 1 through n,

using rank.i/ to denote the rank of applicant i, and adopt the convention that a

higher rank corresponds to a better qualiﬁed applicant. The ordered list hrank.1/;

rank.2/; : : : ; rank.n/i is a permutation of the list h1; 2; : : : ; ni. Saying that the

applicants come in a random order is equivalent to saying that this list of ranks is

equally likely to be any one of the nŠ permutations of the numbers 1 through n.

Alternatively, we say that the ranks form a uniform random permutation; that is,

each of the possible nŠ permutations appears with equal probability.

Section 5.2 contains a probabilistic analysis of the hiring problem.

Randomized algorithms

In order to use probabilistic analysis, we need to know something about the distri-

bution of the inputs. In many cases, we know very little about the input distribution.

Even if we do know something about the distribution, we may not be able to model

this knowledge computationally. Yet we often can use probability and randomness

as a tool for algorithm design and analysis, by making the behavior of part of the

algorithm random.

In the hiring problem, it may seem as if the candidates are being presented to us

in a random order, but we have no way of knowing whether or not they really are.

Thus, in order to develop a randomized algorithm for the hiring problem, we must

have greater control over the order in which we interview the candidates. We will,

therefore, change the model slightly. We say that the employment agency has n

candidates, and they send us a list of the candidates in advance. On each day, we

choose, randomly, which candidate to interview. Although we know nothing about

5.1 The hiring problem 117

the candidates (besides their names), we have made a signiﬁcant change. Instead

of relying on a guess that the candidates come to us in a random order, we have

instead gained control of the process and enforced a random order.

More generally, we call an algorithm randomized if its behavior is determined

not only by its input but also by values produced by a random-number gener-

ator. We shall assume that we have at our disposal a random-number generator

RANDOM. A call to RANDOM.a; b/ returns an integer between a and b, inclu-

sive, with each such integer being equally likely. For example, RANDOM.0; 1/

produces 0 with probability 1=2, and it produces 1 with probability 1=2. A call to

RANDOM.3; 7/ returns either 3, 4, 5, 6, or 7, each with probability 1=5. Each inte-

ger returned by RANDOM is independent of the integers returned on previous calls.

You may imagine RANDOM as rolling a .b a C 1/-sided die to obtain its out-

put. (In practice, most programming environments offer a pseudorandom-number

generator: a deterministic algorithm returning numbers that “look” statistically

random.)

When analyzing the running time of a randomized algorithm, we take the expec-

tation of the running time over the distribution of values returned by the random

number generator. We distinguish these algorithms from those in which the input

is random by referring to the running time of a randomized algorithm as an ex-

pected running time. In general, we discuss the average-case running time when

the probability distribution is over the inputs to the algorithm, and we discuss the

expected running time when the algorithm itself makes random choices.

Exercises

5.1-1

Show that the assumption that we are always able to determine which candidate is

best, in line 4 of procedure HIRE-ASSISTANT, implies that we know a total order

on the ranks of the candidates.

5.1-2 ?

Describe an implementation of the procedure RANDOM.a; b/ that only makes calls

to RANDOM.0; 1/. What is the expected running time of your procedure, as a

function of a and b?

5.1-3 ?

Suppose that you want to output 0 with probability 1=2 and 1 with probability 1=2.

At your disposal is a procedure BIASED-RANDOM, that outputs either 0 or 1. It

outputs 1 with some probability p and 0 with probability 1 p, where 0 < p < 1,

but you do not know what p is. Give an algorithm that uses BIASED-RANDOM

as a subroutine, and returns an unbiased answer, returning 0 with probability 1=2

118 Chapter 5 Probabilistic Analysis and Randomized Algorithms

and 1 with probability 1=2. What is the expected running time of your algorithm

as a function of p?

5.2 Indicator random variables

In order to analyze many algorithms, including the hiring problem, we use indicator

random variables. Indicator random variables provide a convenient method for

converting between probabilities and expectations. Suppose we are given a sample

space S and an event A. Then the indicator random variable I fAg associated with

event A is deﬁned as

I fAg D

(

1 if A occurs ;

0 if A does not occur :

(5.1)

As a simple example, let us determine the expected number of heads that we

obtain when ﬂipping a fair coin. Our sample space is S D fH; T g, with Pr fHg D

Pr fT g D 1=2. We can then deﬁne an indicator random variable X H , associated

with the coin coming up heads, which is the event H. This variable counts the

number of heads obtained in this ﬂip, and it is 1 if the coin comes up heads and 0

otherwise. We write

X H D I fHg

D

(

1 if H occurs ;

0 if T occurs :

The expected number of heads obtained in one ﬂip of the coin is simply the ex-

pected value of our indicator variable X H :

E ŒX H D E ŒI fHg

D 1 Pr fHg C 0 Pr fT g

D 1 .1=2/ C 0 .1=2/

D 1=2 :

Thus the expected number of heads obtained by one ﬂip of a fair coin is 1=2. As

the following lemma shows, the expected value of an indicator random variable

associated with an event A is equal to the probability that A occurs.

Lemma 5.1

Given a sample space S and an event A in the sample space S, let X A D I fAg.

Then E ŒX A D Pr fAg.

5.2 Indicator random variables 119

Proof By the deﬁnition of an indicator random variable from equation (5.1) and

the deﬁnition of expected value, we have

E ŒX A D E ŒI fAg

D 1 Pr fAg C 0 Pr

˚

A

D Pr fAg ;

where A denotes S A, the complement of A.

Although indicator random variables may seem cumbersome for an application

such as counting the expected number of heads on a ﬂip of a single coin, they are

useful for analyzing situations in which we perform repeated random trials. For

example, indicator random variables give us a simple way to arrive at the result

of equation (C.37). In this equation, we compute the number of heads in n coin

ﬂips by considering separately the probability of obtaining 0 heads, 1 head, 2 heads,

etc. The simpler method proposed in equation (C.38) instead uses indicator random

variables implicitly. Making this argument more explicit, we let X i be the indicator

random variable associated with the event in which the ith ﬂip comes up heads:

X i D I fthe ith ﬂip results in the event Hg. Let X be the random variable denoting

the total number of heads in the n coin ﬂips, so that

X D

n X

iD1

X i :

We wish to compute the expected number of heads, and so we take the expectation

of both sides of the above equation to obtain

E ŒX D E

"

n X

iD1

X i

#

:

The above equation gives the expectation of the sum of n indicator random vari-

ables. By Lemma 5.1, we can easily compute the expectation of each of the random

variables. By equation (C.21)—linearity of expectation—it is easy to compute the

expectation of the sum: it equals the sum of the expectations of the n random

variables. Linearity of expectation makes the use of indicator random variables a

powerful analytical technique; it applies even when there is dependence among the

random variables. We now can easily compute the expected number of heads:

120 Chapter 5 Probabilistic Analysis and Randomized Algorithms

E ŒX D E

"

n X

iD1

X i

#

D

n X

iD1

E ŒX i

D

n X

iD1

1=2

D n=2 :

Thus, compared to the method used in equation (C.37), indicator random variables

greatly simplify the calculation. We shall use indicator random variables through-

out this book.

Analysis of the hiring problem using indicator random variables

Returning to the hiring problem, we now wish to compute the expected number of

times that we hire a new ofﬁce assistant. In order to use a probabilistic analysis, we

assume that the candidates arrive in a random order, as discussed in the previous

section. (We shall see in Section 5.3 how to remove this assumption.) Let X be the

random variable whose value equals the number of times we hire a new ofﬁce as-

sistant. We could then apply the deﬁnition of expected value from equation (C.20)

to obtain

E ŒX D

n X

xD1

x Pr fX D xg ;

but this calculation would be cumbersome. We shall instead use indicator random

variables to greatly simplify the calculation.

To use indicator random variables, instead of computing E ŒX by deﬁning one

variable associated with the number of times we hire a new ofﬁce assistant, we

deﬁne n variables related to whether or not each particular candidate is hired. In

particular, we let X i be the indicator random variable associated with the event in

which the ith candidate is hired. Thus,

X i D I fcandidate i is hiredg

D

(

1 if candidate i is hired ;

0 if candidate i is not hired ;

and

X D X 1 C X 2 C C X n : (5.2)

5.2 Indicator random variables 121

By Lemma 5.1, we have that

E ŒX i D Pr fcandidate i is hiredg ;

and we must therefore compute the probability that lines 5–6 of HIRE-ASSISTANT

are executed.

Candidate i is hired, in line 6, exactly when candidate i is better than each of

candidates 1 through i 1. Because we have assumed that the candidates arrive in

a random order, the ﬁrst i candidates have appeared in a random order. Any one of

these ﬁrst i candidates is equally likely to be the best-qualiﬁed so far. Candidate i

has a probability of 1=i of being better qualiﬁed than candidates 1 through i 1

and thus a probability of 1=i of being hired. By Lemma 5.1, we conclude that

E ŒX i D 1=i : (5.3)

Now we can compute E ŒX:

E ŒX D E

"

n X

iD1

X i

#

(by equation (5.2)) (5.4)

D

n X

iD1

E ŒX i (by linearity of expectation)

D

n X

iD1

1=i (by equation (5.3))

D ln n C O.1/ (by equation (A.7)) . (5.5)

Even though we interview n people, we actually hire only approximately ln n of

them, on average. We summarize this result in the following lemma.

Lemma 5.2

Assuming that the candidates are presented in a random order, algorithm HIRE-

ASSISTANT has an average-case total hiring cost of O.c h ln n/.

Proof The bound follows immediately from our deﬁnition of the hiring cost

and equation (5.5), which shows that the expected number of hires is approxi-

mately ln n.

The average-case hiring cost is a signiﬁcant improvement over the worst-case

hiring cost of O.c h n/.

122 Chapter 5 Probabilistic Analysis and Randomized Algorithms

Exercises

5.2-1

In HIRE-ASSISTANT, assuming that the candidates are presented in a random or-

der, what is the probability that you hire exactly one time? What is the probability

that you hire exactly n times?

5.2-2

In HIRE-ASSISTANT, assuming that the candidates are presented in a random or-

der, what is the probability that you hire exactly twice?

5.2-3

Use indicator random variables to compute the expected value of the sum of n dice.

5.2-4

Use indicator random variables to solve the following problem, which is known as

the hat-check problem. Each of n customers gives a hat to a hat-check person at a

restaurant. The hat-check person gives the hats back to the customers in a random

order. What is the expected number of customers who get back their own hat?

5.2-5

Let AŒ1 : : n be an array of n distinct numbers. If i < j and AŒi > AŒj , then

the pair .i; j / is called an inversion of A. (See Problem 2-4 for more on inver-

sions.) Suppose that the elements of A form a uniform random permutation of

h1; 2; : : : ; ni. Use indicator random variables to compute the expected number of

inversions.

5.3 Randomized algorithms

In the previous section, we showed how knowing a distribution on the inputs can

help us to analyze the average-case behavior of an algorithm. Many times, we do

not have such knowledge, thus precluding an average-case analysis. As mentioned

in Section 5.1, we may be able to use a randomized algorithm.

For a problem such as the hiring problem, in which it is helpful to assume that

all permutations of the input are equally likely, a probabilistic analysis can guide

the development of a randomized algorithm. Instead of assuming a distribution

of inputs, we impose a distribution. In particular, before running the algorithm,

we randomly permute the candidates in order to enforce the property that every

permutation is equally likely. Although we have modiﬁed the algorithm, we still

expect to hire a new ofﬁce assistant approximately ln n times. But now we expect

5.3 Randomized algorithms 123

this to be the case for any input, rather than for inputs drawn from a particular

distribution.

Let us further explore the distinction between probabilistic analysis and random-

ized algorithms. In Section 5.2, we claimed that, assuming that the candidates ar-

rive in a random order, the expected number of times we hire a new ofﬁce assistant

is about ln n. Note that the algorithm here is deterministic; for any particular input,

the number of times a new ofﬁce assistant is hired is always the same. Furthermore,

the number of times we hire a new ofﬁce assistant differs for different inputs, and it

depends on the ranks of the various candidates. Since this number depends only on

the ranks of the candidates, we can represent a particular input by listing, in order,

the ranks of the candidates, i.e., hrank.1/; rank.2/; : : : ; rank.n/i. Given the rank

list A 1 D h1;2;3;4;5;6;7;8;9;10i, a new ofﬁce assistant is always hired 10 times,

since each successive candidate is better than the previous one, and lines 5–6 are

executed in each iteration. Given the list of ranks A 2 D h10; 9; 8; 7; 6; 5; 4; 3; 2; 1i,

a new ofﬁce assistant is hired only once, in the ﬁrst iteration. Given a list of ranks

A 3 D h5; 2; 1; 8; 4; 7; 10; 9; 3; 6i, a new ofﬁce assistant is hired three times,

upon interviewing the candidates with ranks 5, 8, and 10. Recalling that the cost

of our algorithm depends on how many times we hire a new ofﬁce assistant, we

see that there are expensive inputs such as A 1 , inexpensive inputs such as A 2 , and

moderately expensive inputs such as A 3 .

Consider, on the other hand, the randomized algorithm that ﬁrst permutes the

candidates and then determines the best candidate. In this case, we randomize in

the algorithm, not in the input distribution. Given a particular input, say A 3 above,

we cannot say how many times the maximum is updated, because this quantity

differs with each run of the algorithm. The ﬁrst time we run the algorithm on A 3 ,

it may produce the permutation A 1 and perform 10 updates; but the second time

we run the algorithm, we may produce the permutation A 2 and perform only one

update. The third time we run it, we may perform some other number of updates.

Each time we run the algorithm, the execution depends on the random choices

made and is likely to differ from the previous execution of the algorithm. For this

algorithm and many other randomized algorithms, no particular input elicits its

worst-case behavior. Even your worst enemy cannot produce a bad input array,

since the random permutation makes the input order irrelevant. The randomized

algorithm performs badly only if the random-number generator produces an “un-

lucky” permutation.

For the hiring problem, the only change needed in the code is to randomly per-

mute the array.

124 Chapter 5 Probabilistic Analysis and Randomized Algorithms

RANDOMIZED-HIRE-ASSISTANT.n/

1 randomly permute the list of candidates

2 best D 0 // candidate 0 is a least-qualiﬁed dummy candidate

3 for i D 1 to n

4 interview candidate i

5 if candidate i is better than candidate best

6 best D i

7 hire candidate i

With this simple change, we have created a randomized algorithm whose perfor-

mance matches that obtained by assuming that the candidates were presented in a

random order.

Lemma 5.3

The expected hiring cost of the procedure RANDOMIZED-HIRE-ASSISTANT is

O.c h ln n/.

Proof After permuting the input array, we have achieved a situation identical to

that of the probabilistic analysis of HIRE-ASSISTANT.

Comparing Lemmas 5.2 and 5.3 highlights the difference between probabilistic

analysis and randomized algorithms. In Lemma 5.2, we make an assumption about

the input. In Lemma 5.3, we make no such assumption, although randomizing the

input takes some additional time. To remain consistent with our terminology, we

couched Lemma 5.2 in terms of the average-case hiring cost and Lemma 5.3 in

terms of the expected hiring cost. In the remainder of this section, we discuss some

issues involved in randomly permuting inputs.

Randomly permuting arrays

Many randomized algorithms randomize the input by permuting the given input

array. (There are other ways to use randomization.) Here, we shall discuss two

methods for doing so. We assume that we are given an array A which, without loss

of generality, contains the elements 1 through n. Our goal is to produce a random

permutation of the array.

One common method is to assign each element AŒi of the array a random pri-

ority P Œi, and then sort the elements of A according to these priorities. For ex-

ample, if our initial array is A D h1; 2; 3; 4i and we choose random priorities

P D h36; 3; 62; 19i, we would produce an array B D h2; 4; 1; 3i, since the second

priority is the smallest, followed by the fourth, then the ﬁrst, and ﬁnally the third.

We call this procedure PERMUTE-BY-SORTING:

5.3 Randomized algorithms 125

PERMUTE-BY-SORTING.A/

1 n D A:length

2 let P Œ1 : : n be a new array

3 for i D 1 to n

4 P Œi D RANDOM.1; n 3 /

5 sort A, using P as sort keys

Line 4 chooses a random number between 1 and n 3

. We use a range of 1 to n 3

to make it likely that all the priorities in P are unique. (Exercise 5.3-5 asks you

to prove that the probability that all entries are unique is at least 1 1=n, and

Exercise 5.3-6 asks how to implement the algorithm even if two or more priorities

are identical.) Let us assume that all the priorities are unique.

The time-consuming step in this procedure is the sorting in line 5. As we shall

see in Chapter 8, if we use a comparison sort, sorting takes .n lg n/ time. We

can achieve this lower bound, since we have seen that merge sort takes ‚.n lg n/

time. (We shall see other comparison sorts that take ‚.n lg n/ time in Part II.

Exercise 8.3-4 asks you to solve the very similar problem of sorting numbers in the

range 0 to n 3 1 in O.n/ time.) After sorting, if P Œi is the j th smallest priority,

then AŒi lies in position j of the output. In this manner we obtain a permutation. It

remains to prove that the procedure produces a uniform random permutation, that

is, that the procedure is equally likely to produce every permutation of the numbers

1 through n.

Lemma 5.4

Procedure PERMUTE-BY-SORTING produces a uniform random permutation of the

input, assuming that all priorities are distinct.

Proof We start by considering the particular permutation in which each ele-

ment AŒi receives the ith smallest priority. We shall show that this permutation

occurs with probability exactly 1=nŠ. For i D 1; 2; : : : ; n, let E i be the event

that element AŒi receives the ith smallest priority. Then we wish to compute the

probability that for all i, event E i occurs, which is

Pr fE 1 \ E 2 \ E 3 \ \ E n1 \ E n g :

Using Exercise C.2-5, this probability is equal to

Pr fE 1 g Pr fE 2 j E 1 g Pr fE 3 j E 2 \ E 1 g Pr fE 4 j E 3 \ E 2 \ E 1 g

Pr fE i j E i1 \ E i2 \ \ E 1 g Pr fE n j E n1 \ \ E 1 g :

We have that Pr fE 1 g D 1=n because it is the probability that one priority

chosen randomly out of a set of n is the smallest priority. Next, we observe

126 Chapter 5 Probabilistic Analysis and Randomized Algorithms

that Pr fE 2 j E 1 g D 1=.n 1/ because given that element AŒ1 has the small-

est priority, each of the remaining n 1 elements has an equal chance of hav-

ing the second smallest priority. In general, for i D 2; 3; : : : ; n, we have that

Pr fE i j E i1 \ E i2 \ \ E 1 g D 1=.n i C 1/, since, given that elements AŒ1

through AŒi 1 have the i 1 smallest priorities (in order), each of the remaining

n .i 1/ elements has an equal chance of having the ith smallest priority. Thus,

we have

Pr fE 1 \ E 2 \ E 3 \ \ E n1 \ E n g D

1

n

1

n 1

1

2

1

1

D

1

nŠ

;

and we have shown that the probability of obtaining the identity permutation

is 1=nŠ.

We can extend this proof to work for any permutation of priorities. Consider

any ﬁxed permutation D h .1/; .2/; : : : ; .n/i of the set f1; 2; : : : ; ng. Let us

denote by r i the rank of the priority assigned to element AŒi, where the element

with the j th smallest priority has rank j . If we deﬁne E i as the event in which

element AŒi receives the .i/th smallest priority, or r i D .i/, the same proof

still applies. Therefore, if we calculate the probability of obtaining any particular

permutation, the calculation is identical to the one above, so that the probability of

obtaining this permutation is also 1=nŠ.

You might think that to prove that a permutation is a uniform random permuta-

tion, it sufﬁces to show that, for each element AŒi, the probability that the element

winds up in position j is 1=n. Exercise 5.3-4 shows that this weaker condition is,

in fact, insufﬁcient.

A better method for generating a random permutation is to permute the given

array in place. The procedure RANDOMIZE-IN-PLACE does so in O.n/ time. In

its ith iteration, it chooses the element AŒi randomly from among elements AŒi

through AŒn. Subsequent to the ith iteration, AŒi is never altered.

RANDOMIZE-IN-PLACE.A/

1 n D A:length

2 for i D 1 to n

3 swap AŒi with AŒRANDOM.i; n/

We shall use a loop invariant to show that procedure RANDOMIZE-IN-PLACE

produces a uniform random permutation. A k-permutation on a set of n ele-

ments is a sequence containing k of the n elements, with no repetitions. (See

Appendix C.) There are nŠ=.n k/Š such possible k-permutations.

5.3 Randomized algorithms 127

Lemma 5.5

Procedure RANDOMIZE-IN-PLACE computes a uniform random permutation.

Proof We use the following loop invariant:

Just prior to the ith iteration of the for loop of lines 2–3, for each possible

.i 1/-permutation of the n elements, the subarray AŒ1 : : i 1 contains

this .i 1/-permutation with probability .n i C 1/Š=nŠ.

We need to show that this invariant is true prior to the ﬁrst loop iteration, that each

iteration of the loop maintains the invariant, and that the invariant provides a useful

property to show correctness when the loop terminates.

Initialization: Consider the situation just before the ﬁrst loop iteration, so that

i D 1. The loop invariant says that for each possible 0-permutation, the sub-

array AŒ1 : : 0 contains this 0-permutation with probability .n i C 1/Š=nŠ D

nŠ=nŠ D 1. The subarray AŒ1 : : 0 is an empty subarray, and a 0-permutation

has no elements. Thus, AŒ1 : : 0 contains any 0-permutation with probability 1,

and the loop invariant holds prior to the ﬁrst iteration.

Maintenance: We assume that just before the ith iteration, each possible

.i 1/-permutation appears in the subarray AŒ1 : : i 1 with probability

.n i C 1/Š=nŠ, and we shall show that after the ith iteration, each possible

i-permutation appears in the subarray AŒ1 : : i with probability .n i/Š=nŠ.

Incrementing i for the next iteration then maintains the loop invariant.

Let us examine the ith iteration. Consider a particular i-permutation, and de-

note the elements in it by hx 1 ; x 2 ; : : : ; x i i. This permutation consists of an

.i 1/-permutation hx 1 ; : : : ; x i1 i followed by the value x i that the algorithm

places in AŒi. Let E 1 denote the event in which the ﬁrst i 1 iterations have

created the particular .i 1/-permutation hx 1 ;: : : ;x i1 i in AŒ1 : : i 1. By the

loop invariant, Pr fE 1 g D .n i C 1/Š=nŠ. Let E 2 be the event that ith iteration

puts x i in position AŒi. The i-permutation hx 1 ;: : : ;x i i appears in AŒ1 : : i pre-

cisely when both E 1 and E 2 occur, and so we wish to compute Pr fE 2 \ E 1 g.

Using equation (C.14), we have

Pr fE 2 \ E 1 g D Pr fE 2 j E 1 g Pr fE 1 g :

The probability Pr fE 2 j E 1 g equals 1=.niC1/ because in line 3 the algorithm

chooses x i randomly from the n i C 1 values in positions AŒi : : n. Thus, we

have

128 Chapter 5 Probabilistic Analysis and Randomized Algorithms

Pr fE 2 \ E 1 g D Pr fE 2 j E 1 g Pr fE 1 g

D

1

n i C 1

.n i C 1/Š

nŠ

D

.n i/Š

nŠ

:

Termination: At termination, i D n C 1, and we have that the subarray AŒ1 : : n

is a given n-permutation with probability .n.nC1/C1/=nŠ D 0Š=nŠ D 1=nŠ.

Thus, RANDOMIZE-IN-PLACE produces a uniform random permutation.

A randomized algorithm is often the simplest and most efﬁcient way to solve a

problem. We shall use randomized algorithms occasionally throughout this book.

Exercises

5.3-1

Professor Marceau objects to the loop invariant used in the proof of Lemma 5.5. He

questions whether it is true prior to the ﬁrst iteration. He reasons that we could just

as easily declare that an empty subarray contains no 0-permutations. Therefore,

the probability that an empty subarray contains a 0-permutation should be 0, thus

invalidating the loop invariant prior to the ﬁrst iteration. Rewrite the procedure

RANDOMIZE-IN-PLACE so that its associated loop invariant applies to a nonempty

subarray prior to the ﬁrst iteration, and modify the proof of Lemma 5.5 for your

procedure.

5.3-2

Professor Kelp decides to write a procedure that produces at random any permuta-

tion besides the identity permutation. He proposes the following procedure:

PERMUTE-WITHOUT-IDENTITY.A/

1 n D A:length

2 for i D 1 to n 1

3 swap AŒi with AŒRANDOM.i C 1; n/

Does this code do what Professor Kelp intends?

5.3-3

Suppose that instead of swapping element AŒi with a random element from the

subarray AŒi : : n, we swapped it with a random element from anywhere in the

array:

5.3 Randomized algorithms 129

PERMUTE-WITH-ALL.A/

1 n D A:length

2 for i D 1 to n

3 swap AŒi with AŒRANDOM.1; n/

Does this code produce a uniform random permutation? Why or why not?

5.3-4

Professor Armstrong suggests the following procedure for generating a uniform

random permutation:

PERMUTE-BY-CYCLIC.A/

1 n D A:length

2 let BŒ1 : : n be a new array

3 offset D RANDOM.1; n/

4 for i D 1 to n

5 dest D i C offset

6 if dest > n

7 dest D dest n

8 BŒdest D AŒi

9 return B

Show that each element AŒi has a 1=n probability of winding up in any particular

position in B. Then show that Professor Armstrong is mistaken by showing that

the resulting permutation is not uniformly random.

5.3-5 ?

Prove that in the array P in procedure PERMUTE-BY-SORTING, the probability

that all elements are unique is at least 1 1=n.

5.3-6

Explain how to implement the algorithm PERMUTE-BY-SORTING to handle the

case in which two or more priorities are identical. That is, your algorithm should

produce a uniform random permutation, even if two or more priorities are identical.

5.3-7

Suppose we want to create a random sample of the set f1; 2; 3; : : : ; ng, that is,

an m-element subset S, where 0 m n, such that each m-subset is equally

likely to be created. One way would be to set AŒi D i for i D 1; 2; 3; : : : ; n,

call RANDOMIZE-IN-PLACE.A/, and then take just the ﬁrst m array elements.

This method would make n calls to the RANDOM procedure. If n is much larger

than m, we can create a random sample with fewer calls to RANDOM. Show that

130 Chapter 5 Probabilistic Analysis and Randomized Algorithms

the following recursive procedure returns a random m-subset S of f1; 2; 3; : : : ; ng,

in which each m-subset is equally likely, while making only m calls to RANDOM:

RANDOM-SAMPLE.m; n/

1 if m == 0

2 return ;

3 else S D RANDOM-SAMPLE.m 1; n 1/

4 i D RANDOM.1; n/

5 if i 2 S

6 S D S [ fng

7 else S D S [ fig

8 return S

? 5.4 Probabilistic analysis and further uses of indicator random variables

This advanced section further illustrates probabilistic analysis by way of four ex-

amples. The ﬁrst determines the probability that in a room of k people, two of

them share the same birthday. The second example examines what happens when

we randomly toss balls into bins. The third investigates “streaks” of consecutive

heads when we ﬂip coins. The ﬁnal example analyzes a variant of the hiring prob-

lem in which you have to make decisions without actually interviewing all the

candidates.

5.4.1 The birthday paradox

Our ﬁrst example is the birthday paradox. How many people must there be in a

room before there is a 50% chance that two of them were born on the same day of

the year? The answer is surprisingly few. The paradox is that it is in fact far fewer

than the number of days in a year, or even half the number of days in a year, as we

shall see.

To answer this question, we index the people in the room with the integers

1; 2; : : : ; k, where k is the number of people in the room. We ignore the issue

of leap years and assume that all years have n D 365 days. For i D 1; 2; : : : ; k,

let b i be the day of the year on which person i’s birthday falls, where 1 b i n.

We also assume that birthdays are uniformly distributed across the n days of the

year, so that Pr fb i D rg D 1=n for i D 1; 2; : : : ; k and r D 1; 2; : : : ; n.

The probability that two given people, say i and j , have matching birthdays

depends on whether the random selection of birthdays is independent. We assume

from now on that birthdays are independent, so that the probability that i’s birthday

5.4 Probabilistic analysis and further uses of indicator random variables 131

and j ’s birthday both fall on day r is

Pr fb i D r and b j D rg D Pr fb i D rg Pr fb j D rg

D 1=n

2

:

Thus, the probability that they both fall on the same day is

Pr fb i D b j g D

n X

rD1

Pr fb i D r and b j D rg

D

n X

rD1

.1=n

2

/

D 1=n : (5.6)

More intuitively, once b i is chosen, the probability that b j is chosen to be the same

day is 1=n. Thus, the probability that i and j have the same birthday is the same

as the probability that the birthday of one of them falls on a given day. Notice,

however, that this coincidence depends on the assumption that the birthdays are

independent.

We can analyze the probability of at least 2 out of k people having matching

birthdays by looking at the complementary event. The probability that at least two

of the birthdays match is 1 minus the probability that all the birthdays are different.

The event that k people have distinct birthdays is

B k D

k \

iD1

A i ;

where A i is the event that person i’s birthday is different from person j ’s for

all j < i. Since we can write B k D A k \ B k1 , we obtain from equation (C.16)

the recurrence

Pr fB k g D Pr fB k1 g Pr fA k j B k1 g ; (5.7)

where we take Pr fB 1 g D Pr fA 1 g D 1 as an initial condition. In other words,

the probability that b 1 ; b 2 ; : : : ; b k are distinct birthdays is the probability that

b 1 ; b 2 ; : : : ; b k1 are distinct birthdays times the probability that b k ¤ b i for

i D 1; 2; : : : ; k 1, given that b 1 ; b 2 ; : : : ; b k1 are distinct.

If b 1 ; b 2 ; : : : ; b k1 are distinct, the conditional probability that b k ¤ b i for

i D 1; 2; : : : ; k 1 is Pr fA k j B k1 g D .n k C 1/=n, since out of the n days,

n .k 1/ days are not taken. We iteratively apply the recurrence (5.7) to obtain

132 Chapter 5 Probabilistic Analysis and Randomized Algorithms

Pr fB k g D Pr fB k1 g Pr fA k j B k1 g

D Pr fB k2 g Pr fA k1 j B k2 g Pr fA k j B k1 g

:

:

:

D Pr fB 1 g Pr fA 2 j B 1 g Pr fA 3 j B 2 g Pr fA k j B k1 g

D 1

n 1

n

n 2

n

n k C 1

n

D 1

1

1

n

1

2

n

1

k 1

n

:

Inequality (3.12), 1 C x e x

, gives us

Pr fB k g e

1=n

e

2=n

e

.k1/=n

D e

P k1

iD1 i=n

D e

k.k1/=2n

1=2

when k.k 1/=2n ln.1=2/. The probability that all k birthdays are distinct

is at most 1=2 when k.k 1/ 2n ln 2 or, solving the quadratic equation, when

k .1 C

p

1 C .8 ln 2/n/=2. For n D 365, we must have k 23. Thus, if at

least 23 people are in a room, the probability is at least 1=2 that at least two people

have the same birthday. On Mars, a year is 669 Martian days long; it therefore

takes 31 Martians to get the same effect.

An analysis using indicator random variables

We can use indicator random variables to provide a simpler but approximate anal-

ysis of the birthday paradox. For each pair .i; j / of the k people in the room, we

deﬁne the indicator random variable X ij , for 1 i < j k, by

X ij D I fperson i and person j have the same birthdayg

D

(

1 if person i and person j have the same birthday ;

0 otherwise :

By equation (5.6), the probability that two people have matching birthdays is 1=n,

and thus by Lemma 5.1, we have

E ŒX ij D Pr fperson i and person j have the same birthdayg

D 1=n :

Letting X be the random variable that counts the number of pairs of individuals

having the same birthday, we have

5.4 Probabilistic analysis and further uses of indicator random variables 133

X D

k X

iD1

k X

j DiC1

X ij :

Taking expectations of both sides and applying linearity of expectation, we obtain

E ŒX D E

"

k X

iD1

k X

j DiC1

X ij

#

D

k X

iD1

k X

j DiC1

E ŒX ij

D

k

2

!

1

n

D

k.k 1/

2n

:

When k.k 1/ 2n, therefore, the expected number of pairs of people with the

same birthday is at least 1. Thus, if we have at least

p

2nC1 individuals in a room,

we can expect at least two to have the same birthday. For n D 365, if k D 28, the

expected number of pairs with the same birthday is .28 27/=.2 365/ 1:0356.

Thus, with at least 28 people, we expect to ﬁnd at least one matching pair of birth-

days. On Mars, where a year is 669 Martian days long, we need at least 38 Mar-

tians.

The ﬁrst analysis, which used only probabilities, determined the number of peo-

ple required for the probability to exceed 1=2 that a matching pair of birthdays

exists, and the second analysis, which used indicator random variables, determined

the number such that the expected number of matching birthdays is 1. Although

the exact numbers of people differ for the two situations, they are the same asymp-

totically: ‚.

p

n/.

5.4.2 Balls and bins

Consider a process in which we randomly toss identical balls into b bins, numbered

1; 2; : : : ; b. The tosses are independent, and on each toss the ball is equally likely

to end up in any bin. The probability that a tossed ball lands in any given bin is 1=b.

Thus, the ball-tossing process is a sequence of Bernoulli trials (see Appendix C.4)

with a probability 1=b of success, where success means that the ball falls in the

given bin. This model is particularly useful for analyzing hashing (see Chapter 11),

and we can answer a variety of interesting questions about the ball-tossing process.

(Problem C-1 asks additional questions about balls and bins.)

134 Chapter 5 Probabilistic Analysis and Randomized Algorithms

How many balls fall in a given bin? The number of balls that fall in a given bin

follows the binomial distribution b.kI n; 1=b/. If we toss n balls, equation (C.37)

tells us that the expected number of balls that fall in the given bin is n=b.

How many balls must we toss, on the average, until a given bin contains a ball?

The number of tosses until the given bin receives a ball follows the geometric

distribution with probability 1=b and, by equation (C.32), the expected number of

tosses until success is 1=.1=b/ D b.

How many balls must we toss until every bin contains at least one ball? Let us

call a toss in which a ball falls into an empty bin a “hit.” We want to know the

expected number n of tosses required to get b hits.

Using the hits, we can partition the n tosses into stages. The ith stage consists of

the tosses after the .i 1/st hit until the ith hit. The ﬁrst stage consists of the ﬁrst

toss, since we are guaranteed to have a hit when all bins are empty. For each toss

during the ith stage, i 1 bins contain balls and b i C 1 bins are empty. Thus,

for each toss in the ith stage, the probability of obtaining a hit is .b i C 1/=b.

Let n i denote the number of tosses in the ith stage. Thus, the number of tosses

required to get b hits is n D

P b

iD1

n i . Each random variable n i has a geometric

distribution with probability of success .b i C 1/=b and thus, by equation (C.32),

we have

E Œn i D

b

b i C 1

:

By linearity of expectation, we have

E Œn D E

"

b X

iD1

n i

#

D

b X

iD1

E Œn i

D

b X

iD1

b

b i C 1

D b

b X

iD1

1

i

D b.ln b C O.1// (by equation (A.7)) .

It therefore takes approximately b ln b tosses before we can expect that every bin

has a ball. This problem is also known as the coupon collector’s problem, which

says that a person trying to collect each of b different coupons expects to acquire

approximately b ln b randomly obtained coupons in order to succeed.

5.4 Probabilistic analysis and further uses of indicator random variables 135

5.4.3 Streaks

Suppose you ﬂip a fair coin n times. What is the longest streak of consecutive

heads that you expect to see? The answer is ‚.lg n/, as the following analysis

shows.

We ﬁrst prove that the expected length of the longest streak of heads is O.lg n/.

The probability that each coin ﬂip is a head is 1=2. Let A ik be the event that a

streak of heads of length at least k begins with the ith coin ﬂip or, more precisely,

the event that the k consecutive coin ﬂips i; i C 1; : : : ; i C k 1 yield only heads,

where 1 k n and 1 i nkC1. Since coin ﬂips are mutually independent,

for any given event A ik , the probability that all k ﬂips are heads is

Pr fA ik g D 1=2

k

: (5.8)

For k D 2 dlg ne,

Pr fA i;2dlg ne g D 1=2

2dlg ne

1=2

2 lg n

D 1=n

2

;

and thus the probability that a streak of heads of length at least 2 dlg ne begins in

position i is quite small. There are at most n 2 dlg ne C 1 positions where such

a streak can begin. The probability that a streak of heads of length at least 2 dlg ne

begins anywhere is therefore

Pr

(

n2dlg neC1 [

iD1

A i;2dlg ne

)

n2dlg neC1 X

iD1

1=n

2

<

n X

iD1

1=n

2

D 1=n ; (5.9)

since by Boole’s inequality (C.19), the probability of a union of events is at most

the sum of the probabilities of the individual events. (Note that Boole’s inequality

holds even for events such as these that are not independent.)

We now use inequality (5.9) to bound the length of the longest streak. For

j D 0; 1; 2; : : : ; n, let L j be the event that the longest streak of heads has length ex-

actly j , and let L be the length of the longest streak. By the deﬁnition of expected

value, we have

E ŒL D

n X

j D0

j Pr fL j g : (5.10)

136 Chapter 5 Probabilistic Analysis and Randomized Algorithms

We could try to evaluate this sum using upper bounds on each Pr fL j g similar to

those computed in inequality (5.9). Unfortunately, this method would yield weak

bounds. We can use some intuition gained by the above analysis to obtain a good

bound, however. Informally, we observe that for no individual term in the sum-

mation in equation (5.10) are both the factors j and Pr fL j g large. Why? When

j 2 dlg ne, then Pr fL j g is very small, and when j < 2 dlg ne, then j is fairly

small. More formally, we note that the events L j for j D 0; 1; : : : ; n are disjoint,

and so the probability that a streak of heads of length at least 2 dlg ne begins any-

where is

P n

j D2dlg ne

Pr fL j g. By inequality (5.9), we have

P n

j D2dlg ne

Pr fL j g < 1=n.

Also, noting that

P n

j D0

Pr fL j g D 1, we have that

P 2dlg ne1

j D0

Pr fL j g 1. Thus,

we obtain

E ŒL D

n X

j D0

j Pr fL j g

D

2dlg ne1 X

j D0

j Pr fL j g C

n X

j D2dlg ne

j Pr fL j g

<

2dlg ne1 X

j D0

.2 dlg ne/ Pr fL j g C

n X

j D2dlg ne

n Pr fL j g

D 2 dlg ne

2dlg ne1 X

j D0

Pr fL j g C n

n X

j D2dlg ne

Pr fL j g

< 2 dlg ne 1 C n .1=n/

D O.lg n/ :

The probability that a streak of heads exceeds r dlg ne ﬂips diminishes quickly

with r. For r 1, the probability that a streak of at least r dlg ne heads starts in

position i is

Pr fA i;rdlg ne g D 1=2

rdlg ne

1=n

r

:

Thus, the probability is at most n=n r D 1=n r1

that the longest streak is at

least r dlg ne, or equivalently, the probability is at least 1 1=n r1

that the longest

streak has length less than r dlg ne.

As an example, for n D 1000 coin ﬂips, the probability of having a streak of at

least 2 dlg ne D 20 heads is at most 1=n D 1=1000. The chance of having a streak

longer than 3 dlg ne D 30 heads is at most 1=n 2 D 1=1,000,000.

We now prove a complementary lower bound: the expected length of the longest

streak of heads in n coin ﬂips is .lg n/. To prove this bound, we look for streaks

5.4 Probabilistic analysis and further uses of indicator random variables 137

of length s by partitioning the n ﬂips into approximately n=s groups of s ﬂips

each. If we choose s D b.lg n/=2c, we can show that it is likely that at least one

of these groups comes up all heads, and hence it is likely that the longest streak

has length at least s D .lg n/. We then show that the longest streak has expected

length .lg n/.

We partition the n coin ﬂips into at least bn= b.lg n/=2cc groups of b.lg n/=2c

consecutive ﬂips, and we bound the probability that no group comes up all heads.

By equation (5.8), the probability that the group starting in position i comes up all

heads is

Pr fA i;b.lg n/=2c g D 1=2

b.lg n/=2c

1=

p

n :

The probability that a streak of heads of length at least b.lg n/=2c does not begin

in position i is therefore at most 1 1=

p

n. Since the bn= b.lg n/=2cc groups are

formed from mutually exclusive, independent coin ﬂips, the probability that every

one of these groups fails to be a streak of length b.lg n/=2c is at most

1 1=

p

n

bn=b.lg n/=2cc

1 1=

p

n

n=b.lg n/=2c1

1 1=

p

n

2n= lg n1

e

.2n= lg n1/=

p

n

D O.e

lg n

/

D O.1=n/ :

For this argument, we used inequality (3.12), 1 C x e x

, and the fact, which you

might want to verify, that .2n= lg n 1/=

p

n lg n for sufﬁciently large n.

Thus, the probability that the longest streak exceeds b.lg n/=2c is

n X

j Db.lg n/=2cC1

Pr fL j g 1 O.1=n/ : (5.11)

We can now calculate a lower bound on the expected length of the longest streak,

beginning with equation (5.10) and proceeding in a manner similar to our analysis

of the upper bound:

138 Chapter 5 Probabilistic Analysis and Randomized Algorithms

E ŒL D

n X

j D0

j Pr fL j g

D

b.lg n/=2c X

j D0

j Pr fL j g C

n X

j Db.lg n/=2cC1

j Pr fL j g

b.lg n/=2c X

j D0

0 Pr fL j g C

n X

j Db.lg n/=2cC1

b.lg n/=2c Pr fL j g

D 0

b.lg n/=2c X

j D0

Pr fL j g C b.lg n/=2c

n X

j Db.lg n/=2cC1

Pr fL j g

0 C b.lg n/=2c .1 O.1=n// (by inequality (5.11))

D .lg n/ :

As with the birthday paradox, we can obtain a simpler but approximate analysis

using indicator random variables. We let X ik D I fA ik g be the indicator random

variable associated with a streak of heads of length at least k beginning with the

ith coin ﬂip. To count the total number of such streaks, we deﬁne

X D

nkC1 X

iD1

X ik :

Taking expectations and using linearity of expectation, we have

E ŒX D E

"

nkC1 X

iD1

X ik

#

D

nkC1 X

iD1

E ŒX ik

D

nkC1 X

iD1

Pr fA ik g

D

nkC1 X

iD1

1=2

k

D

n k C 1

2 k

:

By plugging in various values for k, we can calculate the expected number of

streaks of length k. If this number is large (much greater than 1), then we expect

many streaks of length k to occur and the probability that one occurs is high. If

5.4 Probabilistic analysis and further uses of indicator random variables 139

this number is small (much less than 1), then we expect few streaks of length k to

occur and the probability that one occurs is low. If k D c lg n, for some positive

constant c, we obtain

E ŒX D

n c lg n C 1

2 c lg n

D

n c lg n C 1

n c

D

1

n c1

.c lg n 1/=n

n c1

D ‚.1=n

c1

/ :

If c is large, the expected number of streaks of length c lg n is small, and we con-

clude that they are unlikely to occur. On the other hand, if c D 1=2, then we obtain

E ŒX D ‚.1=n 1=21 / D ‚.n 1=2 /, and we expect that there are a large number

of streaks of length .1=2/ lg n. Therefore, one streak of such a length is likely to

occur. From these rough estimates alone, we can conclude that the expected length

of the longest streak is ‚.lg n/.

5.4.4 The on-line hiring problem

As a ﬁnal example, we consider a variant of the hiring problem. Suppose now that

we do not wish to interview all the candidates in order to ﬁnd the best one. We

also do not wish to hire and ﬁre as we ﬁnd better and better applicants. Instead, we

are willing to settle for a candidate who is close to the best, in exchange for hiring

exactly once. We must obey one company requirement: after each interview we

must either immediately offer the position to the applicant or immediately reject the

applicant. What is the trade-off between minimizing the amount of interviewing

and maximizing the quality of the candidate hired?

We can model this problem in the following way. After meeting an applicant,

we are able to give each one a score; let score.i/ denote the score we give to the ith

applicant, and assume that no two applicants receive the same score. After we have

seen j applicants, we know which of the j has the highest score, but we do not

know whether any of the remaining nj applicants will receive a higher score. We

decide to adopt the strategy of selecting a positive integer k < n, interviewing and

then rejecting the ﬁrst k applicants, and hiring the ﬁrst applicant thereafter who has

a higher score than all preceding applicants. If it turns out that the best-qualiﬁed

applicant was among the ﬁrst k interviewed, then we hire the nth applicant. We

formalize this strategy in the procedure ON-LINE-MAXIMUM.k; n/, which returns

the index of the candidate we wish to hire.

140 Chapter 5 Probabilistic Analysis and Randomized Algorithms

ON-LINE-MAXIMUM.k; n/

1 bestscore D 1

2 for i D 1 to k

3 if score.i/ > bestscore

4 bestscore D score.i/

5 for i D k C 1 to n

6 if score.i/ > bestscore

7 return i

8 return n

We wish to determine, for each possible value of k, the probability that we

hire the most qualiﬁed applicant. We then choose the best possible k, and

implement the strategy with that value. For the moment, assume that k is

ﬁxed. Let M.j / D max 1ij fscore.i/g denote the maximum score among ap-

plicants 1 through j . Let S be the event that we succeed in choosing the best-

qualiﬁed applicant, and let S i be the event that we succeed when the best-qualiﬁed

applicant is the ith one interviewed. Since the various S i are disjoint, we have

that Pr fSg D

P n

iD1

Pr fS i g. Noting that we never succeed when the best-qualiﬁed

applicant is one of the ﬁrst k, we have that Pr fS i g D 0 for i D 1; 2; : : : ; k. Thus,

we obtain

Pr fSg D

n X

iDkC1

Pr fS i g : (5.12)

We now compute Pr fS i g. In order to succeed when the best-qualiﬁed applicant

is the ith one, two things must happen. First, the best-qualiﬁed applicant must be

in position i, an event which we denote by B i . Second, the algorithm must not

select any of the applicants in positions k C1 through i 1, which happens only if,

for each j such that k C1 j i 1, we ﬁnd that score.j / < bestscore in line 6.

(Because scores are unique, we can ignore the possibility of score.j / D bestscore.)

In other words, all of the values score.k C 1/ through score.i 1/ must be less

than M.k/; if any are greater than M.k/, we instead return the index of the ﬁrst

one that is greater. We use O i to denote the event that none of the applicants in

position k C 1 through i 1 are chosen. Fortunately, the two events B i and O i

are independent. The event O i depends only on the relative ordering of the values

in positions 1 through i 1, whereas B i depends only on whether the value in

position i is greater than the values in all other positions. The ordering of the

values in positions 1 through i 1 does not affect whether the value in position i

is greater than all of them, and the value in position i does not affect the ordering

of the values in positions 1 through i 1. Thus we can apply equation (C.15) to

obtain

5.4 Probabilistic analysis and further uses of indicator random variables 141

Pr fS i g D Pr fB i \ O i g D Pr fB i g Pr fO i g :

The probability Pr fB i g is clearly 1=n, since the maximum is equally likely to

be in any one of the n positions. For event O i to occur, the maximum value in

positions 1 through i1, which is equally likely to be in any of these i1 positions,

must be in one of the ﬁrst k positions. Consequently, Pr fO i g D k=.i 1/ and

Pr fS i g D k=.n.i 1//. Using equation (5.12), we have

Pr fSg D

n X

iDkC1

Pr fS i g

D

n X

iDkC1

k

n.i 1/

D

k

n

n X

iDkC1

1

i 1

D

k

n

n1 X

iDk

1

i

:

We approximate by integrals to bound this summation from above and below. By

the inequalities (A.12), we have

Z n

k

1

x

dx

n1 X

iDk

1

i

Z n1

k1

1

x

dx :

Evaluating these deﬁnite integrals gives us the bounds

k

n

.ln n ln k/ Pr fSg

k

n

.ln.n 1/ ln.k 1// ;

which provide a rather tight bound for Pr fSg. Because we wish to maximize our

probability of success, let us focus on choosing the value of k that maximizes the

lower bound on Pr fSg. (Besides, the lower-bound expression is easier to maximize

than the upper-bound expression.) Differentiating the expression .k=n/.ln nln k/

with respect to k, we obtain

1

n

.ln n ln k 1/ :

Setting this derivative equal to 0, we see that we maximize the lower bound on the

probability when ln k D ln n1 D ln.n=e/ or, equivalently, when k D n=e. Thus,

if we implement our strategy with k D n=e, we succeed in hiring our best-qualiﬁed

applicant with probability at least 1=e.

142 Chapter 5 Probabilistic Analysis and Randomized Algorithms

Exercises

5.4-1

How many people must there be in a room before the probability that someone

has the same birthday as you do is at least 1=2? How many people must there be

before the probability that at least two people have a birthday on July 4 is greater

than 1=2?

5.4-2

Suppose that we toss balls into b bins until some bin contains two balls. Each toss

is independent, and each ball is equally likely to end up in any bin. What is the

expected number of ball tosses?

5.4-3 ?

For the analysis of the birthday paradox, is it important that the birthdays be mutu-

ally independent, or is pairwise independence sufﬁcient? Justify your answer.

5.4-4 ?

How many people should be invited to a party in order to make it likely that there

are three people with the same birthday?

5.4-5 ?

What is the probability that a k-string over a set of size n forms a k-permutation?

How does this question relate to the birthday paradox?

5.4-6 ?

Suppose that n balls are tossed into n bins, where each toss is independent and the

ball is equally likely to end up in any bin. What is the expected number of empty

bins? What is the expected number of bins with exactly one ball?

5.4-7 ?

Sharpen the lower bound on streak length by showing that in n ﬂips of a fair coin,

the probability is less than 1=n that no streak longer than lg n2 lg lg n consecutive

heads occurs.

Problems for Chapter 5 143

Problems

5-1 Probabilistic counting

With a b-bit counter, we can ordinarily only count up to 2 b 1. With R. Morris’s

probabilistic counting, we can count up to a much larger value at the expense of

some loss of precision.

We let a counter value of i represent a count of n i for i D 0; 1; : : : ; 2 b 1, where

the n i form an increasing sequence of nonnegative values. We assume that the ini-

tial value of the counter is 0, representing a count of n 0 D 0. The INCREMENT

operation works on a counter containing the value i in a probabilistic manner. If

i D 2 b 1, then the operation reports an overﬂow error. Otherwise, the INCRE-

MENT operation increases the counter by 1 with probability 1=.n iC1 n i /, and it

leaves the counter unchanged with probability 1 1=.n iC1 n i /.

If we select n i D i for all i 0, then the counter is an ordinary one. More

interesting situations arise if we select, say, n i D 2 i1

for i > 0 or n i D F i (the

ith Fibonacci number—see Section 3.2).

For this problem, assume that n 2 b 1 is large enough that the probability of an

overﬂow error is negligible.

a. Show that the expected value represented by the counter after n INCREMENT

operations have been performed is exactly n.

b. The analysis of the variance of the count represented by the counter depends

on the sequence of the n i . Let us consider a simple case: n i D 100i for

all i 0. Estimate the variance in the value represented by the register after n

INCREMENT operations have been performed.

5-2 Searching an unsorted array

This problem examines three algorithms for searching for a value x in an unsorted

array A consisting of n elements.

Consider the following randomized strategy: pick a random index i into A. If

AŒi D x, then we terminate; otherwise, we continue the search by picking a new

random index into A. We continue picking random indices into A until we ﬁnd an

index j such that AŒj D x or until we have checked every element of A. Note

that we pick from the whole set of indices each time, so that we may examine a

given element more than once.

a. Write pseudocode for a procedure RANDOM-SEARCH to implement the strat-

egy above. Be sure that your algorithm terminates when all indices into A have

been picked.

144 Chapter 5 Probabilistic Analysis and Randomized Algorithms

b. Suppose that there is exactly one index i such that AŒi D x. What is the

expected number of indices into A that we must pick before we ﬁnd x and

RANDOM-SEARCH terminates?

c. Generalizing your solution to part (b), suppose that there are k 1 indices i

such that AŒi D x. What is the expected number of indices into A that we

must pick before we ﬁnd x and RANDOM-SEARCH terminates? Your answer

should be a function of n and k.

d. Suppose that there are no indices i such that AŒi D x. What is the expected

number of indices into A that we must pick before we have checked all elements

of A and RANDOM-SEARCH terminates?

Now consider a deterministic linear search algorithm, which we refer to as

DETERMINISTIC-SEARCH. Speciﬁcally, the algorithm searches A for x in order,

considering AŒ1; AŒ2; AŒ3; : : : ; AŒn until either it ﬁnds AŒi D x or it reaches

the end of the array. Assume that all possible permutations of the input array are

equally likely.

e. Suppose that there is exactly one index i such that AŒi D x. What is the

average-case running time of DETERMINISTIC-SEARCH? What is the worst-

case running time of DETERMINISTIC-SEARCH?

f. Generalizing your solution to part (e), suppose that there are k 1 indices i

such that AŒi D x. What is the average-case running time of DETERMINISTIC-

SEARCH? What is the worst-case running time of DETERMINISTIC-SEARCH?

Your answer should be a function of n and k.

g. Suppose that there are no indices i such that AŒi D x. What is the average-case

running time of DETERMINISTIC-SEARCH? What is the worst-case running

time of DETERMINISTIC-SEARCH?

Finally, consider a randomized algorithm SCRAMBLE-SEARCH that works by

ﬁrst randomly permuting the input array and then running the deterministic lin-

ear search given above on the resulting permuted array.

h. Letting k be the number of indices i such that AŒi D x, give the worst-case and

expected running times of SCRAMBLE-SEARCH for the cases in which k D 0

and k D 1. Generalize your solution to handle the case in which k 1.

i. Which of the three searching algorithms would you use? Explain your answer.

Notes for Chapter 5 145

Chapter notes

Bollob´as [53], Hofri [174], and Spencer [321] contain a wealth of advanced prob-

abilistic techniques. The advantages of randomized algorithms are discussed and

surveyed by Karp [200] and Rabin [288]. The textbook by Motwani and Raghavan

[262] gives an extensive treatment of randomized algorithms.

Several variants of the hiring problem have been widely studied. These problems

are more commonly referred to as “secretary problems.” An example of work in

this area is the paper by Ajtai, Meggido, and Waarts [11].

II Sorting and Order Statistics

Introduction

This part presents several algorithms that solve the following sorting problem:

Input: A sequence of n numbers ha 1 ; a 2 ; : : : ; a n i.

Output: A permutation (reordering) ha 0

1

; a 0

2

; : : : ; a 0

n

i of the input sequence such

that a 0

1

a 0

2

a 0

n

.

The input sequence is usually an n-element array, although it may be represented

in some other fashion, such as a linked list.

The structure of the data

In practice, the numbers to be sorted are rarely isolated values. Each is usually part

of a collection of data called a record. Each record contains a key, which is the

value to be sorted. The remainder of the record consists of satellite data, which are

usually carried around with the key. In practice, when a sorting algorithm permutes

the keys, it must permute the satellite data as well. If each record includes a large

amount of satellite data, we often permute an array of pointers to the records rather

than the records themselves in order to minimize data movement.

In a sense, it is these implementation details that distinguish an algorithm from

a full-blown program. A sorting algorithm describes the method by which we

determine the sorted order, regardless of whether we are sorting individual numbers

or large records containing many bytes of satellite data. Thus, when focusing on the

problem of sorting, we typically assume that the input consists only of numbers.

Translating an algorithm for sorting numbers into a program for sorting records

148 Part II Sorting and Order Statistics

is conceptually straightforward, although in a given engineering situation other

subtleties may make the actual programming task a challenge.

Why sorting?

Many computer scientists consider sorting to be the most fundamental problem in

the study of algorithms. There are several reasons:

Sometimes an application inherently needs to sort information. For example,

in order to prepare customer statements, banks need to sort checks by check

number.

Algorithms often use sorting as a key subroutine. For example, a program that

renders graphical objects which are layered on top of each other might have

to sort the objects according to an “above” relation so that it can draw these

objects from bottom to top. We shall see numerous algorithms in this text that

use sorting as a subroutine.

We can draw from among a wide variety of sorting algorithms, and they em-

ploy a rich set of techniques. In fact, many important techniques used through-

out algorithm design appear in the body of sorting algorithms that have been

developed over the years. In this way, sorting is also a problem of historical

interest.

We can prove a nontrivial lower bound for sorting (as we shall do in Chapter 8).

Our best upper bounds match the lower bound asymptotically, and so we know

that our sorting algorithms are asymptotically optimal. Moreover, we can use

the lower bound for sorting to prove lower bounds for certain other problems.

Many engineering issues come to the fore when implementing sorting algo-

rithms. The fastest sorting program for a particular situation may depend on

many factors, such as prior knowledge about the keys and satellite data, the

memory hierarchy (caches and virtual memory) of the host computer, and the

software environment. Many of these issues are best dealt with at the algorith-

mic level, rather than by “tweaking” the code.

Sorting algorithms

We introduced two algorithms that sort n real numbers in Chapter 2. Insertion sort

takes ‚.n 2 / time in the worst case. Because its inner loops are tight, however,

it is a fast in-place sorting algorithm for small input sizes. (Recall that a sorting

algorithm sorts in place if only a constant number of elements of the input ar-

ray are ever stored outside the array.) Merge sort has a better asymptotic running

time, ‚.n lg n/, but the MERGE procedure it uses does not operate in place.

Part II Sorting and Order Statistics 149

In this part, we shall introduce two more algorithms that sort arbitrary real num-

bers. Heapsort, presented in Chapter 6, sorts n numbers in place in O.n lg n/ time.

It uses an important data structure, called a heap, with which we can also imple-

ment a priority queue.

Quicksort, in Chapter 7, also sorts n numbers in place, but its worst-case running

time is ‚.n 2 /. Its expected running time is ‚.n lg n/, however, and it generally

outperforms heapsort in practice. Like insertion sort, quicksort has tight code, and

so the hidden constant factor in its running time is small. It is a popular algorithm

for sorting large input arrays.

Insertion sort, merge sort, heapsort, and quicksort are all comparison sorts: they

determine the sorted order of an input array by comparing elements. Chapter 8 be-

gins by introducing the decision-tree model in order to study the performance limi-

tations of comparison sorts. Using this model, we prove a lower bound of .n lg n/

on the worst-case running time of any comparison sort on n inputs, thus showing

that heapsort and merge sort are asymptotically optimal comparison sorts.

Chapter 8 then goes on to show that we can beat this lower bound of .n lg n/

if we can gather information about the sorted order of the input by means other

than comparing elements. The counting sort algorithm, for example, assumes that

the input numbers are in the set f0; 1; : : : ; kg. By using array indexing as a tool

for determining relative order, counting sort can sort n numbers in ‚.k C n/ time.

Thus, when k D O.n/, counting sort runs in time that is linear in the size of the

input array. A related algorithm, radix sort, can be used to extend the range of

counting sort. If there are n integers to sort, each integer has d digits, and each

digit can take on up to k possible values, then radix sort can sort the numbers

in ‚.d.n C k// time. When d is a constant and k is O.n/, radix sort runs in

linear time. A third algorithm, bucket sort, requires knowledge of the probabilistic

distribution of numbers in the input array. It can sort n real numbers uniformly

distributed in the half-open interval Œ0; 1/ in average-case O.n/ time.

The following table summarizes the running times of the sorting algorithms from

Chapters 2 and 6–8. As usual, n denotes the number of items to sort. For counting

sort, the items to sort are integers in the set f0; 1; : : : ; kg. For radix sort, each item

is a d-digit number, where each digit takes on k possible values. For bucket sort,

we assume that the keys are real numbers uniformly distributed in the half-open

interval Œ0; 1/. The rightmost column gives the average-case or expected running

time, indicating which it gives when it differs from the worst-case running time.

We omit the average-case running time of heapsort because we do not analyze it in

this book.

150 Part II Sorting and Order Statistics

Worst-case Average-case/expected

Algorithm running time running time

Insertion sort ‚.n 2 / ‚.n 2 /

Merge sort ‚.n lg n/ ‚.n lg n/

Heapsort O.n lg n/ —

Quicksort ‚.n 2 / ‚.n lg n/ (expected)

Counting sort ‚.k C n/ ‚.k C n/

Radix sort ‚.d.n C k// ‚.d.n C k//

Bucket sort ‚.n 2 / ‚.n/ (average-case)

Order statistics

The ith order statistic of a set of n numbers is the ith smallest number in the set.

We can, of course, select the ith order statistic by sorting the input and indexing

the ith element of the output. With no assumptions about the input distribution,

this method runs in .n lg n/ time, as the lower bound proved in Chapter 8 shows.

In Chapter 9, we show that we can ﬁnd the ith smallest element in O.n/ time,

even when the elements are arbitrary real numbers. We present a randomized algo-

rithm with tight pseudocode that runs in ‚.n 2 / time in the worst case, but whose

expected running time is O.n/. We also give a more complicated algorithm that

runs in O.n/ worst-case time.

Background

Although most of this part does not rely on difﬁcult mathematics, some sections

do require mathematical sophistication. In particular, analyses of quicksort, bucket

sort, and the order-statistic algorithm use probability, which is reviewed in Ap-

pendix C, and the material on probabilistic analysis and randomized algorithms in

Chapter 5. The analysis of the worst-case linear-time algorithm for order statis-

tics involves somewhat more sophisticated mathematics than the other worst-case

analyses in this part.

6 Heapsort

In this chapter, we introduce another sorting algorithm: heapsort. Like merge sort,

but unlike insertion sort, heapsort’s running time is O.n lg n/. Like insertion sort,

but unlike merge sort, heapsort sorts in place: only a constant number of array

elements are stored outside the input array at any time. Thus, heapsort combines

the better attributes of the two sorting algorithms we have already discussed.

Heapsort also introduces another algorithm design technique: using a data struc-

ture, in this case one we call a “heap,” to manage information. Not only is the heap

data structure useful for heapsort, but it also makes an efﬁcient priority queue. The

heap data structure will reappear in algorithms in later chapters.

The term “heap” was originally coined in the context of heapsort, but it has since

come to refer to “garbage-collected storage,” such as the programming languages

Java and Lisp provide. Our heap data structure is not garbage-collected storage,

and whenever we refer to heaps in this book, we shall mean a data structure rather

than an aspect of garbage collection.

6.1 Heaps

The (binary) heap data structure is an array object that we can view as a

nearly complete binary tree (see Section B.5.3), as shown in Figure 6.1. Each

node of the tree corresponds to an element of the array. The tree is com-

pletely ﬁlled on all levels except possibly the lowest, which is ﬁlled from the

left up to a point. An array A that represents a heap is an object with two at-

tributes: A:length, which (as usual) gives the number of elements in the array, and

A:heap-size, which represents how many elements in the heap are stored within

array A. That is, although AŒ1 : : A:length may contain numbers, only the ele-

ments in AŒ1 : : A:heap-size, where 0 A:heap-size A:length, are valid ele-

ments of the heap. The root of the tree is AŒ1, and given the index i of a node, we

can easily compute the indices of its parent, left child, and right child:

152 Chapter 6 Heapsort

(a)

16 14 10 8 7 9 3 2 4 1

1 2 3 4 5 6 7 8 9 10

(b)

1

2 3

4 5 6 7

8 9 10

16

14 10

8 7 9 3

2 4 1

Figure 6.1 A max-heap viewed as (a) a binary tree and (b) an array. The number within the circle

at each node in the tree is the value stored at that node. The number above a node is the corresponding

index in the array. Above and below the array are lines showing parent-child relationships; parents

are always to the left of their children. The tree has height three; the node at index 4 (with value 8)

has height one.

PARENT.i/

1 return bi=2c

LEFT.i/

1 return 2i

RIGHT.i/

1 return 2i C 1

On most computers, the LEFT procedure can compute 2i in one instruction by

simply shifting the binary representation of i left by one bit position. Similarly, the

RIGHT procedure can quickly compute 2i C1 by shifting the binary representation

of i left by one bit position and then adding in a 1 as the low-order bit. The

PARENT procedure can compute bi=2c by shifting i right one bit position. Good

implementations of heapsort often implement these procedures as “macros” or “in-

line” procedures.

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds,

the values in the nodes satisfy a heap property, the speciﬁcs of which depend on

the kind of heap. In a max-heap, the max-heap property is that for every node i

other than the root,

AŒPARENT.i/ AŒi ;

that is, the value of a node is at most the value of its parent. Thus, the largest

element in a max-heap is stored at the root, and the subtree rooted at a node contains

6.1 Heaps 153

values no larger than that contained at the node itself. A min-heap is organized in

the opposite way; the min-heap property is that for every node i other than the

root,

AŒPARENT.i/ AŒi :

The smallest element in a min-heap is at the root.

For the heapsort algorithm, we use max-heaps. Min-heaps commonly imple-

ment priority queues, which we discuss in Section 6.5. We shall be precise in

specifying whether we need a max-heap or a min-heap for any particular applica-

tion, and when properties apply to either max-heaps or min-heaps, we just use the

term “heap.”

Viewing a heap as a tree, we deﬁne the height of a node in a heap to be the

number of edges on the longest simple downward path from the node to a leaf, and

we deﬁne the height of the heap to be the height of its root. Since a heap of n ele-

ments is based on a complete binary tree, its height is ‚.lg n/ (see Exercise 6.1-2).

We shall see that the basic operations on heaps run in time at most proportional

to the height of the tree and thus take O.lg n/ time. The remainder of this chapter

presents some basic procedures and shows how they are used in a sorting algorithm

and a priority-queue data structure.

The MAX-HEAPIFY procedure, which runs in O.lg n/ time, is the key to main-

taining the max-heap property.

The BUILD-MAX-HEAP procedure, which runs in linear time, produces a max-

heap from an unordered input array.

The HEAPSORT procedure, which runs in O.n lg n/ time, sorts an array in

place.

The MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY,

and HEAP-MAXIMUM procedures, which run in O.lg n/ time, allow the heap

data structure to implement a priority queue.

Exercises

6.1-1

What are the minimum and maximum numbers of elements in a heap of height h?

6.1-2

Show that an n-element heap has height blg nc.

6.1-3

Show that in any subtree of a max-heap, the root of the subtree contains the largest

value occurring anywhere in that subtree.

154 Chapter 6 Heapsort

6.1-4

Where in a max-heap might the smallest element reside, assuming that all elements

are distinct?

6.1-5

Is an array that is in sorted order a min-heap?

6.1-6

Is the array with values h23; 17; 14; 6; 13; 10; 1; 5; 7; 12i a max-heap?

6.1-7

Show that, with the array representation for storing an n-element heap, the leaves

are the nodes indexed by bn=2c C 1; bn=2c C 2; : : : ; n.

6.2 Maintaining the heap property

In order to maintain the max-heap property, we call the procedure MAX-HEAPIFY.

Its inputs are an array A and an index i into the array. When it is called, MAX-

HEAPIFY assumes that the binary trees rooted at LEFT.i/ and RIGHT.i/ are max-

heaps, but that AŒi might be smaller than its children, thus violating the max-heap

property. MAX-HEAPIFY lets the value at AŒi “ﬂoat down” in the max-heap so

that the subtree rooted at index i obeys the max-heap property.

MAX-HEAPIFY.A; i/

1 l D LEFT.i/

2 r D RIGHT.i/

3 if l A:heap-size and AŒl > AŒi

4 largest D l

5 else largest D i

6 if r A:heap-size and AŒr > AŒlargest

7 largest D r

8 if largest ¤ i

9 exchange AŒi with AŒlargest

10 MAX-HEAPIFY.A; largest/

Figure 6.2 illustrates the action of MAX-HEAPIFY. At each step, the largest of

the elements AŒi, AŒLEFT.i/, and AŒRIGHT.i/ is determined, and its index is

stored in largest. If AŒi is largest, then the subtree rooted at node i is already a

max-heap and the procedure terminates. Otherwise, one of the two children has the

largest element, and AŒi is swapped with AŒlargest, which causes node i and its

6.2 Maintaining the heap property 155

16

4 10

14 7 9

2 8 1

(a)

16

14 10

4 7 9 3

2 8 1

(b)

16

14 10

8 7 9 3

2 4 1

(c)

3

1

3

4 5 6 7

9 10

2

8

1

3

4 5 6 7

9 10

2

8

1

3

4 5 6 7

9 10

2

8

i

i

i

Figure 6.2 The action of MAX-HEAPIFY.A; 2/, where A:heap-size D 10. (a) The initial con-

ﬁguration, with AŒ2 at node i D 2 violating the max-heap property since it is not larger than

both children. The max-heap property is restored for node 2 in (b) by exchanging AŒ2 with AŒ4,

which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY.A; 4/ now

has i D 4. After swapping AŒ4 with AŒ9, as shown in (c), node 4 is ﬁxed up, and the recursive call

MAX-HEAPIFY.A; 9/ yields no further change to the data structure.

children to satisfy the max-heap property. The node indexed by largest, however,

now has the original value AŒi, and thus the subtree rooted at largest might violate

the max-heap property. Consequently, we call MAX-HEAPIFY recursively on that

subtree.

The running time of MAX-HEAPIFY on a subtree of size n rooted at a given

node i is the ‚.1/ time to ﬁx up the relationships among the elements AŒi,

AŒLEFT.i/, and AŒRIGHT.i/, plus the time to run MAX-HEAPIFY on a subtree

rooted at one of the children of node i (assuming that the recursive call occurs).

The children’s subtrees each have size at most 2n=3—the worst case occurs when

the bottom level of the tree is exactly half full—and therefore we can describe the

running time of MAX-HEAPIFY by the recurrence

T .n/ T .2n=3/ C ‚.1/ :

156 Chapter 6 Heapsort

The solution to this recurrence, by case 2 of the master theorem (Theorem 4.1),

is T .n/ D O.lg n/. Alternatively, we can characterize the running time of MAX-

HEAPIFY on a node of height h as O.h/.

Exercises

6.2-1

Using Figure 6.2 as a model, illustrate the operation of MAX-HEAPIFY.A; 3/ on

the array A D h27; 17; 3; 16; 13; 10; 1; 5; 7; 12; 4; 8; 9; 0i.

6.2-2

Starting with the procedure MAX-HEAPIFY, write pseudocode for the procedure

MIN-HEAPIFY.A; i/, which performs the corresponding manipulation on a min-

heap. How does the running time of MIN-HEAPIFY compare to that of MAX-

HEAPIFY?

6.2-3

What is the effect of calling MAX-HEAPIFY.A; i/ when the element AŒi is larger

than its children?

6.2-4

What is the effect of calling MAX-HEAPIFY.A; i/ for i > A:heap-size=2?

6.2-5

The code for MAX-HEAPIFY is quite efﬁcient in terms of constant factors, except

possibly for the recursive call in line 10, which might cause some compilers to

produce inefﬁcient code. Write an efﬁcient MAX-HEAPIFY that uses an iterative

control construct (a loop) instead of recursion.

6.2-6

Show that the worst-case running time of MAX-HEAPIFY on a heap of size n

is .lg n/. (Hint: For a heap with n nodes, give node values that cause MAX-

HEAPIFY to be called recursively at every node on a simple path from the root

down to a leaf.)

6.3 Building a heap

We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an

array AŒ1 : : n, where n D A:length, into a max-heap. By Exercise 6.1-7, the

elements in the subarray AŒ.bn=2cC1/ : : n are all leaves of the tree, and so each is

6.3 Building a heap 157

a 1-element heap to begin with. The procedure BUILD-MAX-HEAP goes through

the remaining nodes of the tree and runs MAX-HEAPIFY on each one.

BUILD-MAX-HEAP.A/

1 A:heap-size D A:length

2 for i D bA:length=2c downto 1

3 MAX-HEAPIFY.A; i/

Figure 6.3 shows an example of the action of BUILD-MAX-HEAP.

To show why BUILD-MAX-HEAP works correctly, we use the following loop

invariant:

At the start of each iteration of the for loop of lines 2–3, each node i C 1;

i C 2; : : : ; n is the root of a max-heap.

We need to show that this invariant is true prior to the ﬁrst loop iteration, that each

iteration of the loop maintains the invariant, and that the invariant provides a useful

property to show correctness when the loop terminates.

Initialization: Prior to the ﬁrst iteration of the loop, i D bn=2c. Each node

bn=2c C 1; bn=2c C 2; : : : ; n is a leaf and is thus the root of a trivial max-heap.

Maintenance: To see that each iteration maintains the loop invariant, observe that

the children of node i are numbered higher than i. By the loop invariant, there-

fore, they are both roots of max-heaps. This is precisely the condition required

for the call MAX-HEAPIFY.A; i/ to make node i a max-heap root. Moreover,

the MAX-HEAPIFY call preserves the property that nodes i C 1; i C 2; : : : ; n

are all roots of max-heaps. Decrementing i in the for loop update reestablishes

the loop invariant for the next iteration.

Termination: At termination, i D 0. By the loop invariant, each node 1; 2; : : : ; n

is the root of a max-heap. In particular, node 1 is.

We can compute a simple upper bound on the running time of BUILD-MAX-

HEAP as follows. Each call to MAX-HEAPIFY costs O.lg n/ time, and BUILD-

MAX-HEAP makes O.n/ such calls. Thus, the running time is O.n lg n/. This

upper bound, though correct, is not asymptotically tight.

We can derive a tighter bound by observing that the time for MAX-HEAPIFY to

run at a node varies with the height of the node in the tree, and the heights of most

nodes are small. Our tighter analysis relies on the properties that an n-element heap

has height blg nc (see Exercise 6.1-2) and at most

˙

n=2 hC1

nodes of any height h

(see Exercise 6.3-3).

The time required by MAX-HEAPIFY when called on a node of height h is O.h/,

and so we can express the total cost of BUILD-MAX-HEAP as being bounded from

above by

158 Chapter 6 Heapsort

1

2 3

4 5 6 7

8 9 10

1

2 3

4 5 6 7

8 9 10

1

2 3

4 5 6 7

8 9 10

1

2 3

4 5 6 7

8 9 10

1

2 3

4 5 6 7

8 9 10

1

2 3

4 5 6 7

8 9 10

4

1 3

2 9 10

14 8 7

(a)

16

4 1 2 3 16 9 10 14 8 7

4

1 3

2 9 10

14 8 7

(b)

16

4

1 3

14 9 10

2 8 7

(c)

16

4

1 10

14 9 3

2 8 7

(d)

16

4

16 10

14 9 3

2 8 1

(e)

7

16

14 10

8 9 3

2 4 1

(f)

7

A

i i

i i

i

Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to

MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. (a) A 10-element input array A and the bi-

nary tree it represents. The ﬁgure shows that the loop index i refers to node 5 before the call

MAX-HEAPIFY.A; i/. (b) The data structure that results. The loop index i for the next iteration

refers to node 4. (c)–(e) Subsequent iterations of the for loop in BUILD-MAX-HEAP. Observe that

whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps.

(f) The max-heap after BUILD-MAX-HEAP ﬁnishes.

6.4 The heapsort algorithm 159

blg nc X

hD0

l

n

2 hC1

m

O.h/ D O

n

blg nc X

hD0

h

2 h

!

:

We evalaute the last summation by substituting x D 1=2 in the formula (A.8),

yielding

1 X

hD0

h

2 h

D

1=2

.1 1=2/ 2

D 2 :

Thus, we can bound the running time of BUILD-MAX-HEAP as

O

n

blg nc X

hD0

h

2 h

!

D O

n

1 X

hD0

h

2 h

!

D O.n/ :

Hence, we can build a max-heap from an unordered array in linear time.

We can build a min-heap by the procedure BUILD-MIN-HEAP, which is the

same as BUILD-MAX-HEAP but with the call to MAX-HEAPIFY in line 3 replaced

by a call to MIN-HEAPIFY (see Exercise 6.2-2). BUILD-MIN-HEAP produces a

min-heap from an unordered linear array in linear time.

Exercises

6.3-1

Using Figure 6.3 as a model, illustrate the operation of BUILD-MAX-HEAP on the

array A D h5; 3; 17; 10; 84; 19; 6; 22; 9i.

6.3-2

Why do we want the loop index i in line 2 of BUILD-MAX-HEAP to decrease from

bA:length=2c to 1 rather than increase from 1 to bA:length=2c?

6.3-3

Show that there are at most

˙

n=2 hC1

nodes of height h in any n-element heap.

6.4 The heapsort algorithm

The heapsort algorithm starts by using BUILD-MAX-HEAP to build a max-heap

on the input array AŒ1 : : n, where n D A:length. Since the maximum element

of the array is stored at the root AŒ1, we can put it into its correct ﬁnal position

160 Chapter 6 Heapsort

by exchanging it with AŒn. If we now discard node n from the heap—and we

can do so by simply decrementing A:heap-size—we observe that the children of

the root remain max-heaps, but the new root element might violate the max-heap

property. All we need to do to restore the max-heap property, however, is call

MAX-HEAPIFY.A; 1/, which leaves a max-heap in AŒ1 : : n 1. The heapsort

algorithm then repeats this process for the max-heap of size n 1 down to a heap

of size 2. (See Exercise 6.4-2 for a precise loop invariant.)

HEAPSORT.A/

1 BUILD-MAX-HEAP.A/

2 for i D A:length downto 2

3 exchange AŒ1 with AŒi

4 A:heap-size D A:heap-size 1

5 MAX-HEAPIFY.A; 1/

Figure 6.4 shows an example of the operation of HEAPSORT after line 1 has built

the initial max-heap. The ﬁgure shows the max-heap before the ﬁrst iteration of

the for loop of lines 2–5 and after each iteration.

The HEAPSORT procedure takes time O.n lg n/, since the call to BUILD-MAX-

HEAP takes time O.n/ and each of the n 1 calls to MAX-HEAPIFY takes

time O.lg n/.

Exercises

6.4-1

Using Figure 6.4 as a model, illustrate the operation of HEAPSORT on the array

A D h5; 13; 2; 25; 7; 17; 20; 8; 4i.

6.4-2

Argue the correctness of HEAPSORT using the following loop invariant:

At the start of each iteration of the for loop of lines 2–5, the subarray

AŒ1 : : i is a max-heap containing the i smallest elements of AŒ1 : : n, and

the subarray AŒi C 1 : : n contains the n i largest elements of AŒ1 : : n,

sorted.

6.4-3

What is the running time of HEAPSORT on an array A of length n that is already

sorted in increasing order? What about decreasing order?

6.4-4

Show that the worst-case running time of HEAPSORT is .n lg n/.

6.4 The heapsort algorithm 161

(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k)

1 2 3 4 7 8 9 10 14 16

10

2

1 3

4 7 8 9

16 14

1

2 3

4 7 8 9

16 14 10

3

2 1

9 8 7 4

10 14 16

4

2 3

9 8 7 1

10 14 16

8

3 7

4 2 1 9

16 14 10

7

4 3

9 8 2 1

10 14 16

9

8 3

2 1 7 4

16 14 10

10

8 9

3 1 7 4

16 14 2

14

8 10

3 9 7 4

16 1 2

16

14 10

3 9 7 8

1 4 2

A

i

i

i

i i

i i

i

i

Figure 6.4 The operation of HEAPSORT. (a) The max-heap data structure just after BUILD-MAX-

HEAP has built it in line 1. (b)–(j) The max-heap just after each call of MAX-HEAPIFY in line 5,

showing the value of i at that time. Only lightly shaded nodes remain in the heap. (k) The resulting

sorted array A.

162 Chapter 6 Heapsort

6.4-5 ?

Show that when all elements are distinct, the best-case running time of HEAPSORT

is .n lg n/.

6.5 Priority queues

Heapsort is an excellent algorithm, but a good implementation of quicksort, pre-

sented in Chapter 7, usually beats it in practice. Nevertheless, the heap data struc-

ture itself has many uses. In this section, we present one of the most popular ap-

plications of a heap: as an efﬁcient priority queue. As with heaps, priority queues

come in two forms: max-priority queues and min-priority queues. We will focus

here on how to implement max-priority queues, which are in turn based on max-

heaps; Exercise 6.5-3 asks you to write the procedures for min-priority queues.

A priority queue is a data structure for maintaining a set S of elements, each

with an associated value called a key. A max-priority queue supports the following

operations:

INSERT.S; x/ inserts the element x into the set S, which is equivalent to the oper-

ation S D S [ fxg.

MAXIMUM.S/ returns the element of S with the largest key.

EXTRACT-MAX.S/ removes and returns the element of S with the largest key.

INCREASE-KEY.S; x; k/ increases the value of element x’s key to the new value k,

which is assumed to be at least as large as x’s current key value.

Among their other applications, we can use max-priority queues to schedule

jobs on a shared computer. The max-priority queue keeps track of the jobs to

be performed and their relative priorities. When a job is ﬁnished or interrupted,

the scheduler selects the highest-priority job from among those pending by calling

EXTRACT-MAX. The scheduler can add a new job to the queue at any time by

calling INSERT.

Alternatively, a min-priority queue supports the operations INSERT, MINIMUM,

EXTRACT-MIN, and DECREASE-KEY. A min-priority queue can be used in an

event-driven simulator. The items in the queue are events to be simulated, each

with an associated time of occurrence that serves as its key. The events must be

simulated in order of their time of occurrence, because the simulation of an event

can cause other events to be simulated in the future. The simulation program calls

EXTRACT-MIN at each step to choose the next event to simulate. As new events are

produced, the simulator inserts them into the min-priority queue by calling INSERT.

6.5 Priority queues 163

We shall see other uses for min-priority queues, highlighting the DECREASE-KEY

operation, in Chapters 23 and 24.

Not surprisingly, we can use a heap to implement a priority queue. In a given ap-

plication, such as job scheduling or event-driven simulation, elements of a priority

queue correspond to objects in the application. We often need to determine which

application object corresponds to a given priority-queue element, and vice versa.

When we use a heap to implement a priority queue, therefore, we often need to

store a handle to the corresponding application object in each heap element. The

exact makeup of the handle (such as a pointer or an integer) depends on the ap-

plication. Similarly, we need to store a handle to the corresponding heap element

in each application object. Here, the handle would typically be an array index.

Because heap elements change locations within the array during heap operations,

an actual implementation, upon relocating a heap element, would also have to up-

date the array index in the corresponding application object. Because the details

of accessing application objects depend heavily on the application and its imple-

mentation, we shall not pursue them here, other than noting that in practice, these

handles do need to be correctly maintained.

Now we discuss how to implement the operations of a max-priority queue. The

procedure HEAP-MAXIMUM implements the MAXIMUM operation in ‚.1/ time.

HEAP-MAXIMUM.A/

1 return AŒ1

The procedure HEAP-EXTRACT-MAX implements the EXTRACT-MAX opera-

tion. It is similar to the for loop body (lines 3–5) of the HEAPSORT procedure.

HEAP-EXTRACT-MAX.A/

1 if A:heap-size < 1

2 error “heap underﬂow”

3 max D AŒ1

4 AŒ1 D AŒA:heap-size

5 A:heap-size D A:heap-size 1

6 MAX-HEAPIFY.A; 1/

7 return max

The running time of HEAP-EXTRACT-MAX is O.lg n/, since it performs only a

constant amount of work on top of the O.lg n/ time for MAX-HEAPIFY.

The procedure HEAP-INCREASE-KEY implements the INCREASE-KEY opera-

tion. An index i into the array identiﬁes the priority-queue element whose key we

wish to increase. The procedure ﬁrst updates the key of element AŒi to its new

value. Because increasing the key of AŒi might violate the max-heap property,

164 Chapter 6 Heapsort

the procedure then, in a manner reminiscent of the insertion loop (lines 5–7) of

INSERTION-SORT from Section 2.1, traverses a simple path from this node toward

the root to ﬁnd a proper place for the newly increased key. As HEAP-INCREASE-

KEY traverses this path, it repeatedly compares an element to its parent, exchang-

ing their keys and continuing if the element’s key is larger, and terminating if the el-

ement’s key is smaller, since the max-heap property now holds. (See Exercise 6.5-5

for a precise loop invariant.)

HEAP-INCREASE-KEY.A; i; key/

1 if key < AŒi

2 error “new key is smaller than current key”

3 AŒi D key

4 while i > 1 and AŒPARENT.i/ < AŒi

5 exchange AŒi with AŒPARENT.i/

6 i D PARENT.i/

Figure 6.5 shows an example of a HEAP-INCREASE-KEY operation. The running

time of HEAP-INCREASE-KEY on an n-element heap is O.lg n/, since the path

traced from the node updated in line 3 to the root has length O.lg n/.

The procedure MAX-HEAP-INSERT implements the INSERT operation. It takes

as an input the key of the new element to be inserted into max-heap A. The proce-

dure ﬁrst expands the max-heap by adding to the tree a new leaf whose key is 1.

Then it calls HEAP-INCREASE-KEY to set the key of this new node to its correct

value and maintain the max-heap property.

MAX-HEAP-INSERT.A; key/

1 A:heap-size D A:heap-size C 1

2 AŒA:heap-size D 1

3 HEAP-INCREASE-KEY.A; A:heap-size; key/

The running time of MAX-HEAP-INSERT on an n-element heap is O.lg n/.

In summary, a heap can support any priority-queue operation on a set of size n

in O.lg n/ time.

Exercises

6.5-1

Illustrate the operation of HEAP-EXTRACT-MAX on the heap A D h15; 13; 9; 5;

12; 8; 7; 4; 0; 6; 2; 1i.

6.5 Priority queues 165

16

14 10

8 7 9 3

2 4 1

(a)

i

16

14 10

8 7 9 3

2 15 1

(b)

16

14 10

8

7 9 3

2

15

1

(c)

i

i

16

14

10

8

7 9 3

2

15

1

(d)

i

Figure 6.5 The operation of HEAP-INCREASE-KEY. (a) The max-heap of Figure 6.4(a) with a

node whose index is i heavily shaded. (b) This node has its key increased to 15. (c) After one

iteration of the while loop of lines 4–6, the node and its parent have exchanged keys, and the index i

moves up to the parent. (d) The max-heap after one more iteration of the while loop. At this point,

AŒPARENT.i/ AŒi. The max-heap property now holds and the procedure terminates.

6.5-2

Illustrate the operation of MAX-HEAP-INSERT.A; 10/ on the heap A D h15;13;9;

5; 12; 8; 7; 4; 0; 6; 2; 1i.

6.5-3

Write pseudocode for the procedures HEAP-MINIMUM, HEAP-EXTRACT-MIN,

HEAP-DECREASE-KEY, and MIN-HEAP-INSERT that implement a min-priority

queue with a min-heap.

6.5-4

Why do we bother setting the key of the inserted node to 1 in line 2 of MAX-

HEAP-INSERT when the next thing we do is increase its key to the desired value?

166 Chapter 6 Heapsort

6.5-5

Argue the correctness of HEAP-INCREASE-KEY using the following loop invari-

ant:

At the start of each iteration of the while loop of lines 4–6, the subarray

AŒ1 : : A:heap-size satisﬁes the max-heap property, except that there may

be one violation: AŒi may be larger than AŒPARENT.i/.

You may assume that the subarray AŒ1 : : A:heap-size satisﬁes the max-heap prop-

erty at the time HEAP-INCREASE-KEY is called.

6.5-6

Each exchange operation on line 5 of HEAP-INCREASE-KEY typically requires

three assignments. Show how to use the idea of the inner loop of INSERTION-

SORT to reduce the three assignments down to just one assignment.

6.5-7

Show how to implement a ﬁrst-in, ﬁrst-out queue with a priority queue. Show

how to implement a stack with a priority queue. (Queues and stacks are deﬁned in

Section 10.1.)

6.5-8

The operation HEAP-DELETE.A; i/ deletes the item in node i from heap A. Give

an implementation of HEAP-DELETE that runs in O.lg n/ time for an n-element

max-heap.

6.5-9

Give an O.n lg k/-time algorithm to merge k sorted lists into one sorted list,

where n is the total number of elements in all the input lists. (Hint: Use a min-

heap for k-way merging.)

Problems

6-1 Building a heap using insertion

We can build a heap by repeatedly calling MAX-HEAP-INSERT to insert the ele-

ments into the heap. Consider the following variation on the BUILD-MAX-HEAP

procedure:

Problems for Chapter 6 167

BUILD-MAX-HEAP

0 .A/

1 A:heap-size D 1

2 for i D 2 to A:length

3 MAX-HEAP-INSERT.A; AŒi/

a. Do the procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP

0

always create

the same heap when run on the same input array? Prove that they do, or provide

a counterexample.

b. Show that in the worst case, BUILD-MAX-HEAP

0

requires ‚.n lg n/ time to

build an n-element heap.

6-2 Analysis of d-ary heaps

A d-ary heap is like a binary heap, but (with one possible exception) non-leaf

nodes have d children instead of 2 children.

a. How would you represent a d-ary heap in an array?

b. What is the height of a d-ary heap of n elements in terms of n and d?

c. Give an efﬁcient implementation of EXTRACT-MAX in a d-ary max-heap. An-

alyze its running time in terms of d and n.

d. Give an efﬁcient implementation of INSERT in a d-ary max-heap. Analyze its

running time in terms of d and n.

e. Give an efﬁcient implementation of INCREASE-KEY.A; i; k/, which ﬂags an

error if k < AŒi, but otherwise sets AŒi D k and then updates the d-ary max-

heap structure appropriately. Analyze its running time in terms of d and n.

6-3 Young tableaus

An m n Young tableau is an m n matrix such that the entries of each row are

in sorted order from left to right and the entries of each column are in sorted order

from top to bottom. Some of the entries of a Young tableau may be 1, which we

treat as nonexistent elements. Thus, a Young tableau can be used to hold r mn

ﬁnite numbers.

a. Draw a 4 4 Young tableau containing the elements f9; 16; 3; 2; 4; 8; 5; 14; 12g.

b. Argue that an m n Young tableau Y is empty if Y Œ1; 1 D 1. Argue that Y

is full (contains mn elements) if Y Œm; n < 1.

168 Chapter 6 Heapsort

c. Give an algorithm to implement EXTRACT-MIN on a nonempty m n Young

tableau that runs in O.m C n/ time. Your algorithm should use a recur-

sive subroutine that solves an m n problem by recursively solving either

an .m 1/ n or an m .n 1/ subproblem. (Hint: Think about MAX-

HEAPIFY.) Deﬁne T .p/, where p D m C n, to be the maximum running time

of EXTRACT-MIN on any m n Young tableau. Give and solve a recurrence

for T .p/ that yields the O.m C n/ time bound.

d. Show how to insert a new element into a nonfull m n Young tableau in

O.m C n/ time.

e. Using no other sorting method as a subroutine, show how to use an n n Young

tableau to sort n 2

numbers in O.n 3 / time.

f. Give an O.m C n/-time algorithm to determine whether a given number is

stored in a given m n Young tableau.

Chapter notes

The heapsort algorithm was invented by Williams [357], who also described how

to implement a priority queue with a heap. The BUILD-MAX-HEAP procedure

was suggested by Floyd [106].

We use min-heaps to implement min-priority queues in Chapters 16, 23, and 24.

We also give an implementation with improved time bounds for certain operations

in Chapter 19 and, assuming that the keys are drawn from a bounded set of non-

negative integers, Chapter 20.

If the data are b-bit integers, and the computer memory consists of addressable

b-bit words, Fredman and Willard [115] showed how to implement MINIMUM in

O.1/ time and INSERT and EXTRACT-MIN in O.

p

lg n/ time. Thorup [337] has

improved the O.

p

lg n/ bound to O.lg lg n/ time. This bound uses an amount of

space unbounded in n, but it can be implemented in linear space by using random-

ized hashing.

An important special case of priority queues occurs when the sequence of

EXTRACT-MIN operations is monotone, that is, the values returned by succes-

sive EXTRACT-MIN operations are monotonically increasing over time. This case

arises in several important applications, such as Dijkstra’s single-source shortest-

paths algorithm, which we discuss in Chapter 24, and in discrete-event simula-

tion. For Dijkstra’s algorithm it is particularly important that the DECREASE-KEY

operation be implemented efﬁciently. For the monotone case, if the data are in-

tegers in the range 1; 2; : : : ; C , Ahuja, Mehlhorn, Orlin, and Tarjan [8] describe

Notes for Chapter 6 169

how to implement EXTRACT-MIN and INSERT in O.lg C / amortized time (see

Chapter 17 for more on amortized analysis) and DECREASE-KEY in O.1/ time,

using a data structure called a radix heap. The O.lg C / bound can be improved

to O.

p

lg C / using Fibonacci heaps (see Chapter 19) in conjunction with radix

heaps. Cherkassky, Goldberg, and Silverstein [65] further improved the bound to

O.lg

1=3C

C / expected time by combining the multilevel bucketing structure of

Denardo and Fox [85] with the heap of Thorup mentioned earlier. Raman [291]

further improved these results to obtain a bound of O.min.lg

1=4C

C; lg

1=3C

n//,

for any ﬁxed > 0.

7 Quicksort

The quicksort algorithm has a worst-case running time of ‚.n 2 / on an input array

of n numbers. Despite this slow worst-case running time, quicksort is often the best

practical choice for sorting because it is remarkably efﬁcient on the average: its

expected running time is ‚.n lg n/, and the constant factors hidden in the ‚.n lg n/

notation are quite small. It also has the advantage of sorting in place (see page 17),

and it works well even in virtual-memory environments.

Section 7.1 describes the algorithm and an important subroutine used by quick-

sort for partitioning. Because the behavior of quicksort is complex, we start with

an intuitive discussion of its performance in Section 7.2 and postpone its precise

analysis to the end of the chapter. Section 7.3 presents a version of quicksort that

uses random sampling. This algorithm has a good expected running time, and no

particular input elicits its worst-case behavior. Section 7.4 analyzes the random-

ized algorithm, showing that it runs in ‚.n 2 / time in the worst case and, assuming

distinct elements, in expected O.n lg n/ time.

7.1 Description of quicksort

Quicksort, like merge sort, applies the divide-and-conquer paradigm introduced

in Section 2.3.1. Here is the three-step divide-and-conquer process for sorting a

typical subarray AŒp : : r:

Divide: Partition (rearrange) the array AŒp : : r into two (possibly empty) subar-

rays AŒp : : q 1 and AŒq C 1 : : r such that each element of AŒp : : q 1 is

less than or equal to AŒq, which is, in turn, less than or equal to each element

of AŒq C 1 : : r. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays AŒp : : q 1 and AŒq C 1 : : r by recursive calls

to quicksort.

7.1 Description of quicksort 171

Combine: Because the subarrays are already sorted, no work is needed to combine

them: the entire array AŒp : : r is now sorted.

The following procedure implements quicksort:

QUICKSORT.A; p; r/

1 if p < r

2 q D PARTITION.A; p; r/

3 QUICKSORT.A; p; q 1/

4 QUICKSORT.A; q C 1; r/

To sort an entire array A, the initial call is QUICKSORT.A; 1; A:length/.

Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subar-

ray AŒp : : r in place.

PARTITION.A; p; r/

1 x D AŒr

2 i D p 1

3 for j D p to r 1

4 if AŒj x

5 i D i C 1

6 exchange AŒi with AŒj

7 exchange AŒi C 1 with AŒr

8 return i C 1

Figure 7.1 shows how PARTITION works on an 8-element array. PARTITION

always selects an element x D AŒr as a pivot element around which to partition the

subarray AŒp : : r. As the procedure runs, it partitions the array into four (possibly

empty) regions. At the start of each iteration of the for loop in lines 3–6, the regions

satisfy certain properties, shown in Figure 7.2. We state these properties as a loop

invariant:

At the beginning of each iteration of the loop of lines 3–6, for any array

index k,

1. If p k i, then AŒk x.

2. If i C 1 k j 1, then AŒk > x.

3. If k D r, then AŒk D x.

172 Chapter 7 Quicksort

2 8 7 1 3 5 6 4

p,j r i

(a)

2 8 7 1 3 5 6 4

p,i r j

(b)

2 8 7 1 3 5 6 4

p,i r j

(c)

2 8 7 1 3 5 6 4

p,i r j

(d)

2 8 7 1 3 5 6 4

p r j

(e)

i

2 8 7 1 3 5 6 4

p r j

(f)

i

2 8 7 1 3 5 6 4

p r j

(g)

i

2 8 7 1 3 5 6 4

p r

(h)

i

2 8 7 1 3 5 6 4

p r

(i)

i

Figure 7.1 The operation of PARTITION on a sample array. Array entry AŒr becomes the pivot

element x. Lightly shaded array elements are all in the ﬁrst partition with values no greater than x.

Heavily shaded elements are in the second partition with values greater than x. The unshaded el-

ements have not yet been put in one of the ﬁrst two partitions, and the ﬁnal white element is the

pivot x. (a) The initial array and variable settings. None of the elements have been placed in either

of the ﬁrst two partitions. (b) The value 2 is “swapped with itself” and put in the partition of smaller

values. (c)–(d) The values 8 and 7 are added to the partition of larger values. (e) The values 1 and 8

are swapped, and the smaller partition grows. (f) The values 3 and 7 are swapped, and the smaller

partition grows. (g)–(h) The larger partition grows to include 5 and 6, and the loop terminates. (i) In

lines 7–8, the pivot element is swapped so that it lies between the two partitions.

The indices between j and r 1 are not covered by any of the three cases, and the

values in these entries have no particular relationship to the pivot x.

We need to show that this loop invariant is true prior to the ﬁrst iteration, that

each iteration of the loop maintains the invariant, and that the invariant provides a

useful property to show correctness when the loop terminates.

7.1 Description of quicksort 173

≤ x > x unrestricted

x

p i j r

Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray AŒp : : r. The

values in AŒp : : i are all less than or equal to x, the values in AŒi C 1 : : j 1 are all greater than x,

and AŒr D x. The subarray AŒj : : r 1 can take on any values.

Initialization: Prior to the ﬁrst iteration of the loop, i D p 1 and j D p. Be-

cause no values lie between p and i and no values lie between i C 1 and j 1,

the ﬁrst two conditions of the loop invariant are trivially satisﬁed. The assign-

ment in line 1 satisﬁes the third condition.

Maintenance: As Figure 7.3 shows, we consider two cases, depending on the

outcome of the test in line 4. Figure 7.3(a) shows what happens when AŒj > x;

the only action in the loop is to increment j . After j is incremented, condition 2

holds for AŒj 1 and all other entries remain unchanged. Figure 7.3(b) shows

what happens when AŒj x; the loop increments i, swaps AŒi and AŒj ,

and then increments j . Because of the swap, we now have that AŒi x, and

condition 1 is satisﬁed. Similarly, we also have that AŒj 1 > x, since the

item that was swapped into AŒj 1 is, by the loop invariant, greater than x.

Termination: At termination, j D r. Therefore, every entry in the array is in one

of the three sets described by the invariant, and we have partitioned the values

in the array into three sets: those less than or equal to x, those greater than x,

and a singleton set containing x.

The ﬁnal two lines of PARTITION ﬁnish up by swapping the pivot element with

the leftmost element greater than x, thereby moving the pivot into its correct place

in the partitioned array, and then returning the pivot’s new index. The output of

PARTITION now satisﬁes the speciﬁcations given for the divide step. In fact, it

satisﬁes a slightly stronger condition: after line 2 of QUICKSORT, AŒq is strictly

less than every element of AŒq C 1 : : r.

The running time of PARTITION on the subarray AŒp : : r is ‚.n/, where

n D r p C 1 (see Exercise 7.1-3).

Exercises

7.1-1

Using Figure 7.1 as a model, illustrate the operation of PARTITION on the array

A D h13; 19; 9; 5; 12; 8; 7; 4; 21; 2; 6; 11i.

174 Chapter 7 Quicksort

≤ x > x

x

p i j r

>x (a)

≤ x > x

x

p i j r

≤ x > x

x

p i j r

≤ x (b)

≤ x > x

x

p i j r

Figure 7.3 The two cases for one iteration of procedure PARTITION. (a) If AŒj > x, the only

action is to increment j , which maintains the loop invariant. (b) If AŒj x, index i is incremented,

AŒi and AŒj are swapped, and then j is incremented. Again, the loop invariant is maintained.

7.1-2

What value of q does PARTITION return when all elements in the array AŒp : : r

have the same value? Modify PARTITION so that q D b.p C r/=2c when all

elements in the array AŒp : : r have the same value.

7.1-3

Give a brief argument that the running time of PARTITION on a subarray of size n

is ‚.n/.

7.1-4

How would you modify QUICKSORT to sort into nonincreasing order?

7.2 Performance of quicksort

The running time of quicksort depends on whether the partitioning is balanced or

unbalanced, which in turn depends on which elements are used for partitioning.

If the partitioning is balanced, the algorithm runs asymptotically as fast as merge

7.2 Performance of quicksort 175

sort. If the partitioning is unbalanced, however, it can run asymptotically as slowly

as insertion sort. In this section, we shall informally investigate how quicksort

performs under the assumptions of balanced versus unbalanced partitioning.

Worst-case partitioning

The worst-case behavior for quicksort occurs when the partitioning routine pro-

duces one subproblem with n 1 elements and one with 0 elements. (We prove

this claim in Section 7.4.1.) Let us assume that this unbalanced partitioning arises

in each recursive call. The partitioning costs ‚.n/ time. Since the recursive call

on an array of size 0 just returns, T .0/ D ‚.1/, and the recurrence for the running

time is

T .n/ D T .n 1/ C T .0/ C ‚.n/

D T .n 1/ C ‚.n/ :

Intuitively, if we sum the costs incurred at each level of the recursion, we get

an arithmetic series (equation (A.2)), which evaluates to ‚.n 2 /. Indeed, it is

straightforward to use the substitution method to prove that the recurrence T .n/ D

T .n 1/ C ‚.n/ has the solution T .n/ D ‚.n 2 /. (See Exercise 7.2-1.)

Thus, if the partitioning is maximally unbalanced at every recursive level of the

algorithm, the running time is ‚.n 2 /. Therefore the worst-case running time of

quicksort is no better than that of insertion sort. Moreover, the ‚.n 2 / running time

occurs when the input array is already completely sorted—a common situation in

which insertion sort runs in O.n/ time.

Best-case partitioning

In the most even possible split, PARTITION produces two subproblems, each of

size no more than n=2, since one is of size bn=2c and one of size dn=2e1. In this

case, quicksort runs much faster. The recurrence for the running time is then

T .n/ D 2T .n=2/ C ‚.n/ ;

where we tolerate the sloppiness from ignoring the ﬂoor and ceiling and from sub-

tracting 1. By case 2 of the master theorem (Theorem 4.1), this recurrence has the

solution T .n/ D ‚.n lg n/. By equally balancing the two sides of the partition at

every level of the recursion, we get an asymptotically faster algorithm.

Balanced partitioning

The average-case running time of quicksort is much closer to the best case than to

the worst case, as the analyses in Section 7.4 will show. The key to understand-

176 Chapter 7 Quicksort

n

cn

cn

cn

cn

cn

cn

1

1

O.n lg n/

log 10 n

log 10=9 n

1

10

n

9

10

n

1

100

n

9

100

n

9

100

n

81

100

n

81

1000

n

729

1000

n

Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split,

yielding a running time of O.n lg n/. Nodes show subproblem sizes, with per-level costs on the right.

The per-level costs include the constant c implicit in the ‚.n/ term.

ing why is to understand how the balance of the partitioning is reﬂected in the

recurrence that describes the running time.

Suppose, for example, that the partitioning algorithm always produces a 9-to-1

proportional split, which at ﬁrst blush seems quite unbalanced. We then obtain the

recurrence

T .n/ D T .9n=10/ C T .n=10/ C cn ;

on the running time of quicksort, where we have explicitly included the constant c

hidden in the ‚.n/ term. Figure 7.4 shows the recursion tree for this recurrence.

Notice that every level of the tree has cost cn, until the recursion reaches a bound-

ary condition at depth log 10 n D ‚.lg n/, and then the levels have cost at most cn.

The recursion terminates at depth log 10=9 n D ‚.lg n/. The total cost of quick-

sort is therefore O.n lg n/. Thus, with a 9-to-1 proportional split at every level of

recursion, which intuitively seems quite unbalanced, quicksort runs in O.n lg n/

time—asymptotically the same as if the split were right down the middle. Indeed,

even a 99-to-1 split yields an O.n lg n/ running time. In fact, any split of constant

proportionality yields a recursion tree of depth ‚.lg n/, where the cost at each level

is O.n/. The running time is therefore O.n lg n/ whenever the split has constant

proportionality.

7.2 Performance of quicksort 177

n

0 n–1

(n–1)/2 – 1 (n–1)/2

n

(n–1)/2

(a) (b)

(n–1)/2

Θ (n) Θ (n)

Figure 7.5 (a) Two levels of a recursion tree for quicksort. The partitioning at the root costs n

and produces a “bad” split: two subarrays of sizes 0 and n 1. The partitioning of the subarray of

size n 1 costs n 1 and produces a “good” split: subarrays of size .n 1/=2 1 and .n 1/=2.

(b) A single level of a recursion tree that is very well balanced. In both parts, the partitioning cost for

the subproblems shown with elliptical shading is ‚.n/. Yet the subproblems remaining to be solved

in (a), shown with square shading, are no larger than the corresponding subproblems remaining to be

solved in (b).

Intuition for the average case

To develop a clear notion of the randomized behavior of quicksort, we must make

an assumption about how frequently we expect to encounter the various inputs.

The behavior of quicksort depends on the relative ordering of the values in the

array elements given as the input, and not by the particular values in the array. As

in our probabilistic analysis of the hiring problem in Section 5.2, we will assume

for now that all permutations of the input numbers are equally likely.

When we run quicksort on a random input array, the partitioning is highly un-

likely to happen in the same way at every level, as our informal analysis has as-

sumed. We expect that some of the splits will be reasonably well balanced and

that some will be fairly unbalanced. For example, Exercise 7.2-6 asks you to show

that about 80 percent of the time PARTITION produces a split that is more balanced

than 9 to 1, and about 20 percent of the time it produces a split that is less balanced

than 9 to 1.

In the average case, PARTITION produces a mix of “good” and “bad” splits. In a

recursion tree for an average-case execution of PARTITION, the good and bad splits

are distributed randomly throughout the tree. Suppose, for the sake of intuition,

that the good and bad splits alternate levels in the tree, and that the good splits

are best-case splits and the bad splits are worst-case splits. Figure 7.5(a) shows

the splits at two consecutive levels in the recursion tree. At the root of the tree,

the cost is n for partitioning, and the subarrays produced have sizes n 1 and 0:

the worst case. At the next level, the subarray of size n 1 undergoes best-case

partitioning into subarrays of size .n 1/=2 1 and .n 1/=2. Let’s assume that

the boundary-condition cost is 1 for the subarray of size 0.

178 Chapter 7 Quicksort

The combination of the bad split followed by the good split produces three sub-

arrays of sizes 0, .n 1/=2 1, and .n 1/=2 at a combined partitioning cost

of ‚.n/ C ‚.n 1/ D ‚.n/. Certainly, this situation is no worse than that in

Figure 7.5(b), namely a single level of partitioning that produces two subarrays of

size .n 1/=2, at a cost of ‚.n/. Yet this latter situation is balanced! Intuitively,

the ‚.n 1/ cost of the bad split can be absorbed into the ‚.n/ cost of the good

split, and the resulting split is good. Thus, the running time of quicksort, when lev-

els alternate between good and bad splits, is like the running time for good splits

alone: still O.n lg n/, but with a slightly larger constant hidden by the O-notation.

We shall give a rigorous analysis of the expected running time of a randomized

version of quicksort in Section 7.4.2.

Exercises

7.2-1

Use the substitution method to prove that the recurrence T .n/ D T .n 1/ C ‚.n/

has the solution T .n/ D ‚.n 2 /, as claimed at the beginning of Section 7.2.

7.2-2

What is the running time of QUICKSORT when all elements of array A have the

same value?

7.2-3

Show that the running time of QUICKSORT is ‚.n 2 / when the array A contains

distinct elements and is sorted in decreasing order.

7.2-4

Banks often record transactions on an account in order of the times of the transac-

tions, but many people like to receive their bank statements with checks listed in

order by check number. People usually write checks in order by check number, and

merchants usually cash them with reasonable dispatch. The problem of converting

time-of-transaction ordering to check-number ordering is therefore the problem of

sorting almost-sorted input. Argue that the procedure INSERTION-SORT would

tend to beat the procedure QUICKSORT on this problem.

7.2-5

Suppose that the splits at every level of quicksort are in the proportion 1 ˛ to ˛,

where 0 < ˛ 1=2 is a constant. Show that the minimum depth of a leaf in the re-

cursion tree is approximately lg n= lg ˛ and the maximum depth is approximately

lg n= lg.1 ˛/. (Don’t worry about integer round-off.)

7.3 A randomized version of quicksort 179

7.2-6 ?

Argue that for any constant 0 < ˛ 1=2, the probability is approximately 1 2˛

that on a random input array, PARTITION produces a split more balanced than 1˛

to ˛.

7.3 A randomized version of quicksort

In exploring the average-case behavior of quicksort, we have made an assumption

that all permutations of the input numbers are equally likely. In an engineering

situation, however, we cannot always expect this assumption to hold. (See Exer-

cise 7.2-4.) As we saw in Section 5.3, we can sometimes add randomization to an

algorithm in order to obtain good expected performance over all inputs. Many peo-

ple regard the resulting randomized version of quicksort as the sorting algorithm

of choice for large enough inputs.

In Section 5.3, we randomized our algorithm by explicitly permuting the in-

put. We could do so for quicksort also, but a different randomization technique,

called random sampling, yields a simpler analysis. Instead of always using AŒr

as the pivot, we will select a randomly chosen element from the subarray AŒp : : r.

We do so by ﬁrst exchanging element AŒr with an element chosen at random

from AŒp : : r. By randomly sampling the range p; : : : ; r, we ensure that the pivot

element x D AŒr is equally likely to be any of the r p C 1 elements in the

subarray. Because we randomly choose the pivot element, we expect the split of

the input array to be reasonably well balanced on average.

The changes to PARTITION and QUICKSORT are small. In the new partition

procedure, we simply implement the swap before actually partitioning:

RANDOMIZED-PARTITION.A; p; r/

1 i D RANDOM.p; r/

2 exchange AŒr with AŒi

3 return PARTITION.A; p; r/

The new quicksort calls RANDOMIZED-PARTITION in place of PARTITION:

RANDOMIZED-QUICKSORT.A; p; r/

1 if p < r

2 q D RANDOMIZED-PARTITION.A; p; r/

3 RANDOMIZED-QUICKSORT.A; p; q 1/

4 RANDOMIZED-QUICKSORT.A; q C 1; r/

We analyze this algorithm in the next section.

180 Chapter 7 Quicksort

Exercises

7.3-1

Why do we analyze the expected running time of a randomized algorithm and not

its worst-case running time?

7.3-2

When RANDOMIZED-QUICKSORT runs, how many calls are made to the random-

number generator RANDOM in the worst case? How about in the best case? Give

your answer in terms of ‚-notation.

7.4 Analysis of quicksort

Section 7.2 gave some intuition for the worst-case behavior of quicksort and for

why we expect it to run quickly. In this section, we analyze the behavior of quick-

sort more rigorously. We begin with a worst-case analysis, which applies to either

QUICKSORT or RANDOMIZED-QUICKSORT, and conclude with an analysis of the

expected running time of RANDOMIZED-QUICKSORT.

7.4.1 Worst-case analysis

We saw in Section 7.2 that a worst-case split at every level of recursion in quicksort

produces a ‚.n 2 / running time, which, intuitively, is the worst-case running time

of the algorithm. We now prove this assertion.

Using the substitution method (see Section 4.3), we can show that the running

time of quicksort is O.n 2 /. Let T .n/ be the worst-case time for the procedure

QUICKSORT on an input of size n. We have the recurrence

T .n/ D max

0qn1

.T .q/ C T .n q 1// C ‚.n/ ; (7.1)

where the parameter q ranges from 0 to n 1 because the procedure PARTITION

produces two subproblems with total size n 1. We guess that T .n/ cn 2

for

some constant c. Substituting this guess into recurrence (7.1), we obtain

T .n/ max

0qn1

.cq

2

C c.n q 1/

2

/ C ‚.n/

D c max

0qn1

.q

2

C .n q 1/

2

/ C ‚.n/ :

The expression q 2 C .n q 1/ 2

achieves a maximum over the parameter’s

range 0 q n 1 at either endpoint. To verify this claim, note that the second

derivative of the expression with respect to q is positive (see Exercise 7.4-3). This

7.4 Analysis of quicksort 181

observation gives us the bound max 0qn1 .q 2 C .n q 1/ 2 / .n 1/ 2 D

n 2 2n C 1. Continuing with our bounding of T .n/, we obtain

T .n/ cn

2

c.2n 1/ C ‚.n/

cn

2

;

since we can pick the constant c large enough so that the c.2n 1/ term dom-

inates the ‚.n/ term. Thus, T .n/ D O.n 2 /. We saw in Section 7.2 a speciﬁc

case in which quicksort takes .n 2 / time: when partitioning is unbalanced. Al-

ternatively, Exercise 7.4-1 asks you to show that recurrence (7.1) has a solution of

T .n/ D .n 2 /. Thus, the (worst-case) running time of quicksort is ‚.n 2 /.

7.4.2 Expected running time

We have already seen the intuition behind why the expected running time of

RANDOMIZED-QUICKSORT is O.n lg n/: if, in each level of recursion, the split

induced by RANDOMIZED-PARTITION puts any constant fraction of the elements

on one side of the partition, then the recursion tree has depth ‚.lg n/, and O.n/

work is performed at each level. Even if we add a few new levels with the most un-

balanced split possible between these levels, the total time remains O.n lg n/. We

can analyze the expected running time of RANDOMIZED-QUICKSORT precisely

by ﬁrst understanding how the partitioning procedure operates and then using this

understanding to derive an O.n lg n/ bound on the expected running time. This

upper bound on the expected running time, combined with the ‚.n lg n/ best-case

bound we saw in Section 7.2, yields a ‚.n lg n/ expected running time. We assume

throughout that the values of the elements being sorted are distinct.

Running time and comparisons

The QUICKSORT and RANDOMIZED-QUICKSORT procedures differ only in how

they select pivot elements; they are the same in all other respects. We can therefore

couch our analysis of RANDOMIZED-QUICKSORT by discussing the QUICKSORT

and PARTITION procedures, but with the assumption that pivot elements are se-

lected randomly from the subarray passed to RANDOMIZED-PARTITION.

The running time of QUICKSORT is dominated by the time spent in the PARTI-

TION procedure. Each time the PARTITION procedure is called, it selects a pivot

element, and this element is never included in any future recursive calls to QUICK-

SORT and PARTITION. Thus, there can be at most n calls to PARTITION over the

entire execution of the quicksort algorithm. One call to PARTITION takes O.1/

time plus an amount of time that is proportional to the number of iterations of the

for loop in lines 3–6. Each iteration of this for loop performs a comparison in

line 4, comparing the pivot element to another element of the array A. Therefore,

182 Chapter 7 Quicksort

if we can count the total number of times that line 4 is executed, we can bound the

total time spent in the for loop during the entire execution of QUICKSORT.

Lemma 7.1

Let X be the number of comparisons performed in line 4 of PARTITION over the

entire execution of QUICKSORT on an n-element array. Then the running time of

QUICKSORT is O.n C X/.

Proof By the discussion above, the algorithm makes at most n calls to PARTI-

TION, each of which does a constant amount of work and then executes the for

loop some number of times. Each iteration of the for loop executes line 4.

Our goal, therefore, is to compute X, the total number of comparisons performed

in all calls to PARTITION. We will not attempt to analyze how many comparisons

are made in each call to PARTITION. Rather, we will derive an overall bound on the

total number of comparisons. To do so, we must understand when the algorithm

compares two elements of the array and when it does not. For ease of analysis, we

rename the elements of the array A as ´ 1 ; ´ 2 ; : : : ; ´ n , with ´ i being the ith smallest

element. We also deﬁne the set Z ij D f´ i ; ´ iC1 ; : : : ; ´ j g to be the set of elements

between ´ i and ´ j , inclusive.

When does the algorithm compare ´ i and ´ j ? To answer this question, we ﬁrst

observe that each pair of elements is compared at most once. Why? Elements

are compared only to the pivot element and, after a particular call of PARTITION

ﬁnishes, the pivot element used in that call is never again compared to any other

elements.

Our analysis uses indicator random variables (see Section 5.2). We deﬁne

X ij D I f´ i is compared to ´ j g ;

where we are considering whether the comparison takes place at any time during

the execution of the algorithm, not just during one iteration or one call of PARTI-

TION. Since each pair is compared at most once, we can easily characterize the

total number of comparisons performed by the algorithm:

X D

n1 X

iD1

n X

j DiC1

X ij :

Taking expectations of both sides, and then using linearity of expectation and

Lemma 5.1, we obtain

E ŒX D E

"

n1 X

iD1

n X

j DiC1

X ij

#

7.4 Analysis of quicksort 183

D

n1 X

iD1

n X

j DiC1

E ŒX ij

D

n1 X

iD1

n X

j DiC1

Pr f´ i is compared to ´ j g : (7.2)

It remains to compute Pr f´ i is compared to ´ j g. Our analysis assumes that the

RANDOMIZED-PARTITION procedure chooses each pivot randomly and indepen-

dently.

Let us think about when two items are not compared. Consider an input to

quicksort of the numbers 1 through 10 (in any order), and suppose that the ﬁrst

pivot element is 7. Then the ﬁrst call to PARTITION separates the numbers into two

sets: f1; 2; 3; 4; 5; 6g and f8; 9; 10g. In doing so, the pivot element 7 is compared

to all other elements, but no number from the ﬁrst set (e.g., 2) is or ever will be

compared to any number from the second set (e.g., 9).

In general, because we assume that element values are distinct, once a pivot x

is chosen with ´ i < x < ´ j , we know that ´ i and ´ j cannot be compared at any

subsequent time. If, on the other hand, ´ i is chosen as a pivot before any other item

in Z ij , then ´ i will be compared to each item in Z ij , except for itself. Similarly,

if ´ j is chosen as a pivot before any other item in Z ij , then ´ j will be compared to

each item in Z ij , except for itself. In our example, the values 7 and 9 are compared

because 7 is the ﬁrst item from Z 7;9 to be chosen as a pivot. In contrast, 2 and 9 will

never be compared because the ﬁrst pivot element chosen from Z 2;9 is 7. Thus, ´ i

and ´ j are compared if and only if the ﬁrst element to be chosen as a pivot from Z ij

is either ´ i or ´ j .

We now compute the probability that this event occurs. Prior to the point at

which an element from Z ij has been chosen as a pivot, the whole set Z ij is together

in the same partition. Therefore, any element of Z ij is equally likely to be the ﬁrst

one chosen as a pivot. Because the set Z ij has j iC1 elements, and because pivots

are chosen randomly and independently, the probability that any given element is

the ﬁrst one chosen as a pivot is 1=.j i C 1/. Thus, we have

Pr f´ i is compared to ´ j g D Pr f´ i or ´ j is ﬁrst pivot chosen from Z ij g

D Pr f´ i is ﬁrst pivot chosen from Z ij g

C Pr f´ j is ﬁrst pivot chosen from Z ij g

D

1

j i C 1

C

1

j i C 1

D

2

j i C 1

: (7.3)

184 Chapter 7 Quicksort

The second line follows because the two events are mutually exclusive. Combining

equations (7.2) and (7.3), we get that

E ŒX D

n1 X

iD1

n X

j DiC1

2

j i C 1

:

We can evaluate this sum using a change of variables (k D j i) and the bound

on the harmonic series in equation (A.7):

E ŒX D

n1 X

iD1

n X

j DiC1

2

j i C 1

D

n1 X

iD1

ni X

kD1

2

k C 1

<

n1 X

iD1

n X

kD1

2

k

D

n1 X

iD1

O.lg n/

D O.n lg n/ : (7.4)

Thus we conclude that, using RANDOMIZED-PARTITION, the expected running

time of quicksort is O.n lg n/ when element values are distinct.

Exercises

7.4-1

Show that in the recurrence

T .n/ D max

0qn1

.T .q/ C T .n q 1// C ‚.n/ ;

T .n/ D .n 2 /.

7.4-2

Show that quicksort’s best-case running time is .n lg n/.

7.4-3

Show that the expression q 2 C .n q 1/ 2

achieves a maximum over q D

0; 1; : : : ; n 1 when q D 0 or q D n 1.

7.4-4

Show that RANDOMIZED-QUICKSORT’s expected running time is .n lg n/.

Problems for Chapter 7 185

7.4-5

We can improve the running time of quicksort in practice by taking advantage of the

fast running time of insertion sort when its input is “nearly” sorted. Upon calling

quicksort on a subarray with fewer than k elements, let it simply return without

sorting the subarray. After the top-level call to quicksort returns, run insertion sort

on the entire array to ﬁnish the sorting process. Argue that this sorting algorithm

runs in O.nk C n lg.n=k// expected time. How should we pick k, both in theory

and in practice?

7.4-6 ?

Consider modifying the PARTITION procedure by randomly picking three elements

from array A and partitioning about their median (the middle value of the three

elements). Approximate the probability of getting at worst an ˛-to-.1 ˛/ split, as

a function of ˛ in the range 0 < ˛ < 1.

Problems

7-1 Hoare partition correctness

The version of PARTITION given in this chapter is not the original partitioning

algorithm. Here is the original partition algorithm, which is due to C. A. R. Hoare:

HOARE-PARTITION.A; p; r/

1 x D AŒp

2 i D p 1

3 j D r C 1

4 while TRUE

5 repeat

6 j D j 1

7 until AŒj x

8 repeat

9 i D i C 1

10 until AŒi x

11 if i < j

12 exchange AŒi with AŒj

13 else return j

a. Demonstrate the operation of HOARE-PARTITION on the array A D h13; 19; 9;

5; 12; 8; 7; 4; 11; 2; 6; 21i, showing the values of the array and auxiliary values

after each iteration of the while loop in lines 4–13.

186 Chapter 7 Quicksort

The next three questions ask you to give a careful argument that the procedure

HOARE-PARTITION is correct. Assuming that the subarray AŒp : : r contains at

least two elements, prove the following:

b. The indices i and j are such that we never access an element of A outside the

subarray AŒp : : r.

c. When HOARE-PARTITION terminates, it returns a value j such that p j < r.

d. Every element of AŒp : : j is less than or equal to every element of AŒj C1 : : r

when HOARE-PARTITION terminates.

The PARTITION procedure in Section 7.1 separates the pivot value (originally

in AŒr) from the two partitions it forms. The HOARE-PARTITION procedure, on

the other hand, always places the pivot value (originally in AŒp) into one of the

two partitions AŒp : : j and AŒj C 1 : : r. Since p j < r, this split is always

nontrivial.

e. Rewrite the QUICKSORT procedure to use HOARE-PARTITION.

7-2 Quicksort with equal element values

The analysis of the expected running time of randomized quicksort in Section 7.4.2

assumes that all element values are distinct. In this problem, we examine what

happens when they are not.

a. Suppose that all element values are equal. What would be randomized quick-

sort’s running time in this case?

b. The PARTITION procedure returns an index q such that each element of

AŒp : : q 1 is less than or equal to AŒq and each element of AŒq C 1 : : r

is greater than AŒq. Modify the PARTITION procedure to produce a procedure

PARTITION

0

.A; p; r/, which permutes the elements of AŒp : : r and returns two

indices q and t, where p q t r, such that

all elements of AŒq : : t are equal,

each element of AŒp : : q 1 is less than AŒq, and

each element of AŒt C 1 : : r is greater than AŒq.

Like PARTITION, your PARTITION

0

procedure should take ‚.r p/ time.

c. Modify the RANDOMIZED-QUICKSORT procedure to call PARTITION

0

, and

name the new procedure RANDOMIZED-QUICKSORT

0

. Then modify the

QUICKSORT procedure to produce a procedure QUICKSORT

0

.p; r/ that calls

Problems for Chapter 7 187

RANDOMIZED-PARTITION

0

and recurses only on partitions of elements not

known to be equal to each other.

d. Using QUICKSORT

0

, how would you adjust the analysis in Section 7.4.2 to

avoid the assumption that all elements are distinct?

7-3 Alternative quicksort analysis

An alternative analysis of the running time of randomized quicksort focuses on

the expected running time of each individual recursive call to RANDOMIZED-

QUICKSORT, rather than on the number of comparisons performed.

a. Argue that, given an array of size n, the probability that any particular element

is chosen as the pivot is 1=n. Use this to deﬁne indicator random variables

X i D I fith smallest element is chosen as the pivotg. What is E ŒX i ?

b. Let T .n/ be a random variable denoting the running time of quicksort on an

array of size n. Argue that

E ŒT .n/ D E

"

n X

qD1

X q .T .q 1/ C T .n q/ C ‚.n//

#

: (7.5)

c. Show that we can rewrite equation (7.5) as

E ŒT .n/ D

2

n

n1 X

qD2

E ŒT .q/ C ‚.n/ : (7.6)

d. Show that

n1 X

kD2

k lg k

1

2

n

2

lg n

1

8

n

2

: (7.7)

(Hint: Split the summation into two parts, one for k D 2; 3; : : : ; dn=2e 1 and

one for k D dn=2e ; : : : ; n 1.)

e. Using the bound from equation (7.7), show that the recurrence in equation (7.6)

has the solution E ŒT .n/ D ‚.n lg n/. (Hint: Show, by substitution, that

E ŒT .n/ an lg n for sufﬁciently large n and for some positive constant a.)

188 Chapter 7 Quicksort

7-4 Stack depth for quicksort

The QUICKSORT algorithm of Section 7.1 contains two recursive calls to itself.

After QUICKSORT calls PARTITION, it recursively sorts the left subarray and then

it recursively sorts the right subarray. The second recursive call in QUICKSORT

is not really necessary; we can avoid it by using an iterative control structure.

This technique, called tail recursion, is provided automatically by good compilers.

Consider the following version of quicksort, which simulates tail recursion:

TAIL-RECURSIVE-QUICKSORT.A; p; r/

1 while p < r

2 // Partition and sort left subarray.

3 q D PARTITION.A; p; r/

4 TAIL-RECURSIVE-QUICKSORT.A; p; q 1/

5 p D q C 1

a. Argue that TAIL-RECURSIVE-QUICKSORT.A; 1; A:length/ correctly sorts the

array A.

Compilers usually execute recursive procedures by using a stack that contains per-

tinent information, including the parameter values, for each recursive call. The

information for the most recent call is at the top of the stack, and the information

for the initial call is at the bottom. Upon calling a procedure, its information is

pushed onto the stack; when it terminates, its information is popped. Since we

assume that array parameters are represented by pointers, the information for each

procedure call on the stack requires O.1/ stack space. The stack depth is the max-

imum amount of stack space used at any time during a computation.

b. Describe a scenario in which TAIL-RECURSIVE-QUICKSORT’s stack depth is

‚.n/ on an n-element input array.

c. Modify the code for TAIL-RECURSIVE-QUICKSORT so that the worst-case

stack depth is ‚.lg n/. Maintain the O.n lg n/ expected running time of the

algorithm.

7-5 Median-of-3 partition

One way to improve the RANDOMIZED-QUICKSORT procedure is to partition

around a pivot that is chosen more carefully than by picking a random element

from the subarray. One common approach is the median-of-3 method: choose

the pivot as the median (middle element) of a set of 3 elements randomly selected

from the subarray. (See Exercise 7.4-6.) For this problem, let us assume that the

elements in the input array AŒ1 : : n are distinct and that n 3. We denote the

Problems for Chapter 7 189

sorted output array by A 0 Œ1 : : n. Using the median-of-3 method to choose the

pivot element x, deﬁne p i D Pr fx D A 0 Œig.

a. Give an exact formula for p i as a function of n and i for i D 2; 3; : : : ; n 1.

(Note that p 1 D p n D 0.)

b. By what amount have we increased the likelihood of choosing the pivot as

x D A 0 Œb.n C 1/=2c, the median of AŒ1 : : n, compared with the ordinary

implementation? Assume that n ! 1, and give the limiting ratio of these

probabilities.

c. If we deﬁne a “good” split to mean choosing the pivot as x D A 0 Œi, where

n=3 i 2n=3, by what amount have we increased the likelihood of getting

a good split compared with the ordinary implementation? (Hint: Approximate

the sum by an integral.)

d. Argue that in the .n lg n/ running time of quicksort, the median-of-3 method

affects only the constant factor.

7-6 Fuzzy sorting of intervals

Consider a sorting problem in which we do not know the numbers exactly. In-

stead, for each number, we know an interval on the real line to which it belongs.

That is, we are given n closed intervals of the form Œa i ; b i , where a i b i . We

wish to fuzzy-sort these intervals, i.e., to produce a permutation hi 1 ; i 2 ; : : : ; i n i of

the intervals such that for j D 1; 2; : : : ; n, there exist c j 2 Œa i j ; b i j satisfying

c 1 c 2 c n .

a. Design a randomized algorithm for fuzzy-sorting n intervals. Your algorithm

should have the general structure of an algorithm that quicksorts the left end-

points (the a i values), but it should take advantage of overlapping intervals to

improve the running time. (As the intervals overlap more and more, the prob-

lem of fuzzy-sorting the intervals becomes progressively easier. Your algorithm

should take advantage of such overlapping, to the extent that it exists.)

b. Argue that your algorithm runs in expected time ‚.n lg n/ in general, but runs

in expected time ‚.n/ when all of the intervals overlap (i.e., when there exists a

value x such that x 2 Œa i ; b i for all i). Your algorithm should not be checking

for this case explicitly; rather, its performance should naturally improve as the

amount of overlap increases.

190 Chapter 7 Quicksort

Chapter notes

The quicksort procedure was invented by Hoare [170]; Hoare’s version appears in

Problem 7-1. The PARTITION procedure given in Section 7.1 is due to N. Lomuto.

The analysis in Section 7.4 is due to Avrim Blum. Sedgewick [305] and Bent-

ley [43] provide a good reference on the details of implementation and how they

matter.

McIlroy [248] showed how to engineer a “killer adversary” that produces an

array on which virtually any implementation of quicksort takes ‚.n 2 / time. If the

implementation is randomized, the adversary produces the array after seeing the

random choices of the quicksort algorithm.

8 Sorting in Linear Time

We have now introduced several algorithms that can sort n numbers in O.n lg n/

time. Merge sort and heapsort achieve this upper bound in the worst case; quicksort

achieves it on average. Moreover, for each of these algorithms, we can produce a

sequence of n input numbers that causes the algorithm to run in .n lg n/ time.

These algorithms share an interesting property: the sorted order they determine

is based only on comparisons between the input elements. We call such sorting

algorithms comparison sorts. All the sorting algorithms introduced thus far are

comparison sorts.

In Section 8.1, we shall prove that any comparison sort must make .n lg n/

comparisons in the worst case to sort n elements. Thus, merge sort and heapsort

are asymptotically optimal, and no comparison sort exists that is faster by more

than a constant factor.

Sections 8.2, 8.3, and 8.4 examine three sorting algorithms—counting sort, radix

sort, and bucket sort—that run in linear time. Of course, these algorithms use

operations other than comparisons to determine the sorted order. Consequently,

the .n lg n/ lower bound does not apply to them.

8.1 Lower bounds for sorting

In a comparison sort, we use only comparisons between elements to gain order

information about an input sequence ha 1 ; a 2 ; : : : ; a n i. That is, given two elements

a i and a j , we perform one of the tests a i < a j , a i a j , a i D a j , a i a j , or

a i > a j to determine their relative order. We may not inspect the values of the

elements or gain order information about them in any other way.

In this section, we assume without loss of generality that all the input elements

are distinct. Given this assumption, comparisons of the form a i D a j are useless,

so we can assume that no comparisons of this form are made. We also note that

the comparisons a i a j , a i a j , a i > a j , and a i < a j are all equivalent in that

192 Chapter 8 Sorting in Linear Time

≤ >

≤ >

1:2

2:3 1:3

〈 1,2,3 〉 1:3 〈 2,1,3 〉 2:3

〈 1,3,2 〉 〈 3,1,2 〉 〈 3,2,1 〉

≤ >

≤ >

≤ >

〈 2,3,1 〉

Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node an-

notated by i:j indicates a comparison between ai and aj . A leaf annotated by the permutation

h.1/; .2/; : : : ; .n/i indicates the ordering a.1/ a.2/ a.n/. The shaded path

indicates the decisions made when sorting the input sequence ha1 D 6; a2 D 8; a3 D 5i; the

permutation h3; 1; 2i at the leaf indicates that the sorted ordering is a3 D 5 a1 D 6 a2 D 8.

There are 3Š D 6 possible permutations of the input elements, and so the decision tree must have at

least 6 leaves.

they yield identical information about the relative order of a i and a j . We therefore

assume that all comparisons have the form a i a j .

The decision-tree model

We can view comparison sorts abstractly in terms of decision trees. A decision

tree is a full binary tree that represents the comparisons between elements that

are performed by a particular sorting algorithm operating on an input of a given

size. Control, data movement, and all other aspects of the algorithm are ignored.

Figure 8.1 shows the decision tree corresponding to the insertion sort algorithm

from Section 2.1 operating on an input sequence of three elements.

In a decision tree, we annotate each internal node by i:j for some i and j in the

range 1 i; j n, where n is the number of elements in the input sequence. We

also annotate each leaf by a permutation h.1/; .2/; : : : ; .n/i. (See Section C.1

for background on permutations.) The execution of the sorting algorithm corre-

sponds to tracing a simple path from the root of the decision tree down to a leaf.

Each internal node indicates a comparison a i a j . The left subtree then dictates

subsequent comparisons once we know that a i a j , and the right subtree dictates

subsequent comparisons knowing that a i > a j . When we come to a leaf, the sort-

ing algorithm has established the ordering a .1/ a .2/ a .n/ . Because

any correct sorting algorithm must be able to produce each permutation of its input,

each of the nŠ permutations on n elements must appear as one of the leaves of the

decision tree for a comparison sort to be correct. Furthermore, each of these leaves

must be reachable from the root by a downward path corresponding to an actual

8.1 Lower bounds for sorting 193

execution of the comparison sort. (We shall refer to such leaves as “reachable.”)

Thus, we shall consider only decision trees in which each permutation appears as

a reachable leaf.

A lower bound for the worst case

The length of the longest simple path from the root of a decision tree to any of

its reachable leaves represents the worst-case number of comparisons that the cor-

responding sorting algorithm performs. Consequently, the worst-case number of

comparisons for a given comparison sort algorithm equals the height of its decision

tree. A lower bound on the heights of all decision trees in which each permutation

appears as a reachable leaf is therefore a lower bound on the running time of any

comparison sort algorithm. The following theorem establishes such a lower bound.

Theorem 8.1

Any comparison sort algorithm requires .n lg n/ comparisons in the worst case.

Proof From the preceding discussion, it sufﬁces to determine the height of a

decision tree in which each permutation appears as a reachable leaf. Consider a

decision tree of height h with l reachable leaves corresponding to a comparison

sort on n elements. Because each of the nŠ permutations of the input appears as

some leaf, we have nŠ l. Since a binary tree of height h has no more than 2 h

leaves, we have

nŠ l 2

h

;

which, by taking logarithms, implies

h lg.nŠ/ (since the lg function is monotonically increasing)

D .n lg n/ (by equation (3.19)) .

Corollary 8.2

Heapsort and merge sort are asymptotically optimal comparison sorts.

Proof The O.n lg n/ upper bounds on the running times for heapsort and merge

sort match the .n lg n/ worst-case lower bound from Theorem 8.1.

Exercises

8.1-1

What is the smallest possible depth of a leaf in a decision tree for a comparison

sort?

194 Chapter 8 Sorting in Linear Time

8.1-2

Obtain asymptotically tight bounds on lg.nŠ/ without using Stirling’s approxi-

mation. Instead, evaluate the summation

P n

kD1

lg k using techniques from Sec-

tion A.2.

8.1-3

Show that there is no comparison sort whose running time is linear for at least half

of the nŠ inputs of length n. What about a fraction of 1=n of the inputs of length n?

What about a fraction 1=2 n

?

8.1-4

Suppose that you are given a sequence of n elements to sort. The input sequence

consists of n=k subsequences, each containing k elements. The elements in a given

subsequence are all smaller than the elements in the succeeding subsequence and

larger than the elements in the preceding subsequence. Thus, all that is needed to

sort the whole sequence of length n is to sort the k elements in each of the n=k

subsequences. Show an .n lg k/ lower bound on the number of comparisons

needed to solve this variant of the sorting problem. (Hint: It is not rigorous to

simply combine the lower bounds for the individual subsequences.)

8.2 Counting sort

Counting sort assumes that each of the n input elements is an integer in the range

0 to k, for some integer k. When k D O.n/, the sort runs in ‚.n/ time.

Counting sort determines, for each input element x, the number of elements less

than x. It uses this information to place element x directly into its position in the

output array. For example, if 17 elements are less than x, then x belongs in output

position 18. We must modify this scheme slightly to handle the situation in which

several elements have the same value, since we do not want to put them all in the

same position.

In the code for counting sort, we assume that the input is an array AŒ1 : : n, and

thus A:length D n. We require two other arrays: the array BŒ1 : : n holds the

sorted output, and the array C Œ0 : : k provides temporary working storage.

8.2 Counting sort 195

2 5 3 0 2 3 0 3

1 2 3 4 5 6 7 8

2 0 2 3 0 1

1 2 3 4 5

A

C

(a)

2 2 4 7 7 8 C

(b)

3

1 2 3 4 5 6 7 8

2 2 4 6 7 8

B

C

(c)

3

1 2 3 4 5 6 7 8

1 2 4 6 7 8

B

C

(d)

0 3

1 2 3 4 5 6 7 8

1 2 4 5 7 8

B

C

(e)

0 3

3

1 2 3 4 5 6 7 8

B

(f)

0 3 0 2 2 3 5

0

1 2 3 4 5 0

1 2 3 4 5 0 1 2 3 4 5 0

1 2 3 4 5 0

Figure 8.2 The operation of COUNTING-SORT on an input array AŒ1 : : 8, where each element

of A is a nonnegative integer no larger than k D 5. (a) The array A and the auxiliary array C after

line 5. (b) The array C after line 8. (c)–(e) The output array B and the auxiliary array C after one,

two, and three iterations of the loop in lines 10–12, respectively. Only the lightly shaded elements of

array B have been ﬁlled in. (f) The ﬁnal sorted output array B.

COUNTING-SORT.A; B; k/

1 let C Œ0 : : k be a new array

2 for i D 0 to k

3 C Œi D 0

4 for j D 1 to A:length

5 C ŒAŒj D C ŒAŒj C 1

6 // C Œi now contains the number of elements equal to i.

7 for i D 1 to k

8 C Œi D C Œi C C Œi 1

9 // C Œi now contains the number of elements less than or equal to i.

10 for j D A:length downto 1

11 BŒC ŒAŒj D AŒj

12 C ŒAŒj D C ŒAŒj 1

Figure 8.2 illustrates counting sort. After the for loop of lines 2–3 initializes the

array C to all zeros, the for loop of lines 4–5 inspects each input element. If the

value of an input element is i, we increment C Œi. Thus, after line 5, C Œi holds

the number of input elements equal to i for each integer i D 0; 1; : : : ; k. Lines 7–8

determine for each i D 0; 1; : : : ; k how many input elements are less than or equal

to i by keeping a running sum of the array C .

196 Chapter 8 Sorting in Linear Time

Finally, the for loop of lines 10–12 places each element AŒj into its correct

sorted position in the output array B. If all n elements are distinct, then when we

ﬁrst enter line 10, for each AŒj , the value C ŒAŒj is the correct ﬁnal position

of AŒj in the output array, since there are C ŒAŒj elements less than or equal

to AŒj . Because the elements might not be distinct, we decrement C ŒAŒj each

time we place a value AŒj into the B array. Decrementing C ŒAŒj causes the

next input element with a value equal to AŒj , if one exists, to go to the position

immediately before AŒj in the output array.

How much time does counting sort require? The for loop of lines 2–3 takes

time ‚.k/, the for loop of lines 4–5 takes time ‚.n/, the for loop of lines 7–8 takes

time ‚.k/, and the for loop of lines 10–12 takes time ‚.n/. Thus, the overall time

is ‚.k C n/. In practice, we usually use counting sort when we have k D O.n/, in

which case the running time is ‚.n/.

Counting sort beats the lower bound of .n lg n/ proved in Section 8.1 because

it is not a comparison sort. In fact, no comparisons between input elements occur

anywhere in the code. Instead, counting sort uses the actual values of the elements

to index into an array. The .n lg n/ lower bound for sorting does not apply when

we depart from the comparison sort model.

An important property of counting sort is that it is stable: numbers with the same

value appear in the output array in the same order as they do in the input array. That

is, it breaks ties between two numbers by the rule that whichever number appears

ﬁrst in the input array appears ﬁrst in the output array. Normally, the property of

stability is important only when satellite data are carried around with the element

being sorted. Counting sort’s stability is important for another reason: counting

sort is often used as a subroutine in radix sort. As we shall see in the next section,

in order for radix sort to work correctly, counting sort must be stable.

Exercises

8.2-1

Using Figure 8.2 as a model, illustrate the operation of COUNTING-SORT on the

array A D h6; 0; 2; 0; 1; 3; 4; 6; 1; 3; 2i.

8.2-2

Prove that COUNTING-SORT is stable.

8.2-3

Suppose that we were to rewrite the for loop header in line 10 of the COUNTING-

SORT as

10 for j D 1 to A:length

Show that the algorithm still works properly. Is the modiﬁed algorithm stable?

8.3 Radix sort 197

8.2-4

Describe an algorithm that, given n integers in the range 0 to k, preprocesses its

input and then answers any query about how many of the n integers fall into a

range Œa : : b in O.1/ time. Your algorithm should use ‚.n C k/ preprocessing

time.

8.3 Radix sort

Radix sort is the algorithm used by the card-sorting machines you now ﬁnd only in

computer museums. The cards have 80 columns, and in each column a machine can

punch a hole in one of 12 places. The sorter can be mechanically “programmed”

to examine a given column of each card in a deck and distribute the card into one

of 12 bins depending on which place has been punched. An operator can then

gather the cards bin by bin, so that cards with the ﬁrst place punched are on top of

cards with the second place punched, and so on.

For decimal digits, each column uses only 10 places. (The other two places

are reserved for encoding nonnumeric characters.) A d-digit number would then

occupy a ﬁeld of d columns. Since the card sorter can look at only one column

at a time, the problem of sorting n cards on a d-digit number requires a sorting

algorithm.

Intuitively, you might sort numbers on their most signiﬁcant digit, sort each of

the resulting bins recursively, and then combine the decks in order. Unfortunately,

since the cards in 9 of the 10 bins must be put aside to sort each of the bins, this

procedure generates many intermediate piles of cards that you would have to keep

track of. (See Exercise 8.3-5.)

Radix sort solves the problem of card sorting—counterintuitively—by sorting on

the least signiﬁcant digit ﬁrst. The algorithm then combines the cards into a single

deck, with the cards in the 0 bin preceding the cards in the 1 bin preceding the

cards in the 2 bin, and so on. Then it sorts the entire deck again on the second-least

signiﬁcant digit and recombines the deck in a like manner. The process continues

until the cards have been sorted on all d digits. Remarkably, at that point the cards

are fully sorted on the d-digit number. Thus, only d passes through the deck are

required to sort. Figure 8.3 shows how radix sort operates on a “deck” of seven

3-digit numbers.

In order for radix sort to work correctly, the digit sorts must be stable. The sort

performed by a card sorter is stable, but the operator has to be wary about not

changing the order of the cards as they come out of a bin, even though all the cards

in a bin have the same digit in the chosen column.

198 Chapter 8 Sorting in Linear Time

329

457

657

839

436

720

355

329

457

657

839

436

720

355 329

457

657

839

436

720

355

329

457

657

839

436

720

355

Figure 8.3 The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is

the input. The remaining columns show the list after successive sorts on increasingly signiﬁcant digit

positions. Shading indicates the digit position sorted on to produce each list from the previous one.

In a typical computer, which is a sequential random-access machine, we some-

times use radix sort to sort records of information that are keyed by multiple ﬁelds.

For example, we might wish to sort dates by three keys: year, month, and day. We

could run a sorting algorithm with a comparison function that, given two dates,

compares years, and if there is a tie, compares months, and if another tie occurs,

compares days. Alternatively, we could sort the information three times with a

stable sort: ﬁrst on day, next on month, and ﬁnally on year.

The code for radix sort is straightforward. The following procedure assumes that

each element in the n-element array A has d digits, where digit 1 is the lowest-order

digit and digit d is the highest-order digit.

RADIX-SORT.A; d/

1 for i D 1 to d

2 use a stable sort to sort array A on digit i

Lemma 8.3

Given n d-digit numbers in which each digit can take on up to k possible values,

RADIX-SORT correctly sorts these numbers in ‚.d.n C k// time if the stable sort

it uses takes ‚.n C k/ time.

Proof The correctness of radix sort follows by induction on the column being

sorted (see Exercise 8.3-3). The analysis of the running time depends on the stable

sort used as the intermediate sorting algorithm. When each digit is in the range 0

to k1 (so that it can take on k possible values), and k is not too large, counting sort

is the obvious choice. Each pass over n d-digit numbers then takes time ‚.n C k/.

There are d passes, and so the total time for radix sort is ‚.d.n C k//.

When d is constant and k D O.n/, we can make radix sort run in linear time.

More generally, we have some ﬂexibility in how to break each key into digits.

8.3 Radix sort 199

Lemma 8.4

Given n b-bit numbers and any positive integer r b, RADIX-SORT correctly sorts

these numbers in ‚..b=r/.n C 2 r // time if the stable sort it uses takes ‚.n C k/

time for inputs in the range 0 to k.

Proof For a value r b, we view each key as having d D db=re digits of r bits

each. Each digit is an integer in the range 0 to 2 r 1, so that we can use counting

sort with k D 2 r 1. (For example, we can view a 32-bit word as having four 8-bit

digits, so that b D 32, r D 8, k D 2 r 1 D 255, and d D b=r D 4.) Each pass of

counting sort takes time ‚.n C k/ D ‚.n C 2 r / and there are d passes, for a total

running time of ‚.d.n C 2 r // D ‚..b=r/.n C 2 r //.

For given values of n and b, we wish to choose the value of r, with r b,

that minimizes the expression .b=r/.n C 2 r /. If b < blg nc, then for any value

of r b, we have that .n C 2 r / D ‚.n/. Thus, choosing r D b yields a running

time of .b=b/.n C 2 b / D ‚.n/, which is asymptotically optimal. If b blg nc,

then choosing r D blg nc gives the best time to within a constant factor, which

we can see as follows. Choosing r D blg nc yields a running time of ‚.bn= lg n/.

As we increase r above blg nc, the 2 r

term in the numerator increases faster than

the r term in the denominator, and so increasing r above blg nc yields a running

time of .bn= lg n/. If instead we were to decrease r below blg nc, then the b=r

term increases and the n C 2 r

term remains at ‚.n/.

Is radix sort preferable to a comparison-based sorting algorithm, such as quick-

sort? If b D O.lg n/, as is often the case, and we choose r lg n, then radix sort’s

running time is ‚.n/, which appears to be better than quicksort’s expected running

time of ‚.n lg n/. The constant factors hidden in the ‚-notation differ, however.

Although radix sort may make fewer passes than quicksort over the n keys, each

pass of radix sort may take signiﬁcantly longer. Which sorting algorithm we prefer

depends on the characteristics of the implementations, of the underlying machine

(e.g., quicksort often uses hardware caches more effectively than radix sort), and

of the input data. Moreover, the version of radix sort that uses counting sort as the

intermediate stable sort does not sort in place, which many of the ‚.n lg n/-time

comparison sorts do. Thus, when primary memory storage is at a premium, we

might prefer an in-place algorithm such as quicksort.

Exercises

8.3-1

Using Figure 8.3 as a model, illustrate the operation of RADIX-SORT on the fol-

lowing list of English words: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB,

BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

200 Chapter 8 Sorting in Linear Time

8.3-2

Which of the following sorting algorithms are stable: insertion sort, merge sort,

heapsort, and quicksort? Give a simple scheme that makes any sorting algorithm

stable. How much additional time and space does your scheme entail?

8.3-3

Use induction to prove that radix sort works. Where does your proof need the

assumption that the intermediate sort is stable?

8.3-4

Show how to sort n integers in the range 0 to n 3 1 in O.n/ time.

8.3-5 ?

In the ﬁrst card-sorting algorithm in this section, exactly how many sorting passes

are needed to sort d-digit decimal numbers in the worst case? How many piles of

cards would an operator need to keep track of in the worst case?

8.4 Bucket sort

Bucket sort assumes that the input is drawn from a uniform distribution and has an

average-case running time of O.n/. Like counting sort, bucket sort is fast because

it assumes something about the input. Whereas counting sort assumes that the input

consists of integers in a small range, bucket sort assumes that the input is generated

by a random process that distributes elements uniformly and independently over

the interval Œ0; 1/. (See Section C.2 for a deﬁnition of uniform distribution.)

Bucket sort divides the interval Œ0; 1/ into n equal-sized subintervals, or buckets,

and then distributes the n input numbers into the buckets. Since the inputs are uni-

formly and independently distributed over Œ0; 1/, we do not expect many numbers

to fall into each bucket. To produce the output, we simply sort the numbers in each

bucket and then go through the buckets in order, listing the elements in each.

Our code for bucket sort assumes that the input is an n-element array A and

that each element AŒi in the array satisﬁes 0 AŒi < 1. The code requires an

auxiliary array BŒ0 : : n 1 of linked lists (buckets) and assumes that there is a

mechanism for maintaining such lists. (Section 10.2 describes how to implement

basic operations on linked lists.)

8.4 Bucket sort 201

1

2

3

4

5

6

7

8

9

10

.78

.17

.39

.72

.94

.21

.12

.23

.68

A

(a)

1

2

3

4

5

6

7

8

9

B

(b)

0

.12 .17

.21 .23

.26

.26

.39

.68

.72 .78

.94

Figure 8.4 The operation of BUCKET-SORT for n D 10. (a) The input array AŒ1 : : 10. (b) The

array BŒ0 : : 9 of sorted lists (buckets) after line 8 of the algorithm. Bucket i holds values in the

half-open interval Œi=10; .i C 1/=10/. The sorted output consists of a concatenation in order of the

lists BŒ0; BŒ1; : : : ; BŒ9.

BUCKET-SORT.A/

1 let BŒ0 : : n 1 be a new array

2 n D A:length

3 for i D 0 to n 1

4 make BŒi an empty list

5 for i D 1 to n

6 insert AŒi into list BŒbnAŒic

7 for i D 0 to n 1

8 sort list BŒi with insertion sort

9 concatenate the lists BŒ0; BŒ1; : : : ; BŒn 1 together in order

Figure 8.4 shows the operation of bucket sort on an input array of 10 numbers.

To see that this algorithm works, consider two elements AŒi and AŒj . Assume

without loss of generality that AŒi AŒj . Since bnAŒic bnAŒj c, either

element AŒi goes into the same bucket as AŒj or it goes into a bucket with a lower

index. If AŒi and AŒj go into the same bucket, then the for loop of lines 7–8 puts

them into the proper order. If AŒi and AŒj go into different buckets, then line 9

puts them into the proper order. Therefore, bucket sort works correctly.

To analyze the running time, observe that all lines except line 8 take O.n/ time

in the worst case. We need to analyze the total time taken by the n calls to insertion

sort in line 8.

202 Chapter 8 Sorting in Linear Time

To analyze the cost of the calls to insertion sort, let n i be the random variable

denoting the number of elements placed in bucket BŒi. Since insertion sort runs

in quadratic time (see Section 2.2), the running time of bucket sort is

T .n/ D ‚.n/ C

n1 X

iD0

O.n

2

i

/ :

We now analyze the average-case running time of bucket sort, by computing the

expected value of the running time, where we take the expectation over the input

distribution. Taking expectations of both sides and using linearity of expectation,

we have

E ŒT .n/ D E

"

‚.n/ C

n1 X

iD0

O.n

2

i

/

#

D ‚.n/ C

n1 X

iD0

E

O.n

2

i

/

(by linearity of expectation)

D ‚.n/ C

n1 X

iD0

O

E

n

2

i

(by equation (C.22)) . (8.1)

We claim that

E

n

2

i

D 2 1=n (8.2)

for i D 0; 1; : : : ; n 1. It is no surprise that each bucket i has the same value of

E Œn 2

i

, since each value in the input array A is equally likely to fall in any bucket.

To prove equation (8.2), we deﬁne indicator random variables

X ij D I fAŒj falls in bucket ig

for i D 0; 1; : : : ; n 1 and j D 1; 2; : : : ; n. Thus,

n i D

n X

j D1

X ij :

To compute E Œn 2

i

, we expand the square and regroup terms:

8.4 Bucket sort 203

E

n

2

i

D E

"

n X

j D1

X ij

! 2 #

D E

"

n X

j D1

n X

kD1

X ij X ik

#

D E

2

4

n X

j D1

X

2

ij

C

X

1j n

X

1kn

k¤j

X ij X ik

3

5

D

n X

j D1

E

X

2

ij

C

X

1j n

X

1kn

k¤j

E ŒX ij X ik ; (8.3)

where the last line follows by linearity of expectation. We evaluate the two sum-

mations separately. Indicator random variable X ij is 1 with probability 1=n and 0

otherwise, and therefore

E

X

2

ij

D 1

2

1

n

C 0

2

1

1

n

D

1

n

:

When k ¤ j , the variables X ij and X ik are independent, and hence

E ŒX ij X ik D E ŒX ij E ŒX ik

D

1

n

1

n

D

1

n 2

:

Substituting these two expected values in equation (8.3), we obtain

E

n

2

i

D

n X

j D1

1

n

C

X

1j n

X

1kn

k¤j

1

n 2

D n

1

n

C n.n 1/

1

n 2

D 1 C

n 1

n

D 2

1

n

;

which proves equation (8.2).

204 Chapter 8 Sorting in Linear Time

Using this expected value in equation (8.1), we conclude that the average-case

running time for bucket sort is ‚.n/ C n O.2 1=n/ D ‚.n/.

Even if the input is not drawn from a uniform distribution, bucket sort may still

run in linear time. As long as the input has the property that the sum of the squares

of the bucket sizes is linear in the total number of elements, equation (8.1) tells us

that bucket sort will run in linear time.

Exercises

8.4-1

Using Figure 8.4 as a model, illustrate the operation of BUCKET-SORT on the array

A D h:79; :13; :16; :64; :39; :20; :89; :53; :71; :42i.

8.4-2

Explain why the worst-case running time for bucket sort is ‚.n 2 /. What simple

change to the algorithm preserves its linear average-case running time and makes

its worst-case running time O.n lg n/?

8.4-3

Let X be a random variable that is equal to the number of heads in two ﬂips of a

fair coin. What is E ŒX 2 ? What is E

2 ŒX?

8.4-4 ?

We are given n points in the unit circle, p i D .x i ; y i /, such that 0 < x 2

i

C y 2

i

1

for i D 1; 2; : : : ; n. Suppose that the points are uniformly distributed; that is, the

probability of ﬁnding a point in any region of the circle is proportional to the area

of that region. Design an algorithm with an average-case running time of ‚.n/ to

sort the n points by their distances d i D

p

x 2

i

C y 2

i

from the origin. (Hint: Design

the bucket sizes in BUCKET-SORT to reﬂect the uniform distribution of the points

in the unit circle.)

8.4-5 ?

A probability distribution function P.x/ for a random variable X is deﬁned

by P.x/ D Pr fX xg. Suppose that we draw a list of n random variables

X 1 ; X 2 ; : : : ; X n from a continuous probability distribution function P that is com-

putable in O.1/ time. Give an algorithm that sorts these numbers in linear average-

case time.

Problems for Chapter 8 205

Problems

8-1 Probabilistic lower bounds on comparison sorting

In this problem, we prove a probabilistic .n lg n/ lower bound on the running time

of any deterministic or randomized comparison sort on n distinct input elements.

We begin by examining a deterministic comparison sort A with decision tree T A .

We assume that every permutation of A’s inputs is equally likely.

a. Suppose that each leaf of T A is labeled with the probability that it is reached

given a random input. Prove that exactly nŠ leaves are labeled 1=nŠ and that the

rest are labeled 0.

b. Let D.T / denote the external path length of a decision tree T ; that is, D.T /

is the sum of the depths of all the leaves of T . Let T be a decision tree with

k > 1 leaves, and let LT and RT be the left and right subtrees of T . Show that

D.T / D D.LT/ C D.RT/ C k.

c. Let d.k/ be the minimum value of D.T / over all decision trees T with k > 1

leaves. Show that d.k/ D min 1ik1 fd.i/ C d.k i/ C kg. (Hint: Consider

a decision tree T with k leaves that achieves the minimum. Let i 0 be the number

of leaves in LT and k i 0 the number of leaves in RT.)

d. Prove that for a given value of k > 1 and i in the range 1 i k 1, the

function i lg i C .k i/ lg.k i/ is minimized at i D k=2. Conclude that

d.k/ D .k lg k/.

e. Prove that D.T A / D .nŠ lg.nŠ//, and conclude that the average-case time to

sort n elements is .n lg n/.

Now, consider a randomized comparison sort B. We can extend the decision-

tree model to handle randomization by incorporating two kinds of nodes: ordinary

comparison nodes and “randomization” nodes. A randomization node models a

random choice of the form RANDOM.1; r/ made by algorithm B; the node has r

children, each of which is equally likely to be chosen during an execution of the

algorithm.

f. Show that for any randomized comparison sort B, there exists a deterministic

comparison sort A whose expected number of comparisons is no more than

those made by B.

206 Chapter 8 Sorting in Linear Time

8-2 Sorting in place in linear time

Suppose that we have an array of n data records to sort and that the key of each

record has the value 0 or 1. An algorithm for sorting such a set of records might

possess some subset of the following three desirable characteristics:

1. The algorithm runs in O.n/ time.

2. The algorithm is stable.

3. The algorithm sorts in place, using no more than a constant amount of storage

space in addition to the original array.

a. Give an algorithm that satisﬁes criteria 1 and 2 above.

b. Give an algorithm that satisﬁes criteria 1 and 3 above.

c. Give an algorithm that satisﬁes criteria 2 and 3 above.

d. Can you use any of your sorting algorithms from parts (a)–(c) as the sorting

method used in line 2 of RADIX-SORT, so that RADIX-SORT sorts n records

with b-bit keys in O.bn/ time? Explain how or why not.

e. Suppose that the n records have keys in the range from 1 to k. Show how to

modify counting sort so that it sorts the records in place in O.n C k/ time. You

may use O.k/ storage outside the input array. Is your algorithm stable? (Hint:

How would you do it for k D 3?)

8-3 Sorting variable-length items

a. You are given an array of integers, where different integers may have different

numbers of digits, but the total number of digits over all the integers in the array

is n. Show how to sort the array in O.n/ time.

b. You are given an array of strings, where different strings may have different

numbers of characters, but the total number of characters over all the strings

is n. Show how to sort the strings in O.n/ time.

(Note that the desired order here is the standard alphabetical order; for example,

a < ab < b.)

8-4 Water jugs

Suppose that you are given n red and n blue water jugs, all of different shapes and

sizes. All red jugs hold different amounts of water, as do the blue ones. Moreover,

for every red jug, there is a blue jug that holds the same amount of water, and vice

versa.

Problems for Chapter 8 207

Your task is to ﬁnd a grouping of the jugs into pairs of red and blue jugs that hold

the same amount of water. To do so, you may perform the following operation: pick

a pair of jugs in which one is red and one is blue, ﬁll the red jug with water, and

then pour the water into the blue jug. This operation will tell you whether the red

or the blue jug can hold more water, or that they have the same volume. Assume

that such a comparison takes one time unit. Your goal is to ﬁnd an algorithm that

makes a minimum number of comparisons to determine the grouping. Remember

that you may not directly compare two red jugs or two blue jugs.

a. Describe a deterministic algorithm that uses ‚.n 2 / comparisons to group the

jugs into pairs.

b. Prove a lower bound of .n lg n/ for the number of comparisons that an algo-

rithm solving this problem must make.

c. Give a randomized algorithm whose expected number of comparisons is

O.n lg n/, and prove that this bound is correct. What is the worst-case num-

ber of comparisons for your algorithm?

8-5 Average sorting

Suppose that, instead of sorting an array, we just require that the elements increase

on average. More precisely, we call an n-element array A k-sorted if, for all

i D 1; 2; : : : ; n k, the following holds:

P iCk1

j Di

AŒj

k

P iCk

j DiC1

AŒj

k

:

a. What does it mean for an array to be 1-sorted?

b. Give a permutation of the numbers 1; 2; : : : ; 10 that is 2-sorted, but not sorted.

c. Prove that an n-element array is k-sorted if and only if AŒi AŒi C k for all

i D 1; 2; : : : ; n k.

d. Give an algorithm that k-sorts an n-element array in O.n lg.n=k// time.

We can also show a lower bound on the time to produce a k-sorted array, when k

is a constant.

e. Show that we can sort a k-sorted array of length n in O.n lg k/ time. (Hint:

Use the solution to Exercise 6.5-9. )

f. Show that when k is a constant, k-sorting an n-element array requires .n lg n/

time. (Hint: Use the solution to the previous part along with the lower bound

on comparison sorts.)

208 Chapter 8 Sorting in Linear Time

8-6 Lower bound on merging sorted lists

The problem of merging two sorted lists arises frequently. We have seen a pro-

cedure for it as the subroutine MERGE in Section 2.3.1. In this problem, we will

prove a lower bound of 2n 1 on the worst-case number of comparisons required

to merge two sorted lists, each containing n items.

First we will show a lower bound of 2n o.n/ comparisons by using a decision

tree.

a. Given 2n numbers, compute the number of possible ways to divide them into

two sorted lists, each with n numbers.

b. Using a decision tree and your answer to part (a), show that any algorithm that

correctly merges two sorted lists must perform at least 2n o.n/ comparisons.

Now we will show a slightly tighter 2n 1 bound.

c. Show that if two elements are consecutive in the sorted order and from different

lists, then they must be compared.

d. Use your answer to the previous part to show a lower bound of 2n 1 compar-

isons for merging two sorted lists.

8-7 The 0-1 sorting lemma and columnsort

A compare-exchange operation on two array elements AŒi and AŒj , where i < j ,

has the form

COMPARE-EXCHANGE.A; i; j /

1 if AŒi > AŒj

2 exchange AŒi with AŒj

After the compare-exchange operation, we know that AŒi AŒj .

An oblivious compare-exchange algorithm operates solely by a sequence of

prespeciﬁed compare-exchange operations. The indices of the positions compared

in the sequence must be determined in advance, and although they can depend

on the number of elements being sorted, they cannot depend on the values being

sorted, nor can they depend on the result of any prior compare-exchange operation.

For example, here is insertion sort expressed as an oblivious compare-exchange

algorithm:

INSERTION-SORT.A/

1 for j D 2 to A:length

2 for i D j 1 downto 1

3 COMPARE-EXCHANGE.A; i; i C 1/

Problems for Chapter 8 209

The 0-1 sorting lemma provides a powerful way to prove that an oblivious

compare-exchange algorithm produces a sorted result. It states that if an oblivi-

ous compare-exchange algorithm correctly sorts all input sequences consisting of

only 0s and 1s, then it correctly sorts all inputs containing arbitrary values.

You will prove the 0-1 sorting lemma by proving its contrapositive: if an oblivi-

ous compare-exchange algorithm fails to sort an input containing arbitrary values,

then it fails to sort some 0-1 input. Assume that an oblivious compare-exchange al-

gorithm X fails to correctly sort the array AŒ1 : : n. Let AŒp be the smallest value

in A that algorithm X puts into the wrong location, and let AŒq be the value that

algorithm X moves to the location into which AŒp should have gone. Deﬁne an

array BŒ1 : : n of 0s and 1s as follows:

BŒi D

(

0 if AŒi AŒp ;

1 if AŒi > AŒp :

a. Argue that AŒq > AŒp, so that BŒp D 0 and BŒq D 1.

b. To complete the proof of the 0-1 sorting lemma, prove that algorithm X fails to

sort array B correctly.

Now you will use the 0-1 sorting lemma to prove that a particular sorting algo-

rithm works correctly. The algorithm, columnsort, works on a rectangular array

of n elements. The array has r rows and s columns (so that n D rs), subject to

three restrictions:

r must be even,

s must be a divisor of r, and

r 2s 2

.

When columnsort completes, the array is sorted in column-major order: reading

down the columns, from left to right, the elements monotonically increase.

Columnsort operates in eight steps, regardless of the value of n. The odd steps

are all the same: sort each column individually. Each even step is a ﬁxed permuta-

tion. Here are the steps:

1. Sort each column.

2. Transpose the array, but reshape it back to r rows and s columns. In other

words, turn the leftmost column into the top r=s rows, in order; turn the next

column into the next r=s rows, in order; and so on.

3. Sort each column.

4. Perform the inverse of the permutation performed in step 2.

210 Chapter 8 Sorting in Linear Time

10 14 5

8 7 17

12 1 6

16 9 11

4 15 2

18 3 13

(a)

4 1 2

8 3 5

10 7 6

12 9 11

16 14 13

18 15 17

(b)

4 8 10

12 16 18

1 3 7

9 14 15

2 5 6

11 13 17

(c)

1 3 6

2 5 7

4 8 10

9 13 15

11 14 17

12 16 18

(d)

1 4 11

3 8 14

6 10 17

2 9 12

5 13 16

7 15 18

(e)

1 4 11

2 8 12

3 9 14

5 10 16

6 13 17

7 15 18

(f)

5 10 16

6 13 17

7 15 18

1 4 11

2 8 12

3 9 14

(g)

4 10 16

5 11 17

6 12 18

1 7 13

2 8 14

3 9 15

(h)

1 7 13

2 8 14

3 9 15

4 10 16

5 11 17

6 12 18

(i)

Figure 8.5 The steps of columnsort. (a) The input array with 6 rows and 3 columns. (b) After

sorting each column in step 1. (c) After transposing and reshaping in step 2. (d) After sorting each

column in step 3. (e) After performing step 4, which inverts the permutation from step 2. (f) After

sorting each column in step 5. (g) After shifting by half a column in step 6. (h) After sorting each

column in step 7. (i) After performing step 8, which inverts the permutation from step 6. The array

is now sorted in column-major order.

5. Sort each column.

6. Shift the top half of each column into the bottom half of the same column, and

shift the bottom half of each column into the top half of the next column to the

right. Leave the top half of the leftmost column empty. Shift the bottom half

of the last column into the top half of a new rightmost column, and leave the

bottom half of this new column empty.

7. Sort each column.

8. Perform the inverse of the permutation performed in step 6.

Figure 8.5 shows an example of the steps of columnsort with r D 6 and s D 3.

(Even though this example violates the requirement that r 2s 2

, it happens to

work.)

c. Argue that we can treat columnsort as an oblivious compare-exchange algo-

rithm, even if we do not know what sorting method the odd steps use.

Although it might seem hard to believe that columnsort actually sorts, you will

use the 0-1 sorting lemma to prove that it does. The 0-1 sorting lemma applies

because we can treat columnsort as an oblivious compare-exchange algorithm. A

Notes for Chapter 8 211

couple of deﬁnitions will help you apply the 0-1 sorting lemma. We say that an area

of an array is clean if we know that it contains either all 0s or all 1s. Otherwise,

the area might contain mixed 0s and 1s, and it is dirty. From here on, assume that

the input array contains only 0s and 1s, and that we can treat it as an array with r

rows and s columns.

d. Prove that after steps 1–3, the array consists of some clean rows of 0s at the top,

some clean rows of 1s at the bottom, and at most s dirty rows between them.

e. Prove that after step 4, the array, read in column-major order, starts with a clean

area of 0s, ends with a clean area of 1s, and has a dirty area of at most s 2

elements in the middle.

f. Prove that steps 5–8 produce a fully sorted 0-1 output. Conclude that column-

sort correctly sorts all inputs containing arbitrary values.

g. Now suppose that s does not divide r. Prove that after steps 1–3, the array

consists of some clean rows of 0s at the top, some clean rows of 1s at the

bottom, and at most 2s 1 dirty rows between them. How large must r be,

compared with s, for columnsort to correctly sort when s does not divide r?

h. Suggest a simple change to step 1 that allows us to maintain the requirement

that r 2s 2

even when s does not divide r, and prove that with your change,

columnsort correctly sorts.

Chapter notes

The decision-tree model for studying comparison sorts was introduced by Ford

and Johnson [110]. Knuth’s comprehensive treatise on sorting [211] covers many

variations on the sorting problem, including the information-theoretic lower bound

on the complexity of sorting given here. Ben-Or [39] studied lower bounds for

sorting using generalizations of the decision-tree model.

Knuth credits H. H. Seward with inventing counting sort in 1954, as well as with

the idea of combining counting sort with radix sort. Radix sorting starting with the

least signiﬁcant digit appears to be a folk algorithm widely used by operators of

mechanical card-sorting machines. According to Knuth, the ﬁrst published refer-

ence to the method is a 1929 document by L. J. Comrie describing punched-card

equipment. Bucket sorting has been in use since 1956, when the basic idea was

proposed by E. J. Isaac and R. C. Singleton [188].

Munro and Raman [263] give a stable sorting algorithm that performs O.n 1C /

comparisons in the worst case, where 0 < 1 is any ﬁxed constant. Although

212 Chapter 8 Sorting in Linear Time

any of the O.n lg n/-time algorithms make fewer comparisons, the algorithm by

Munro and Raman moves data only O.n/ times and operates in place.

The case of sorting n b-bit integers in o.n lg n/ time has been considered by

many researchers. Several positive results have been obtained, each under slightly

different assumptions about the model of computation and the restrictions placed

on the algorithm. All the results assume that the computer memory is divided into

addressable b-bit words. Fredman and Willard [115] introduced the fusion tree data

structure and used it to sort n integers in O.n lg n= lg lg n/ time. This bound was

later improved to O.n

p

lg n/ time by Andersson [16]. These algorithms require

the use of multiplication and several precomputed constants. Andersson, Hagerup,

Nilsson, and Raman [17] have shown how to sort n integers in O.n lg lg n/ time

without using multiplication, but their method requires storage that can be un-

bounded in terms of n. Using multiplicative hashing, we can reduce the storage

needed to O.n/, but then the O.n lg lg n/ worst-case bound on the running time

becomes an expected-time bound. Generalizing the exponential search trees of

Andersson [16], Thorup [335] gave an O.n.lg lg n/ 2 /-time sorting algorithm that

does not use multiplication or randomization, and it uses linear space. Combining

these techniques with some new ideas, Han [158] improved the bound for sorting

to O.n lg lg n lg lg lg n/ time. Although these algorithms are important theoretical

breakthroughs, they are all fairly complicated and at the present time seem unlikely

to compete with existing sorting algorithms in practice.

The columnsort algorithm in Problem 8-7 is by Leighton [227].

9 Medians and Order Statistics

The ith order statistic of a set of n elements is the ith smallest element. For

example, the minimum of a set of elements is the ﬁrst order statistic (i D 1),

and the maximum is the nth order statistic (i D n). A median, informally, is

the “halfway point” of the set. When n is odd, the median is unique, occurring at

i D .n C 1/=2. When n is even, there are two medians, occurring at i D n=2 and

i D n=2C1. Thus, regardless of the parity of n, medians occur at i D b.n C 1/=2c

(the lower median) and i D d.n C 1/=2e (the upper median). For simplicity in

this text, however, we consistently use the phrase “the median” to refer to the lower

median.

This chapter addresses the problem of selecting the ith order statistic from a

set of n distinct numbers. We assume for convenience that the set contains dis-

tinct numbers, although virtually everything that we do extends to the situation in

which a set contains repeated values. We formally specify the selection problem

as follows:

Input: A set A of n (distinct) numbers and an integer i, with 1 i n.

Output: The element x 2 A that is larger than exactly i 1 other elements of A.

We can solve the selection problem in O.n lg n/ time, since we can sort the num-

bers using heapsort or merge sort and then simply index the ith element in the

output array. This chapter presents faster algorithms.

In Section 9.1, we examine the problem of selecting the minimum and maxi-

mum of a set of elements. More interesting is the general selection problem, which

we investigate in the subsequent two sections. Section 9.2 analyzes a practical

randomized algorithm that achieves an O.n/ expected running time, assuming dis-

tinct elements. Section 9.3 contains an algorithm of more theoretical interest that

achieves the O.n/ running time in the worst case.

214 Chapter 9 Medians and Order Statistics

9.1 Minimum and maximum

How many comparisons are necessary to determine the minimum of a set of n

elements? We can easily obtain an upper bound of n 1 comparisons: examine

each element of the set in turn and keep track of the smallest element seen so

far. In the following procedure, we assume that the set resides in array A, where

A:length D n.

MINIMUM.A/

1 min D AŒ1

2 for i D 2 to A:length

3 if min > AŒi

4 min D AŒi

5 return min

We can, of course, ﬁnd the maximum with n 1 comparisons as well.

Is this the best we can do? Yes, since we can obtain a lower bound of n 1

comparisons for the problem of determining the minimum. Think of any algorithm

that determines the minimum as a tournament among the elements. Each compar-

ison is a match in the tournament in which the smaller of the two elements wins.

Observing that every element except the winner must lose at least one match, we

conclude that n 1 comparisons are necessary to determine the minimum. Hence,

the algorithm MINIMUM is optimal with respect to the number of comparisons

performed.

Simultaneous minimum and maximum

In some applications, we must ﬁnd both the minimum and the maximum of a set

of n elements. For example, a graphics program may need to scale a set of .x; y/

data to ﬁt onto a rectangular display screen or other graphical output device. To

do so, the program must ﬁrst determine the minimum and maximum value of each

coordinate.

At this point, it should be obvious how to determine both the minimum and the

maximum of n elements using ‚.n/ comparisons, which is asymptotically optimal:

simply ﬁnd the minimum and maximum independently, using n 1 comparisons

for each, for a total of 2n 2 comparisons.

In fact, we can ﬁnd both the minimum and the maximum using at most 3 bn=2c

comparisons. We do so by maintaining both the minimum and maximum elements

seen thus far. Rather than processing each element of the input by comparing it

against the current minimum and maximum, at a cost of 2 comparisons per element,

9.2 Selection in expected linear time 215

we process elements in pairs. We compare pairs of elements from the input ﬁrst

with each other, and then we compare the smaller with the current minimum and

the larger to the current maximum, at a cost of 3 comparisons for every 2 elements.

How we set up initial values for the current minimum and maximum depends

on whether n is odd or even. If n is odd, we set both the minimum and maximum

to the value of the ﬁrst element, and then we process the rest of the elements in

pairs. If n is even, we perform 1 comparison on the ﬁrst 2 elements to determine

the initial values of the minimum and maximum, and then process the rest of the

elements in pairs as in the case for odd n.

Let us analyze the total number of comparisons. If n is odd, then we perform

3 bn=2c comparisons. If n is even, we perform 1 initial comparison followed by

3.n 2/=2 comparisons, for a total of 3n=2 2. Thus, in either case, the total

number of comparisons is at most 3 bn=2c.

Exercises

9.1-1

Show that the second smallest of n elements can be found with n C dlg ne 2

comparisons in the worst case. (Hint: Also ﬁnd the smallest element.)

9.1-2 ?

Prove the lower bound of d3n=2e 2 comparisons in the worst case to ﬁnd both

the maximum and minimum of n numbers. (Hint: Consider how many numbers

are potentially either the maximum or minimum, and investigate how a comparison

affects these counts.)

9.2 Selection in expected linear time

The general selection problem appears more difﬁcult than the simple problem of

ﬁnding a minimum. Yet, surprisingly, the asymptotic running time for both prob-

lems is the same: ‚.n/. In this section, we present a divide-and-conquer algorithm

for the selection problem. The algorithm RANDOMIZED-SELECT is modeled after

the quicksort algorithm of Chapter 7. As in quicksort, we partition the input array

recursively. But unlike quicksort, which recursively processes both sides of the

partition, RANDOMIZED-SELECT works on only one side of the partition. This

difference shows up in the analysis: whereas quicksort has an expected running

time of ‚.n lg n/, the expected running time of RANDOMIZED-SELECT is ‚.n/,

assuming that the elements are distinct.

216 Chapter 9 Medians and Order Statistics

RANDOMIZED-SELECT uses the procedure RANDOMIZED-PARTITION intro-

duced in Section 7.3. Thus, like RANDOMIZED-QUICKSORT, it is a randomized al-

gorithm, since its behavior is determined in part by the output of a random-number

generator. The following code for RANDOMIZED-SELECT returns the ith smallest

element of the array AŒp : : r.

RANDOMIZED-SELECT.A; p; r; i/

1 if p == r

2 return AŒp

3 q D RANDOMIZED-PARTITION.A; p; r/

4 k D q p C 1

5 if i == k // the pivot value is the answer

6 return AŒq

7 elseif i < k

8 return RANDOMIZED-SELECT.A; p; q 1; i/

9 else return RANDOMIZED-SELECT.A; q C 1; r; i k/

The RANDOMIZED-SELECT procedure works as follows. Line 1 checks for the

base case of the recursion, in which the subarray AŒp : : r consists of just one

element. In this case, i must equal 1, and we simply return AŒp in line 2 as the

ith smallest element. Otherwise, the call to RANDOMIZED-PARTITION in line 3

partitions the array AŒp : : r into two (possibly empty) subarrays AŒp : : q 1

and AŒq C 1 : : r such that each element of AŒp : : q 1 is less than or equal

to AŒq, which in turn is less than each element of AŒq C 1 : : r. As in quicksort,

we will refer to AŒq as the pivot element. Line 4 computes the number k of

elements in the subarray AŒp : : q, that is, the number of elements in the low side

of the partition, plus one for the pivot element. Line 5 then checks whether AŒq is

the ith smallest element. If it is, then line 6 returns AŒq. Otherwise, the algorithm

determines in which of the two subarrays AŒp : : q 1 and AŒq C 1 : : r the ith

smallest element lies. If i < k, then the desired element lies on the low side of

the partition, and line 8 recursively selects it from the subarray. If i > k, however,

then the desired element lies on the high side of the partition. Since we already

know k values that are smaller than the ith smallest element of AŒp : : r—namely,

the elements of AŒp : : q—the desired element is the .i k/th smallest element

of AŒq C1 : : r, which line 9 ﬁnds recursively. The code appears to allow recursive

calls to subarrays with 0 elements, but Exercise 9.2-1 asks you to show that this

situation cannot happen.

The worst-case running time for RANDOMIZED-SELECT is ‚.n 2 /, even to ﬁnd

the minimum, because we could be extremely unlucky and always partition around

the largest remaining element, and partitioning takes ‚.n/ time. We will see that

9.2 Selection in expected linear time 217

the algorithm has a linear expected running time, though, and because it is random-

ized, no particular input elicits the worst-case behavior.

To analyze the expected running time of RANDOMIZED-SELECT, we let the run-

ning time on an input array AŒp : : r of n elements be a random variable that we

denote by T .n/, and we obtain an upper bound on E ŒT .n/ as follows. The pro-

cedure RANDOMIZED-PARTITION is equally likely to return any element as the

pivot. Therefore, for each k such that 1 k n, the subarray AŒp : : q has k ele-

ments (all less than or equal to the pivot) with probability 1=n. For k D 1; 2; : : : ; n,

we deﬁne indicator random variables X k where

X k D I fthe subarray AŒp : : q has exactly k elementsg ;

and so, assuming that the elements are distinct, we have

E ŒX k D 1=n : (9.1)

When we call RANDOMIZED-SELECT and choose AŒq as the pivot element, we

do not know, a priori, if we will terminate immediately with the correct answer,

recurse on the subarray AŒp : : q 1, or recurse on the subarray AŒq C 1 : : r.

This decision depends on where the ith smallest element falls relative to AŒq.

Assuming that T .n/ is monotonically increasing, we can upper-bound the time

needed for the recursive call by the time needed for the recursive call on the largest

possible input. In other words, to obtain an upper bound, we assume that the ith

element is always on the side of the partition with the greater number of elements.

For a given call of RANDOMIZED-SELECT, the indicator random variable X k has

the value 1 for exactly one value of k, and it is 0 for all other k. When X k D 1, the

two subarrays on which we might recurse have sizes k 1 and n k. Hence, we

have the recurrence

T .n/

n X

kD1

X k .T .max.k 1; n k// C O.n//

D

n X

kD1

X k T .max.k 1; n k// C O.n/ :

218 Chapter 9 Medians and Order Statistics

Taking expected values, we have

E ŒT .n/

E

"

n X

kD1

X k T .max.k 1; n k// C O.n/

#

D

n X

kD1

E ŒX k T .max.k 1; n k// C O.n/ (by linearity of expectation)

D

n X

kD1

E ŒX k E ŒT .max.k 1; n k// C O.n/ (by equation (C.24))

D

n X

kD1

1

n

E ŒT .max.k 1; n k// C O.n/ (by equation (9.1)) .

In order to apply equation (C.24), we rely on X k and T .max.k 1; n k// being

independent random variables. Exercise 9.2-2 asks you to justify this assertion.

Let us consider the expression max.k 1; n k/. We have

max.k 1; n k/ D

(

k 1 if k > dn=2e ;

n k if k dn=2e :

If n is even, each term from T .dn=2e/ up to T .n 1/ appears exactly twice in

the summation, and if n is odd, all these terms appear twice and T .bn=2c/ appears

once. Thus, we have

E ŒT .n/

2

n

n1 X

kDbn=2c

E ŒT .k/ C O.n/ :

We show that E ŒT .n/ D O.n/ by substitution. Assume that E ŒT .n/ cn for

some constant c that satisﬁes the initial conditions of the recurrence. We assume

that T .n/ D O.1/ for n less than some constant; we shall pick this constant later.

We also pick a constant a such that the function described by the O.n/ term above

(which describes the non-recursive component of the running time of the algo-

rithm) is bounded from above by an for all n > 0. Using this inductive hypothesis,

we have

E ŒT .n/

2

n

n1 X

kDbn=2c

ck C an

D

2c

n

n1 X

kD1

k

bn=2c1 X

kD1

k

!

C an

9.2 Selection in expected linear time 219

D

2c

n

.n 1/n

2

.bn=2c 1/ bn=2c

2

C an

2c

n

.n 1/n

2

.n=2 2/.n=2 1/

2

C an

D

2c

n

n 2 n

2

n 2 =4 3n=2 C 2

2

C an

D

c

n

3n 2

4

C

n

2

2

C an

D c

3n

4

C

1

2

2

n

C an

3cn

4

C

c

2

C an

D cn

cn

4

c

2

an

:

In order to complete the proof, we need to show that for sufﬁciently large n, this

last expression is at most cn or, equivalently, that cn=4 c=2 an 0. If we

add c=2 to both sides and factor out n, we get n.c=4 a/ c=2. As long as we

choose the constant c so that c=4 a > 0, i.e., c > 4a, we can divide both sides

by c=4 a, giving

n

c=2

c=4 a

D

2c

c 4a

:

Thus, if we assume that T .n/ D O.1/ for n < 2c=.c 4a/, then E ŒT .n/ D O.n/.

We conclude that we can ﬁnd any order statistic, and in particular the median, in

expected linear time, assuming that the elements are distinct.

Exercises

9.2-1

Show that RANDOMIZED-SELECT never makes a recursive call to a 0-length array.

9.2-2

Argue that the indicator random variable X k and the value T .max.k 1; n k//

are independent.

9.2-3

Write an iterative version of RANDOMIZED-SELECT.

220 Chapter 9 Medians and Order Statistics

9.2-4

Suppose we use RANDOMIZED-SELECT to select the minimum element of the

array A D h3; 2; 9; 0; 7; 5; 4; 8; 6; 1i. Describe a sequence of partitions that results

in a worst-case performance of RANDOMIZED-SELECT.

9.3 Selection in worst-case linear time

We now examine a selection algorithm whose running time is O.n/ in the worst

case. Like RANDOMIZED-SELECT, the algorithm SELECT ﬁnds the desired ele-

ment by recursively partitioning the input array. Here, however, we guarantee a

good split upon partitioning the array. SELECT uses the deterministic partitioning

algorithm PARTITION from quicksort (see Section 7.1), but modiﬁed to take the

element to partition around as an input parameter.

The SELECT algorithm determines the ith smallest of an input array of n > 1

distinct elements by executing the following steps. (If n D 1, then SELECT merely

returns its only input value as the ith smallest.)

1. Divide the n elements of the input array into bn=5c groups of 5 elements each

and at most one group made up of the remaining n mod 5 elements.

2. Find the median of each of the dn=5e groups by ﬁrst insertion-sorting the ele-

ments of each group (of which there are at most 5) and then picking the median

from the sorted list of group elements.

3. Use SELECT recursively to ﬁnd the median x of the dn=5e medians found in

step 2. (If there are an even number of medians, then by our convention, x is

the lower median.)

4. Partition the input array around the median-of-medians x using the modiﬁed

version of PARTITION. Let k be one more than the number of elements on the

low side of the partition, so that x is the kth smallest element and there are nk

elements on the high side of the partition.

5. If i D k, then return x. Otherwise, use SELECT recursively to ﬁnd the ith

smallest element on the low side if i < k, or the .i k/th smallest element on

the high side if i > k.

To analyze the running time of SELECT, we ﬁrst determine a lower bound on the

number of elements that are greater than the partitioning element x. Figure 9.1

helps us to visualize this bookkeeping. At least half of the medians found in

9.3 Selection in worst-case linear time 221

x

Figure 9.1 Analysis of the algorithm SELECT. The n elements are represented by small circles,

and each group of 5 elements occupies a column. The medians of the groups are whitened, and the

median-of-medians x is labeled. (When ﬁnding the median of an even number of elements, we use

the lower median.) Arrows go from larger elements to smaller, from which we can see that 3 out

of every full group of 5 elements to the right of x are greater than x, and 3 out of every group of 5

elements to the left of x are less than x. The elements known to be greater than x appear on a shaded

background.

step 2 are greater than or equal to the median-of-medians x. 1 Thus, at least half

of the dn=5e groups contribute at least 3 elements that are greater than x, except

for the one group that has fewer than 5 elements if 5 does not divide n exactly, and

the one group containing x itself. Discounting these two groups, it follows that the

number of elements greater than x is at least

3

1

2

l

n

5

m

2

3n

10

6 :

Similarly, at least 3n=10 6 elements are less than x. Thus, in the worst case,

step 5 calls SELECT recursively on at most 7n=10 C 6 elements.

We can now develop a recurrence for the worst-case running time T .n/ of the

algorithm SELECT. Steps 1, 2, and 4 take O.n/ time. (Step 2 consists of O.n/

calls of insertion sort on sets of size O.1/.) Step 3 takes time T .dn=5e/, and step 5

takes time at most T .7n=10 C 6/, assuming that T is monotonically increasing.

We make the assumption, which seems unmotivated at ﬁrst, that any input of fewer

than 140 elements requires O.1/ time; the origin of the magic constant 140 will be

clear shortly. We can therefore obtain the recurrence

1

Because of our assumption that the numbers are distinct, all medians except x are either greater

than or less than x.

222 Chapter 9 Medians and Order Statistics

T .n/

(

O.1/ if n < 140 ;

T .dn=5e/ C T .7n=10 C 6/ C O.n/ if n 140 :

We show that the running time is linear by substitution. More speciﬁcally, we will

show that T .n/ cn for some suitably large constant c and all n > 0. We begin by

assuming that T .n/ cn for some suitably large constant c and all n < 140; this

assumption holds if c is large enough. We also pick a constant a such that the func-

tion described by the O.n/ term above (which describes the non-recursive compo-

nent of the running time of the algorithm) is bounded above by an for all n > 0.

Substituting this inductive hypothesis into the right-hand side of the recurrence

yields

T .n/ c dn=5e C c.7n=10 C 6/ C an

cn=5 C c C 7cn=10 C 6c C an

D 9cn=10 C 7c C an

D cn C .cn=10 C 7c C an/ ;

which is at most cn if

cn=10 C 7c C an 0 : (9.2)

Inequality (9.2) is equivalent to the inequality c 10a.n=.n 70// when n > 70.

Because we assume that n 140, we have n=.n 70/ 2, and so choos-

ing c 20a will satisfy inequality (9.2). (Note that there is nothing special about

the constant 140; we could replace it by any integer strictly greater than 70 and

then choose c accordingly.) The worst-case running time of SELECT is therefore

linear.

As in a comparison sort (see Section 8.1), SELECT and RANDOMIZED-SELECT

determine information about the relative order of elements only by comparing ele-

ments. Recall from Chapter 8 that sorting requires .n lg n/ time in the compari-

son model, even on average (see Problem 8-1). The linear-time sorting algorithms

in Chapter 8 make assumptions about the input. In contrast, the linear-time se-

lection algorithms in this chapter do not require any assumptions about the input.

They are not subject to the .n lg n/ lower bound because they manage to solve

the selection problem without sorting. Thus, solving the selection problem by sort-

ing and indexing, as presented in the introduction to this chapter, is asymptotically

inefﬁcient.

9.3 Selection in worst-case linear time 223

Exercises

9.3-1

In the algorithm SELECT, the input elements are divided into groups of 5. Will

the algorithm work in linear time if they are divided into groups of 7? Argue that

SELECT does not run in linear time if groups of 3 are used.

9.3-2

Analyze SELECT to show that if n 140, then at least dn=4e elements are greater

than the median-of-medians x and at least dn=4e elements are less than x.

9.3-3

Show how quicksort can be made to run in O.n lg n/ time in the worst case, as-

suming that all elements are distinct.

9.3-4 ?

Suppose that an algorithm uses only comparisons to ﬁnd the ith smallest element

in a set of n elements. Show that it can also ﬁnd the i 1 smaller elements and

the n i larger elements without performing any additional comparisons.

9.3-5

Suppose that you have a “black-box” worst-case linear-time median subroutine.

Give a simple, linear-time algorithm that solves the selection problem for an arbi-

trary order statistic.

9.3-6

The kth quantiles of an n-element set are the k 1 order statistics that divide the

sorted set into k equal-sized sets (to within 1). Give an O.n lg k/-time algorithm

to list the kth quantiles of a set.

9.3-7

Describe an O.n/-time algorithm that, given a set S of n distinct numbers and

a positive integer k n, determines the k numbers in S that are closest to the

median of S.

9.3-8

Let XŒ1 : : n and Y Œ1 : : n be two arrays, each containing n numbers already in

sorted order. Give an O.lg n/-time algorithm to ﬁnd the median of all 2n elements

in arrays X and Y .

9.3-9

Professor Olay is consulting for an oil company, which is planning a large pipeline

running east to west through an oil ﬁeld of n wells. The company wants to connect

224 Chapter 9 Medians and Order Statistics

Figure 9.2 Professor Olay needs to determine the position of the east-west oil pipeline that mini-

mizes the total length of the north-south spurs.

a spur pipeline from each well directly to the main pipeline along a shortest route

(either north or south), as shown in Figure 9.2. Given the x- and y-coordinates of

the wells, how should the professor pick the optimal location of the main pipeline,

which would be the one that minimizes the total length of the spurs? Show how to

determine the optimal location in linear time.

Problems

9-1 Largest i numbers in sorted order

Given a set of n numbers, we wish to ﬁnd the i largest in sorted order using a

comparison-based algorithm. Find the algorithm that implements each of the fol-

lowing methods with the best asymptotic worst-case running time, and analyze the

running times of the algorithms in terms of n and i.

a. Sort the numbers, and list the i largest.

b. Build a max-priority queue from the numbers, and call EXTRACT-MAX i times.

c. Use an order-statistic algorithm to ﬁnd the ith largest number, partition around

that number, and sort the i largest numbers.

Problems for Chapter 9 225

9-2 Weighted median

For n distinct elements x 1 ; x 2 ; : : : ; x n with positive weights w 1 ; w 2 ; : : : ; w n such

that

P n

iD1

w i D 1, the weighted (lower) median is the element x k satisfying

X

x i <x k

w i <

1

2

and

X

x i >x k

w i

1

2

:

For example, if the elements are 0:1; 0:35; 0:05; 0:1; 0:15; 0:05; 0:2 and each ele-

ment equals its weight (that is, w i D x i for i D 1; 2; : : : ; 7), then the median is 0:1,

but the weighted median is 0:2.

a. Argue that the median of x 1 ; x 2 ; : : : ; x n is the weighted median of the x i with

weights w i D 1=n for i D 1; 2; : : : ; n.

b. Show how to compute the weighted median of n elements in O.n lg n/ worst-

case time using sorting.

c. Show how to compute the weighted median in ‚.n/ worst-case time using a

linear-time median algorithm such as SELECT from Section 9.3.

The post-ofﬁce location problem is deﬁned as follows. We are given n points

p 1 ; p 2 ; : : : ; p n with associated weights w 1 ; w 2 ; : : : ; w n . We wish to ﬁnd a point p

(not necessarily one of the input points) that minimizes the sum

P n

iD1

w i d.p; p i /,

where d.a; b/ is the distance between points a and b.

d. Argue that the weighted median is a best solution for the 1-dimensional post-

ofﬁce location problem, in which points are simply real numbers and the dis-

tance between points a and b is d.a; b/ D ja bj.

e. Find the best solution for the 2-dimensional post-ofﬁce location problem, in

which the points are .x; y/ coordinate pairs and the distance between points

a D .x 1 ; y 1 / and b D .x 2 ; y 2 / is the Manhattan distance given by d.a; b/ D

jx 1 x 2 j C jy 1 y 2 j.

9-3 Small order statistics

We showed that the worst-case number T .n/ of comparisons used by SELECT

to select the ith order statistic from n numbers satisﬁes T .n/ D ‚.n/, but the

constant hidden by the ‚-notation is rather large. When i is small relative to n, we

can implement a different procedure that uses SELECT as a subroutine but makes

fewer comparisons in the worst case.

226 Chapter 9 Medians and Order Statistics

a. Describe an algorithm that uses U i .n/ comparisons to ﬁnd the ith smallest of n

elements, where

U i .n/ D

(

T .n/ if i n=2 ;

bn=2c C U i .dn=2e/ C T .2i/ otherwise :

(Hint: Begin with bn=2c disjoint pairwise comparisons, and recurse on the set

containing the smaller element from each pair.)

b. Show that, if i < n=2, then U i .n/ D n C O.T .2i/ lg.n=i//.

c. Show that if i is a constant less than n=2, then U i .n/ D n C O.lg n/.

d. Show that if i D n=k for k 2, then U i .n/ D n C O.T .2n=k/ lg k/.

9-4 Alternative analysis of randomized selection

In this problem, we use indicator random variables to analyze the RANDOMIZED-

SELECT procedure in a manner akin to our analysis of RANDOMIZED-QUICKSORT

in Section 7.4.2.

As in the quicksort analysis, we assume that all elements are distinct, and we

rename the elements of the input array A as ´ 1 ; ´ 2 ; : : : ; ´ n , where ´ i is the ith

smallest element. Thus, the call RANDOMIZED-SELECT.A; 1; n; k/ returns ´ k .

For 1 i < j n, let

X ijk D I f´ i is compared with ´ j sometime during the execution of the algorithm

to ﬁnd ´ k g :

a. Give an exact expression for E ŒX ijk . (Hint: Your expression may have differ-

ent values, depending on the values of i, j , and k.)

b. Let X k denote the total number of comparisons between elements of array A

when ﬁnding ´ k . Show that

E ŒX k 2

k X

iD1

n X

j Dk

1

j i C 1

C

n X

j DkC1

j k 1

j k C 1

C

k2 X

iD1

k i 1

k i C 1

!

:

c. Show that E ŒX k 4n.

d. Conclude that, assuming all elements of array A are distinct, RANDOMIZED-

SELECT runs in expected time O.n/.

Notes for Chapter 9 227

Chapter notes

The worst-case linear-time median-ﬁnding algorithm was devised by Blum, Floyd,

Pratt, Rivest, and Tarjan [50]. The fast randomized version is due to Hoare [169].

Floyd and Rivest [108] have developed an improved randomized version that parti-

tions around an element recursively selected from a small sample of the elements.

It is still unknown exactly how many comparisons are needed to determine the

median. Bent and John [41] gave a lower bound of 2n comparisons for median

ﬁnding, and Sch¨onhage, Paterson, and Pippenger [302] gave an upper bound of 3n.

Dor and Zwick have improved on both of these bounds. Their upper bound [93]

is slightly less than 2:95n, and their lower bound [94] is .2 C /n, for a small

positive constant , thereby improving slightly on related work by Dor et al. [92].

Paterson [272] describes some of these results along with other related work.

III Data Structures

Introduction

Sets are as fundamental to computer science as they are to mathematics. Whereas

mathematical sets are unchanging, the sets manipulated by algorithms can grow,

shrink, or otherwise change over time. We call such sets dynamic. The next ﬁve

chapters present some basic techniques for representing ﬁnite dynamic sets and

manipulating them on a computer.

Algorithms may require several different types of operations to be performed on

sets. For example, many algorithms need only the ability to insert elements into,

delete elements from, and test membership in a set. We call a dynamic set that

supports these operations a dictionary. Other algorithms require more complicated

operations. For example, min-priority queues, which Chapter 6 introduced in the

context of the heap data structure, support the operations of inserting an element

into and extracting the smallest element from a set. The best way to implement a

dynamic set depends upon the operations that must be supported.

Elements of a dynamic set

In a typical implementation of a dynamic set, each element is represented by an

object whose attributes can be examined and manipulated if we have a pointer to

the object. (Section 10.3 discusses the implementation of objects and pointers in

programming environments that do not contain them as basic data types.) Some

kinds of dynamic sets assume that one of the object’s attributes is an identifying

key. If the keys are all different, we can think of the dynamic set as being a set

of key values. The object may contain satellite data, which are carried around in

other object attributes but are otherwise unused by the set implementation. It may

230 Part III Data Structures

also have attributes that are manipulated by the set operations; these attributes may

contain data or pointers to other objects in the set.

Some dynamic sets presuppose that the keys are drawn from a totally ordered

set, such as the real numbers, or the set of all words under the usual alphabetic

ordering. A total ordering allows us to deﬁne the minimum element of the set, for

example, or to speak of the next element larger than a given element in a set.

Operations on dynamic sets

Operations on a dynamic set can be grouped into two categories: queries, which

simply return information about the set, and modifying operations, which change

the set. Here is a list of typical operations. Any speciﬁc application will usually

require only a few of these to be implemented.

SEARCH.S; k/

A query that, given a set S and a key value k, returns a pointer x to an element

in S such that x:key D k, or NIL if no such element belongs to S.

INSERT.S; x/

A modifying operation that augments the set S with the element pointed to

by x. We usually assume that any attributes in element x needed by the set

implementation have already been initialized.

DELETE.S; x/

A modifying operation that, given a pointer x to an element in the set S, re-

moves x from S. (Note that this operation takes a pointer to an element x, not

a key value.)

MINIMUM.S/

A query on a totally ordered set S that returns a pointer to the element of S

with the smallest key.

MAXIMUM.S/

A query on a totally ordered set S that returns a pointer to the element of S

with the largest key.

SUCCESSOR.S; x/

A query that, given an element x whose key is from a totally ordered set S,

returns a pointer to the next larger element in S, or NIL if x is the maximum

element.

PREDECESSOR.S; x/

A query that, given an element x whose key is from a totally ordered set S,

returns a pointer to the next smaller element in S, or NIL if x is the minimum

element.

Part III Data Structures 231

In some situations, we can extend the queries SUCCESSOR and PREDECESSOR

so that they apply to sets with nondistinct keys. For a set on n keys, the normal

presumption is that a call to MINIMUM followed by n 1 calls to SUCCESSOR

enumerates the elements in the set in sorted order.

We usually measure the time taken to execute a set operation in terms of the size

of the set. For example, Chapter 13 describes a data structure that can support any

of the operations listed above on a set of size n in time O.lg n/.

Overview of Part III

Chapters 10–14 describe several data structures that we can use to implement

dynamic sets; we shall use many of these later to construct efﬁcient algorithms

for a variety of problems. We already saw another important data structure—the

heap—in Chapter 6.

Chapter 10 presents the essentials of working with simple data structures such

as stacks, queues, linked lists, and rooted trees. It also shows how to implement

objects and pointers in programming environments that do not support them as

primitives. If you have taken an introductory programming course, then much of

this material should be familiar to you.

Chapter 11 introduces hash tables, which support the dictionary operations IN-

SERT, DELETE, and SEARCH. In the worst case, hashing requires ‚.n/ time to per-

form a SEARCH operation, but the expected time for hash-table operations is O.1/.

The analysis of hashing relies on probability, but most of the chapter requires no

background in the subject.

Binary search trees, which are covered in Chapter 12, support all the dynamic-

set operations listed above. In the worst case, each operation takes ‚.n/ time on a

tree with n elements, but on a randomly built binary search tree, the expected time

for each operation is O.lg n/. Binary search trees serve as the basis for many other

data structures.

Chapter 13 introduces red-black trees, which are a variant of binary search trees.

Unlike ordinary binary search trees, red-black trees are guaranteed to perform well:

operations take O.lg n/ time in the worst case. A red-black tree is a balanced search

tree; Chapter 18 in Part V presents another kind of balanced search tree, called a

B-tree. Although the mechanics of red-black trees are somewhat intricate, you can

glean most of their properties from the chapter without studying the mechanics in

detail. Nevertheless, you probably will ﬁnd walking through the code to be quite

instructive.

In Chapter 14, we show how to augment red-black trees to support operations

other than the basic ones listed above. First, we augment them so that we can

dynamically maintain order statistics for a set of keys. Then, we augment them in

a different way to maintain intervals of real numbers.

10 Elementary Data Structures

In this chapter, we examine the representation of dynamic sets by simple data struc-

tures that use pointers. Although we can construct many complex data structures

using pointers, we present only the rudimentary ones: stacks, queues, linked lists,

and rooted trees. We also show ways to synthesize objects and pointers from ar-

rays.

10.1 Stacks and queues

Stacks and queues are dynamic sets in which the element removed from the set

by the DELETE operation is prespeciﬁed. In a stack, the element deleted from

the set is the one most recently inserted: the stack implements a last-in, ﬁrst-out,

or LIFO, policy. Similarly, in a queue, the element deleted is always the one that

has been in the set for the longest time: the queue implements a ﬁrst-in, ﬁrst-out,

or FIFO, policy. There are several efﬁcient ways to implement stacks and queues

on a computer. In this section we show how to use a simple array to implement

each.

Stacks

The INSERT operation on a stack is often called PUSH, and the DELETE opera-

tion, which does not take an element argument, is often called POP. These names

are allusions to physical stacks, such as the spring-loaded stacks of plates used

in cafeterias. The order in which plates are popped from the stack is the reverse

of the order in which they were pushed onto the stack, since only the top plate is

accessible.

As Figure 10.1 shows, we can implement a stack of at most n elements with

an array SŒ1 : : n. The array has an attribute S:top that indexes the most recently

10.1 Stacks and queues 233

1 2 3 4 5 6 7

S 15 6 2 9

1 2 3 4 5 6 7

S 15 6 2 9 17 3

1 2 3 4 5 6 7

S 15 6 2 9 17 3

(a) (b) (c)

S:top D 4 S:top D 6 S:top D 5

Figure 10.1 An array implementation of a stack S. Stack elements appear only in the lightly shaded

positions. (a) Stack S has 4 elements. The top element is 9. (b) Stack S after the calls PUSH.S; 17/

and PUSH.S; 3/. (c) Stack S after the call POP.S/ has returned the element 3, which is the one most

recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is

element 17.

inserted element. The stack consists of elements SŒ1 : : S:top, where SŒ1 is the

element at the bottom of the stack and SŒS:top is the element at the top.

When S:top D 0, the stack contains no elements and is empty. We can test to

see whether the stack is empty by query operation STACK-EMPTY. If we attempt

to pop an empty stack, we say the stack underﬂows, which is normally an error.

If S:top exceeds n, the stack overﬂows. (In our pseudocode implementation, we

don’t worry about stack overﬂow.)

We can implement each of the stack operations with just a few lines of code:

STACK-EMPTY.S/

1 if S:top == 0

2 return TRUE

3 else return FALSE

PUSH.S; x/

1 S:top D S:top C 1

2 SŒS:top D x

POP.S/

1 if STACK-EMPTY.S/

2 error “underﬂow”

3 else S:top D S:top 1

4 return SŒS:top C 1

Figure 10.1 shows the effects of the modifying operations PUSH and POP. Each of

the three stack operations takes O.1/ time.

234 Chapter 10 Elementary Data Structures

1 2 3 4 5 6 7 8 9 10 11 12

Q (a) 15 6 9 8 4

1 2 3 4 5 6 7 8 9 10 11 12

Q (b) 15 6 9 8 4 3 5 17

1 2 3 4 5 6 7 8 9 10 11 12

Q (c) 15 6 9 8 4 3 5 17

Q:head D 7

Q:head D 7 Q:tail D 12

Q:tail D 3

Q:tail D 3

Q:head D 8

Figure 10.2 A queue implemented using an array QŒ1 : : 12. Queue elements appear only in the

lightly shaded positions. (a) The queue has 5 elements, in locations QŒ7 : : 11. (b) The conﬁguration

of the queue after the calls ENQUEUE.Q; 17/, ENQUEUE.Q; 3/, and ENQUEUE.Q; 5/. (c) The

conﬁguration of the queue after the call DEQUEUE.Q/ returns the key value 15 formerly at the

head of the queue. The new head has key 6.

Queues

We call the INSERT operation on a queue ENQUEUE, and we call the DELETE

operation DEQUEUE; like the stack operation POP, DEQUEUE takes no element ar-

gument. The FIFO property of a queue causes it to operate like a line of customers

waiting to pay a cashier. The queue has a head and a tail. When an element is en-

queued, it takes its place at the tail of the queue, just as a newly arriving customer

takes a place at the end of the line. The element dequeued is always the one at

the head of the queue, like the customer at the head of the line who has waited the

longest.

Figure 10.2 shows one way to implement a queue of at most n 1 elements

using an array QŒ1 : : n. The queue has an attribute Q:head that indexes, or points

to, its head. The attribute Q:tail indexes the next location at which a newly arriv-

ing element will be inserted into the queue. The elements in the queue reside in

locations Q:head; Q:head C 1; : : : ; Q:tail 1, where we “wrap around” in the

sense that location 1 immediately follows location n in a circular order. When

Q:head D Q:tail, the queue is empty. Initially, we have Q:head D Q:tail D 1.

If we attempt to dequeue an element from an empty queue, the queue underﬂows.

10.1 Stacks and queues 235

When Q:head D Q:tail C 1, the queue is full, and if we attempt to enqueue an

element, then the queue overﬂows.

In our procedures ENQUEUE and DEQUEUE, we have omitted the error checking

for underﬂow and overﬂow. (Exercise 10.1-4 asks you to supply code that checks

for these two error conditions.) The pseudocode assumes that n D Q:length.

ENQUEUE.Q; x/

1 QŒQ:tail D x

2 if Q:tail == Q:length

3 Q:tail D 1

4 else Q:tail D Q:tail C 1

DEQUEUE.Q/

1 x D QŒQ:head

2 if Q:head == Q:length

3 Q:head D 1

4 else Q:head D Q:head C 1

5 return x

Figure 10.2 shows the effects of the ENQUEUE and DEQUEUE operations. Each

operation takes O.1/ time.

Exercises

10.1-1

Using Figure 10.1 as a model, illustrate the result of each operation in the sequence

PUSH.S; 4/, PUSH.S; 1/, PUSH.S; 3/, POP.S/, PUSH.S; 8/, and POP.S/ on an

initially empty stack S stored in array SŒ1 : : 6.

10.1-2

Explain how to implement two stacks in one array AŒ1 : : n in such a way that

neither stack overﬂows unless the total number of elements in both stacks together

is n. The PUSH and POP operations should run in O.1/ time.

10.1-3

Using Figure 10.2 as a model, illustrate the result of each operation in the

sequence ENQUEUE.Q; 4/, ENQUEUE.Q; 1/, ENQUEUE.Q; 3/, DEQUEUE.Q/,

ENQUEUE.Q; 8/, and DEQUEUE.Q/ on an initially empty queue Q stored in

array QŒ1 : : 6.

10.1-4

Rewrite ENQUEUE and DEQUEUE to detect underﬂow and overﬂow of a queue.

236 Chapter 10 Elementary Data Structures

10.1-5

Whereas a stack allows insertion and deletion of elements at only one end, and a

queue allows insertion at one end and deletion at the other end, a deque (double-

ended queue) allows insertion and deletion at both ends. Write four O.1/-time

procedures to insert elements into and delete elements from both ends of a deque

implemented by an array.

10.1-6

Show how to implement a queue using two stacks. Analyze the running time of the

queue operations.

10.1-7

Show how to implement a stack using two queues. Analyze the running time of the

stack operations.

10.2 Linked lists

A linked list is a data structure in which the objects are arranged in a linear order.

Unlike an array, however, in which the linear order is determined by the array

indices, the order in a linked list is determined by a pointer in each object. Linked

lists provide a simple, ﬂexible representation for dynamic sets, supporting (though

not necessarily efﬁciently) all the operations listed on page 230.

As shown in Figure 10.3, each element of a doubly linked list L is an object with

an attribute key and two other pointer attributes: next and pre. The object may

also contain other satellite data. Given an element x in the list, x:next points to its

successor in the linked list, and x:pre points to its predecessor. If x:pre D NIL,

the element x has no predecessor and is therefore the ﬁrst element, or head, of

the list. If x:next D NIL, the element x has no successor and is therefore the last

element, or tail, of the list. An attribute L:head points to the ﬁrst element of the

list. If L:head D NIL, the list is empty.

A list may have one of several forms. It may be either singly linked or doubly

linked, it may be sorted or not, and it may be circular or not. If a list is singly

linked, we omit the pre pointer in each element. If a list is sorted, the linear order

of the list corresponds to the linear order of keys stored in elements of the list; the

minimum element is then the head of the list, and the maximum element is the

tail. If the list is unsorted, the elements can appear in any order. In a circular list,

the pre pointer of the head of the list points to the tail, and the next pointer of

the tail of the list points to the head. We can think of a circular list as a ring of

10.2 Linked lists 237

9 16 4 1

prev key next

(a)

9 16 4 1 (b) 25

9 16 1 (c) 25 L:head

L:head

L:head

Figure 10.3 (a) A doubly linked list L representing the dynamic set f1; 4; 9; 16g. Each element in

the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous

objects. The next attribute of the tail and the pre attribute of the head are NIL, indicated by a diagonal

slash. The attribute L:head points to the head. (b) Following the execution of LIST-INSERT.L; x/,

where x:key D 25, the linked list has a new object with key 25 as the new head. This new object

points to the old head with key 9. (c) The result of the subsequent call LIST-DELETE.L; x/, where x

points to the object with key 4.

elements. In the remainder of this section, we assume that the lists with which we

are working are unsorted and doubly linked.

Searching a linked list

The procedure LIST-SEARCH.L; k/ ﬁnds the ﬁrst element with key k in list L

by a simple linear search, returning a pointer to this element. If no object with

key k appears in the list, then the procedure returns NIL. For the linked list in

Figure 10.3(a), the call LIST-SEARCH.L; 4/ returns a pointer to the third element,

and the call LIST-SEARCH.L; 7/ returns NIL.

LIST-SEARCH.L; k/

1 x D L:head

2 while x ¤ NIL and x:key ¤ k

3 x D x:next

4 return x

To search a list of n objects, the LIST-SEARCH procedure takes ‚.n/ time in the

worst case, since it may have to search the entire list.

Inserting into a linked list

Given an element x whose key attribute has already been set, the LIST-INSERT

procedure “splices” x onto the front of the linked list, as shown in Figure 10.3(b).

238 Chapter 10 Elementary Data Structures

LIST-INSERT.L; x/

1 x:next D L:head

2 if L:head ¤ NIL

3 L:head:pre D x

4 L:head D x

5 x:pre D NIL

(Recall that our attribute notation can cascade, so that L:head:pre denotes the

pre attribute of the object that L:head points to.) The running time for LIST-

INSERT on a list of n elements is O.1/.

Deleting from a linked list

The procedure LIST-DELETE removes an element x from a linked list L. It must

be given a pointer to x, and it then “splices” x out of the list by updating pointers.

If we wish to delete an element with a given key, we must ﬁrst call LIST-SEARCH

to retrieve a pointer to the element.

LIST-DELETE.L; x/

1 if x:pre ¤ NIL

2 x:pre:next D x:next

3 else L:head D x:next

4 if x:next ¤ NIL

5 x:next:pre D x:pre

Figure 10.3(c) shows how an element is deleted from a linked list. LIST-DELETE

runs in O.1/ time, but if we wish to delete an element with a given key, ‚.n/ time

is required in the worst case because we must ﬁrst call LIST-SEARCH to ﬁnd the

element.

Sentinels

The code for LIST-DELETE would be simpler if we could ignore the boundary

conditions at the head and tail of the list:

LIST-DELETE

0 .L; x/

1 x:pre:next D x:next

2 x:next:pre D x:pre

A sentinel is a dummy object that allows us to simplify boundary conditions. For

example, suppose that we provide with list L an object L:nil that represents NIL

10.2 Linked lists 239

9 16 4 1

9 16 4 1 25

9 16 4 25

(a)

(b)

(c)

(d) L:nil

L:nil

L:nil

L:nil

Figure 10.4 A circular, doubly linked list with a sentinel. The sentinel L:nil appears between the

head and tail. The attribute L:head is no longer needed, since we can access the head of the list

by L:nil:next. (a) An empty list. (b) The linked list from Figure 10.3(a), with key 9 at the head and

key 1 at the tail. (c) The list after executing LIST-INSERT

0

.L; x/, where x:key D 25. The new object

becomes the head of the list. (d) The list after deleting the object with key 1. The new tail is the

object with key 4.

but has all the attributes of the other objects in the list. Wherever we have a ref-

erence to NIL in list code, we replace it by a reference to the sentinel L:nil. As

shown in Figure 10.4, this change turns a regular doubly linked list into a circu-

lar, doubly linked list with a sentinel, in which the sentinel L:nil lies between the

head and tail. The attribute L:nil:next points to the head of the list, and L:nil:pre

points to the tail. Similarly, both the next attribute of the tail and the pre at-

tribute of the head point to L:nil. Since L:nil:next points to the head, we can

eliminate the attribute L:head altogether, replacing references to it by references

to L:nil:next. Figure 10.4(a) shows that an empty list consists of just the sentinel,

and both L:nil:next and L:nil:pre point to L:nil.

The code for LIST-SEARCH remains the same as before, but with the references

to NIL and L:head changed as speciﬁed above:

LIST-SEARCH

0 .L; k/

1 x D L:nil:next

2 while x ¤ L:nil and x:key ¤ k

3 x D x:next

4 return x

We use the two-line procedure LIST-DELETE

0

from before to delete an element

from the list. The following procedure inserts an element into the list:

240 Chapter 10 Elementary Data Structures

LIST-INSERT

0 .L; x/

1 x:next D L:nil:next

2 L:nil:next:pre D x

3 L:nil:next D x

4 x:pre D L:nil

Figure 10.4 shows the effects of LIST-INSERT

0

and LIST-DELETE

0

on a sample list.

Sentinels rarely reduce the asymptotic time bounds of data structure operations,

but they can reduce constant factors. The gain from using sentinels within loops

is usually a matter of clarity of code rather than speed; the linked list code, for

example, becomes simpler when we use sentinels, but we save only O.1/ time in

the LIST-INSERT

0

and LIST-DELETE

0

procedures. In other situations, however, the

use of sentinels helps to tighten the code in a loop, thus reducing the coefﬁcient of,

say, n or n 2

in the running time.

We should use sentinels judiciously. When there are many small lists, the extra

storage used by their sentinels can represent signiﬁcant wasted memory. In this

book, we use sentinels only when they truly simplify the code.

Exercises

10.2-1

Can you implement the dynamic-set operation INSERT on a singly linked list

in O.1/ time? How about DELETE?

10.2-2

Implement a stack using a singly linked list L. The operations PUSH and POP

should still take O.1/ time.

10.2-3

Implement a queue by a singly linked list L. The operations ENQUEUE and DE-

QUEUE should still take O.1/ time.

10.2-4

As written, each loop iteration in the LIST-SEARCH

0

procedure requires two tests:

one for x ¤ L:nil and one for x:key ¤ k. Show how to eliminate the test for

x ¤ L:nil in each iteration.

10.2-5

Implement the dictionary operations INSERT, DELETE, and SEARCH using singly

linked, circular lists. What are the running times of your procedures?

10.3 Implementing pointers and objects 241

10.2-6

The dynamic-set operation UNION takes two disjoint sets S 1 and S 2 as input, and

it returns a set S D S 1 [ S 2 consisting of all the elements of S 1 and S 2 . The

sets S 1 and S 2 are usually destroyed by the operation. Show how to support UNION

in O.1/ time using a suitable list data structure.

10.2-7

Give a ‚.n/-time nonrecursive procedure that reverses a singly linked list of n

elements. The procedure should use no more than constant storage beyond that

needed for the list itself.

10.2-8 ?

Explain how to implement doubly linked lists using only one pointer value x:np per

item instead of the usual two (next and pre). Assume that all pointer values can be

interpreted as k-bit integers, and deﬁne x:np to be x:np D x:next XOR x:pre,

the k-bit “exclusive-or” of x:next and x:pre. (The value NIL is represented by 0.)

Be sure to describe what information you need to access the head of the list. Show

how to implement the SEARCH, INSERT, and DELETE operations on such a list.

Also show how to reverse such a list in O.1/ time.

10.3 Implementing pointers and objects

How do we implement pointers and objects in languages that do not provide them?

In this section, we shall see two ways of implementing linked data structures with-

out an explicit pointer data type. We shall synthesize objects and pointers from

arrays and array indices.

A multiple-array representation of objects

We can represent a collection of objects that have the same attributes by using an

array for each attribute. As an example, Figure 10.5 shows how we can implement

the linked list of Figure 10.3(a) with three arrays. The array key holds the values

of the keys currently in the dynamic set, and the pointers reside in the arrays next

and pre. For a given array index x, the array entries keyŒx, nextŒx, and preŒx

represent an object in the linked list. Under this interpretation, a pointer x is simply

a common index into the key, next, and pre arrays.

In Figure 10.3(a), the object with key 4 follows the object with key 16 in the

linked list. In Figure 10.5, key 4 appears in keyŒ2, and key 16 appears in keyŒ5,

and so nextŒ5 D 2 and preŒ2 D 5. Although the constant NIL appears in the next

242 Chapter 10 Elementary Data Structures

1 2 3 4 5 6 7 8

key

next

prev

L 7

4 1 16 9

3 2 5

5 2 7

Figure 10.5 The linked list of Figure 10.3(a) represented by the arrays key, next, and pre. Each

vertical slice of the arrays represents a single object. Stored pointers correspond to the array indices

shown at the top; the arrows show how to interpret them. Lightly shaded object positions contain list

elements. The variable L keeps the index of the head.

attribute of the tail and the pre attribute of the head, we usually use an integer

(such as 0 or 1) that cannot possibly represent an actual index into the arrays. A

variable L holds the index of the head of the list.

A single-array representation of objects

The words in a computer memory are typically addressed by integers from 0

to M 1, where M is a suitably large integer. In many programming languages,

an object occupies a contiguous set of locations in the computer memory. A pointer

is simply the address of the ﬁrst memory location of the object, and we can address

other memory locations within the object by adding an offset to the pointer.

We can use the same strategy for implementing objects in programming envi-

ronments that do not provide explicit pointer data types. For example, Figure 10.6

shows how to use a single array A to store the linked list from Figures 10.3(a)

and 10.5. An object occupies a contiguous subarray AŒj : : k. Each attribute of

the object corresponds to an offset in the range from 0 to k j , and a pointer to

the object is the index j . In Figure 10.6, the offsets corresponding to key, next, and

pre are 0, 1, and 2, respectively. To read the value of i:pre, given a pointer i, we

add the value i of the pointer to the offset 2, thus reading AŒi C 2.

The single-array representation is ﬂexible in that it permits objects of different

lengths to be stored in the same array. The problem of managing such a heteroge-

neous collection of objects is more difﬁcult than the problem of managing a homo-

geneous collection, where all objects have the same attributes. Since most of the

data structures we shall consider are composed of homogeneous elements, it will

be sufﬁcient for our purposes to use the multiple-array representation of objects.

10.3 Implementing pointers and objects 243

1 2 3 4 5 6 7 8

A

L

4 1 16 9 7 4 4

9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

prev

next

key

19

13 13 19

Figure 10.6 The linked list of Figures 10.3(a) and 10.5 represented in a single array A. Each list

element is an object that occupies a contiguous subarray of length 3 within the array. The three

attributes key, next, and pre correspond to the offsets 0, 1, and 2, respectively, within each object.

A pointer to an object is the index of the ﬁrst element of the object. Objects containing list elements

are lightly shaded, and arrows show the list ordering.

Allocating and freeing objects

To insert a key into a dynamic set represented by a doubly linked list, we must al-

locate a pointer to a currently unused object in the linked-list representation. Thus,

it is useful to manage the storage of objects not currently used in the linked-list

representation so that one can be allocated. In some systems, a garbage collec-

tor is responsible for determining which objects are unused. Many applications,

however, are simple enough that they can bear responsibility for returning an un-

used object to a storage manager. We shall now explore the problem of allocating

and freeing (or deallocating) homogeneous objects using the example of a doubly

linked list represented by multiple arrays.

Suppose that the arrays in the multiple-array representation have length m and

that at some moment the dynamic set contains n m elements. Then n objects

represent elements currently in the dynamic set, and the remaining mn objects are

free; the free objects are available to represent elements inserted into the dynamic

set in the future.

We keep the free objects in a singly linked list, which we call the free list. The

free list uses only the next array, which stores the next pointers within the list.

The head of the free list is held in the global variable free. When the dynamic

set represented by linked list L is nonempty, the free list may be intertwined with

list L, as shown in Figure 10.7. Note that each object in the representation is either

in list L or in the free list, but not in both.

The free list acts like a stack: the next object allocated is the last one freed. We

can use a list implementation of the stack operations PUSH and POP to implement

the procedures for allocating and freeing objects, respectively. We assume that the

global variable free used in the following procedures points to the ﬁrst element of

the free list.

244 Chapter 10 Elementary Data Structures

1 2 3 4 5 6 7 8

key

next

prev

L 7

4 1 16 9

3 2 5

5 2 7

4

8 6 1

free

(a)

1 2 3 4 5 6 7 8

key

next

prev

L 4

4 1 16 9

3 2 5

5 2 7

8

7 6 1

free

(b)

4

25

1 2 3 4 5 6 7 8

key

next

prev

L 4

4 1 9

3 8 2

7 2

5

7 6 1

free

(c)

4

25

Figure 10.7 The effect of the ALLOCATE-OBJECT and FREE-OBJECT procedures. (a) The list

of Figure 10.5 (lightly shaded) and a free list (heavily shaded). Arrows show the free-list structure.

(b) The result of calling ALLOCATE-OBJECT./ (which returns index 4), setting keyŒ4 to 25, and

calling LIST-INSERT.L; 4/. The new free-list head is object 8, which had been nextŒ4 on the free

list. (c) After executing LIST-DELETE.L; 5/, we call FREE-OBJECT.5/. Object 5 becomes the new

free-list head, with object 8 following it on the free list.

ALLOCATE-OBJECT./

1 if free == NIL

2 error “out of space”

3 else x D free

4 free D x:next

5 return x

FREE-OBJECT.x/

1 x:next D free

2 free D x

The free list initially contains all n unallocated objects. Once the free list has been

exhausted, running the ALLOCATE-OBJECT procedure signals an error. We can

even service several linked lists with just a single free list. Figure 10.8 shows two

linked lists and a free list intertwined through key, next, and pre arrays.

The two procedures run in O.1/ time, which makes them quite practical. We

can modify them to work for any homogeneous collection of objects by letting any

one of the attributes in the object act like a next attribute in the free list.

10.3 Implementing pointers and objects 245

1 2 3 4 5 6 7 8 9 10

next

key

prev

free

3

6 2

6 3

7 1 5

7 9

9

10

4 8

1

L

2

L

1

k

1

k

2

k

3

k

5

k

6

k

7

k

9

Figure 10.8 Two linked lists, L1 (lightly shaded) and L2 (heavily shaded), and a free list (dark-

ened) intertwined.

Exercises

10.3-1

Draw a picture of the sequence h13; 4; 8; 19; 5; 11i stored as a doubly linked list

using the multiple-array representation. Do the same for the single-array represen-

tation.

10.3-2

Write the procedures ALLOCATE-OBJECT and FREE-OBJECT for a homogeneous

collection of objects implemented by the single-array representation.

10.3-3

Why don’t we need to set or reset the pre attributes of objects in the implementa-

tion of the ALLOCATE-OBJECT and FREE-OBJECT procedures?

10.3-4

It is often desirable to keep all elements of a doubly linked list compact in storage,

using, for example, the ﬁrst m index locations in the multiple-array representation.

(This is the case in a paged, virtual-memory computing environment.) Explain

how to implement the procedures ALLOCATE-OBJECT and FREE-OBJECT so that

the representation is compact. Assume that there are no pointers to elements of the

linked list outside the list itself. (Hint: Use the array implementation of a stack.)

10.3-5

Let L be a doubly linked list of length n stored in arrays key, pre, and next of

length m. Suppose that these arrays are managed by ALLOCATE-OBJECT and

FREE-OBJECT procedures that keep a doubly linked free list F . Suppose further

that of the m items, exactly n are on list L and m n are on the free list. Write

a procedure COMPACTIFY-LIST.L; F / that, given the list L and the free list F ,

moves the items in L so that they occupy array positions 1; 2; : : : ; n and adjusts the

free list F so that it remains correct, occupying array positions nC1; nC2; : : : ; m.

The running time of your procedure should be ‚.n/, and it should use only a

constant amount of extra space. Argue that your procedure is correct.

246 Chapter 10 Elementary Data Structures

10.4 Representing rooted trees

The methods for representing lists given in the previous section extend to any ho-

mogeneous data structure. In this section, we look speciﬁcally at the problem of

representing rooted trees by linked data structures. We ﬁrst look at binary trees,

and then we present a method for rooted trees in which nodes can have an arbitrary

number of children.

We represent each node of a tree by an object. As with linked lists, we assume

that each node contains a key attribute. The remaining attributes of interest are

pointers to other nodes, and they vary according to the type of tree.

Binary trees

Figure 10.9 shows how we use the attributes p, left, and right to store pointers to

the parent, left child, and right child of each node in a binary tree T . If x:p D NIL,

then x is the root. If node x has no left child, then x:left D NIL, and similarly for

the right child. The root of the entire tree T is pointed to by the attribute T:root. If

T:root D NIL, then the tree is empty.

Rooted trees with unbounded branching

We can extend the scheme for representing a binary tree to any class of trees in

which the number of children of each node is at most some constant k: we replace

the left and right attributes by child 1 ; child 2 ; : : : ; child k . This scheme no longer

works when the number of children of a node is unbounded, since we do not know

how many attributes (arrays in the multiple-array representation) to allocate in ad-

vance. Moreover, even if the number of children k is bounded by a large constant

but most nodes have a small number of children, we may waste a lot of memory.

Fortunately, there is a clever scheme to represent trees with arbitrary numbers of

children. It has the advantage of using only O.n/ space for any n-node rooted tree.

The left-child, right-sibling representation appears in Figure 10.10. As before,

each node contains a parent pointer p, and T:root points to the root of tree T .

Instead of having a pointer to each of its children, however, each node x has only

two pointers:

1. x:left-child points to the leftmost child of node x, and

2. x:right-sibling points to the sibling of x immediately to its right.

If node x has no children, then x:left-child D NIL, and if node x is the rightmost

child of its parent, then x:right-sibling D NIL.

10.4 Representing rooted trees 247

T:root

Figure 10.9 The representation of a binary tree T . Each node x has the attributes x:p (top), x:left

(lower left), and x:right (lower right). The key attributes are not shown.

T:root

Figure 10.10 The left-child, right-sibling representation of a tree T . Each node x has attributes x:p

(top), x:left-child (lower left), and x:right-sibling (lower right). The key attributes are not shown.

248 Chapter 10 Elementary Data Structures

Other tree representations

We sometimes represent rooted trees in other ways. In Chapter 6, for example,

we represented a heap, which is based on a complete binary tree, by a single array

plus the index of the last node in the heap. The trees that appear in Chapter 21 are

traversed only toward the root, and so only the parent pointers are present; there

are no pointers to children. Many other schemes are possible. Which scheme is

best depends on the application.

Exercises

10.4-1

Draw the binary tree rooted at index 6 that is represented by the following at-

tributes:

index key left right

1 12 7 3

2 15 8 NIL

3 4 10 NIL

4 10 5 9

5 2 NIL NIL

6 18 1 4

7 7 NIL NIL

8 14 6 2

9 21 NIL NIL

10 5 NIL NIL

10.4-2

Write an O.n/-time recursive procedure that, given an n-node binary tree, prints

out the key of each node in the tree.

10.4-3

Write an O.n/-time nonrecursive procedure that, given an n-node binary tree,

prints out the key of each node in the tree. Use a stack as an auxiliary data structure.

10.4-4

Write an O.n/-time procedure that prints all the keys of an arbitrary rooted tree

with n nodes, where the tree is stored using the left-child, right-sibling representa-

tion.

10.4-5 ?

Write an O.n/-time nonrecursive procedure that, given an n-node binary tree,

prints out the key of each node. Use no more than constant extra space outside

Problems for Chapter 10 249

of the tree itself and do not modify the tree, even temporarily, during the proce-

dure.

10.4-6 ?

The left-child, right-sibling representation of an arbitrary rooted tree uses three

pointers in each node: left-child, right-sibling, and parent. From any node, its

parent can be reached and identiﬁed in constant time and all its children can be

reached and identiﬁed in time linear in the number of children. Show how to use

only two pointers and one boolean value in each node so that the parent of a node

or all of its children can be reached and identiﬁed in time linear in the number of

children.

Problems

10-1 Comparisons among lists

For each of the four types of lists in the following table, what is the asymptotic

worst-case running time for each dynamic-set operation listed?

unsorted, sorted, unsorted, sorted,

singly singly doubly doubly

linked linked linked linked

SEARCH.L; k/

INSERT.L; x/

DELETE.L; x/

SUCCESSOR.L; x/

PREDECESSOR.L; x/

MINIMUM.L/

MAXIMUM.L/

250 Chapter 10 Elementary Data Structures

10-2 Mergeable heaps using linked lists

A mergeable heap supports the following operations: MAKE-HEAP (which creates

an empty mergeable heap), INSERT, MINIMUM, EXTRACT-MIN, and UNION. 1

Show how to implement mergeable heaps using linked lists in each of the following

cases. Try to make each operation as efﬁcient as possible. Analyze the running

time of each operation in terms of the size of the dynamic set(s) being operated on.

a. Lists are sorted.

b. Lists are unsorted.

c. Lists are unsorted, and dynamic sets to be merged are disjoint.

10-3 Searching a sorted compact list

Exercise 10.3-4 asked how we might maintain an n-element list compactly in the

ﬁrst n positions of an array. We shall assume that all keys are distinct and that the

compact list is also sorted, that is, keyŒi < keyŒnextŒi for all i D 1; 2; : : : ; n such

that nextŒi ¤ NIL. We will also assume that we have a variable L that contains

the index of the ﬁrst element on the list. Under these assumptions, you will show

that we can use the following randomized algorithm to search the list in O.

p

n/

expected time.

COMPACT-LIST-SEARCH.L; n; k/

1 i D L

2 while i ¤ NIL and keyŒi < k

3 j D RANDOM.1; n/

4 if keyŒi < keyŒj and keyŒj k

5 i D j

6 if keyŒi == k

7 return i

8 i D nextŒi

9 if i == NIL or keyŒi > k

10 return NIL

11 else return i

If we ignore lines 3–7 of the procedure, we have an ordinary algorithm for

searching a sorted linked list, in which index i points to each position of the list in

1

Because we have deﬁned a mergeable heap to support MINIMUM and EXTRACT-MIN, we can also

refer to it as a mergeable min-heap. Alternatively, if it supported MAXIMUM and EXTRACT-MAX,

it would be a mergeable max-heap.

Problems for Chapter 10 251

turn. The search terminates once the index i “falls off” the end of the list or once

keyŒi k. In the latter case, if keyŒi D k, clearly we have found a key with the

value k. If, however, keyŒi > k, then we will never ﬁnd a key with the value k,

and so terminating the search was the right thing to do.

Lines 3–7 attempt to skip ahead to a randomly chosen position j . Such a skip

beneﬁts us if keyŒj is larger than keyŒi and no larger than k; in such a case, j

marks a position in the list that i would have to reach during an ordinary list search.

Because the list is compact, we know that any choice of j between 1 and n indexes

some object in the list rather than a slot on the free list.

Instead of analyzing the performance of COMPACT-LIST-SEARCH directly, we

shall analyze a related algorithm, COMPACT-LIST-SEARCH

0

, which executes two

separate loops. This algorithm takes an additional parameter t which determines

an upper bound on the number of iterations of the ﬁrst loop.

COMPACT-LIST-SEARCH

0 .L; n; k; t/

1 i D L

2 for q D 1 to t

3 j D RANDOM.1; n/

4 if keyŒi < keyŒj and keyŒj k

5 i D j

6 if keyŒi == k

7 return i

8 while i ¤ NIL and keyŒi < k

9 i D nextŒi

10 if i == NIL or keyŒi > k

11 return NIL

12 else return i

To compare the execution of the algorithms COMPACT-LIST-SEARCH.L; n; k/

and COMPACT-LIST-SEARCH

0

.L; n; k; t/, assume that the sequence of integers re-

turned by the calls of RANDOM.1; n/ is the same for both algorithms.

a. Suppose that COMPACT-LIST-SEARCH.L; n; k/ takes t iterations of the while

loop of lines 2–8. Argue that COMPACT-LIST-SEARCH

0

.L; n; k; t/ returns the

same answer and that the total number of iterations of both the for and while

loops within COMPACT-LIST-SEARCH

0

is at least t.

In the call COMPACT-LIST-SEARCH

0

.L; n; k; t/, let X t be the random variable that

describes the distance in the linked list (that is, through the chain of next pointers)

from position i to the desired key k after t iterations of the for loop of lines 2–7

have occurred.

252 Chapter 10 Elementary Data Structures

b. Argue that the expected running time of COMPACT-LIST-SEARCH

0

.L; n; k; t/

is O.t C E ŒX t /.

c. Show that E ŒX t

P n

rD1

.1 r=n/ t

. (Hint: Use equation (C.25).)

d. Show that

P n1

rD0

r t n tC1 =.t C 1/.

e. Prove that E ŒX t n=.t C 1/.

f. Show that COMPACT-LIST-SEARCH

0

.L; n; k; t/ runs in O.t C n=t/ expected

time.

g. Conclude that COMPACT-LIST-SEARCH runs in O.

p

n/ expected time.

h. Why do we assume that all keys are distinct in COMPACT-LIST-SEARCH? Ar-

gue that random skips do not necessarily help asymptotically when the list con-

tains repeated key values.

Chapter notes

Aho, Hopcroft, and Ullman [6] and Knuth [209] are excellent references for ele-

mentary data structures. Many other texts cover both basic data structures and their

implementation in a particular programming language. Examples of these types of

textbooks include Goodrich and Tamassia [147], Main [241], Shaffer [311], and

Weiss [352, 353, 354]. Gonnet [145] provides experimental data on the perfor-

mance of many data-structure operations.

The origin of stacks and queues as data structures in computer science is un-

clear, since corresponding notions already existed in mathematics and paper-based

business practices before the introduction of digital computers. Knuth [209] cites

A. M. Turing for the development of stacks for subroutine linkage in 1947.

Pointer-based data structures also seem to be a folk invention. According to

Knuth, pointers were apparently used in early computers with drum memories. The

A-1 language developed by G. M. Hopper in 1951 represented algebraic formulas

as binary trees. Knuth credits the IPL-II language, developed in 1956 by A. Newell,

J. C. Shaw, and H. A. Simon, for recognizing the importance and promoting the

use of pointers. Their IPL-III language, developed in 1957, included explicit stack

operations.

11 Hash Tables

Many applications require a dynamic set that supports only the dictionary opera-

tions INSERT, SEARCH, and DELETE. For example, a compiler that translates a

programming language maintains a symbol table, in which the keys of elements

are arbitrary character strings corresponding to identiﬁers in the language. A hash

table is an effective data structure for implementing dictionaries. Although search-

ing for an element in a hash table can take as long as searching for an element in a

linked list—‚.n/ time in the worst case—in practice, hashing performs extremely

well. Under reasonable assumptions, the average time to search for an element in

a hash table is O.1/.

A hash table generalizes the simpler notion of an ordinary array. Directly ad-

dressing into an ordinary array makes effective use of our ability to examine an

arbitrary position in an array in O.1/ time. Section 11.1 discusses direct address-

ing in more detail. We can take advantage of direct addressing when we can afford

to allocate an array that has one position for every possible key.

When the number of keys actually stored is small relative to the total number of

possible keys, hash tables become an effective alternative to directly addressing an

array, since a hash table typically uses an array of size proportional to the number

of keys actually stored. Instead of using the key as an array index directly, the array

index is computed from the key. Section 11.2 presents the main ideas, focusing on

“chaining” as a way to handle “collisions,” in which more than one key maps to the

same array index. Section 11.3 describes how we can compute array indices from

keys using hash functions. We present and analyze several variations on the basic

theme. Section 11.4 looks at “open addressing,” which is another way to deal with

collisions. The bottom line is that hashing is an extremely effective and practical

technique: the basic dictionary operations require only O.1/ time on the average.

Section 11.5 explains how “perfect hashing” can support searches in O.1/ worst-

case time, when the set of keys being stored is static (that is, when the set of keys

never changes once stored).

254 Chapter 11 Hash Tables

11.1 Direct-address tables

Direct addressing is a simple technique that works well when the universe U of

keys is reasonably small. Suppose that an application needs a dynamic set in which

each element has a key drawn from the universe U D f0; 1; : : : ; m 1g, where m

is not too large. We shall assume that no two elements have the same key.

To represent the dynamic set, we use an array, or direct-address table, denoted

by T Œ0 : : m 1, in which each position, or slot, corresponds to a key in the uni-

verse U . Figure 11.1 illustrates the approach; slot k points to an element in the set

with key k. If the set contains no element with key k, then T Œk D NIL.

The dictionary operations are trivial to implement:

DIRECT-ADDRESS-SEARCH.T; k/

1 return T Œk

DIRECT-ADDRESS-INSERT.T; x/

1 T Œx:key D x

DIRECT-ADDRESS-DELETE.T; x/

1 T Œx:key D NIL

Each of these operations takes only O.1/ time.

T

U

(universe of keys)

K

(actual

keys)

2

3

5

8

1

9

4

0

7

6

2

3

5

8

key satellite data

2

0

1

3

4

5

6

7

8

9

Figure 11.1 How to implement a dynamic set by a direct-address table T . Each key in the universe

U D f0; 1; : : : ; 9g corresponds to an index in the table. The set K D f2; 3; 5; 8g of actual keys

determines the slots in the table that contain pointers to elements. The other slots, heavily shaded,

contain NIL.

11.1 Direct-address tables 255

For some applications, the direct-address table itself can hold the elements in the

dynamic set. That is, rather than storing an element’s key and satellite data in an

object external to the direct-address table, with a pointer from a slot in the table to

the object, we can store the object in the slot itself, thus saving space. We would

use a special key within an object to indicate an empty slot. Moreover, it is often

unnecessary to store the key of the object, since if we have the index of an object

in the table, we have its key. If keys are not stored, however, we must have some

way to tell whether the slot is empty.

Exercises

11.1-1

Suppose that a dynamic set S is represented by a direct-address table T of length m.

Describe a procedure that ﬁnds the maximum element of S. What is the worst-case

performance of your procedure?

11.1-2

A bit vector is simply an array of bits (0s and 1s). A bit vector of length m takes

much less space than an array of m pointers. Describe how to use a bit vector

to represent a dynamic set of distinct elements with no satellite data. Dictionary

operations should run in O.1/ time.

11.1-3

Suggest how to implement a direct-address table in which the keys of stored el-

ements do not need to be distinct and the elements can have satellite data. All

three dictionary operations (INSERT, DELETE, and SEARCH) should run in O.1/

time. (Don’t forget that DELETE takes as an argument a pointer to an object to be

deleted, not a key.)

11.1-4 ?

We wish to implement a dictionary by using direct addressing on a huge array. At

the start, the array entries may contain garbage, and initializing the entire array

is impractical because of its size. Describe a scheme for implementing a direct-

address dictionary on a huge array. Each stored object should use O.1/ space;

the operations SEARCH, INSERT, and DELETE should take O.1/ time each; and

initializing the data structure should take O.1/ time. (Hint: Use an additional array,

treated somewhat like a stack whose size is the number of keys actually stored in

the dictionary, to help determine whether a given entry in the huge array is valid or

not.)

256 Chapter 11 Hash Tables

11.2 Hash tables

The downside of direct addressing is obvious: if the universe U is large, storing

a table T of size jU j may be impractical, or even impossible, given the memory

available on a typical computer. Furthermore, the set K of keys actually stored

may be so small relative to U that most of the space allocated for T would be

wasted.

When the set K of keys stored in a dictionary is much smaller than the uni-

verse U of all possible keys, a hash table requires much less storage than a direct-

address table. Speciﬁcally, we can reduce the storage requirement to ‚.jKj/ while

we maintain the beneﬁt that searching for an element in the hash table still requires

only O.1/ time. The catch is that this bound is for the average-case time, whereas

for direct addressing it holds for the worst-case time.

With direct addressing, an element with key k is stored in slot k. With hashing,

this element is stored in slot h.k/; that is, we use a hash function h to compute the

slot from the key k. Here, h maps the universe U of keys into the slots of a hash

table T Œ0 : : m 1:

h W U ! f0; 1; : : : ; m 1g ;

where the size m of the hash table is typically much less than jU j. We say that an

element with key k hashes to slot h.k/; we also say that h.k/ is the hash value of

key k. Figure 11.2 illustrates the basic idea. The hash function reduces the range

of array indices and hence the size of the array. Instead of a size of jU j, the array

can have size m.

T

U

(universe of keys)

K

(actual

keys)

0

m–1

k

1

k

2

k

3

k

4 k

5

h(k

1

)

h(k

4

)

h(k

3

)

h(k

2

) = h(k

5

)

Figure 11.2 Using a hash function h to map keys to hash-table slots. Because keys k2 and k5 map

to the same slot, they collide.

11.2 Hash tables 257

T

U

(universe of keys)

K

(actual

keys)

k

1

k

2

k

3

k

4

k

5

k

6

k

7

k

8

k

1

k

2

k

3

k

4

k

5

k

6

k

7

k

8

Figure 11.3 Collision resolution by chaining. Each hash-table slot T Œj contains a linked list of

all the keys whose hash value is j . For example, h.k1/ D h.k4/ and h.k5/ D h.k7/ D h.k2/.

The linked list can be either singly or doubly linked; we show it as doubly linked because deletion is

faster that way.

There is one hitch: two keys may hash to the same slot. We call this situation

a collision. Fortunately, we have effective techniques for resolving the conﬂict

created by collisions.

Of course, the ideal solution would be to avoid collisions altogether. We might

try to achieve this goal by choosing a suitable hash function h. One idea is to

make h appear to be “random,” thus avoiding collisions or at least minimizing

their number. The very term “to hash,” evoking images of random mixing and

chopping, captures the spirit of this approach. (Of course, a hash function h must be

deterministic in that a given input k should always produce the same output h.k/.)

Because jU j > m, however, there must be at least two keys that have the same hash

value; avoiding collisions altogether is therefore impossible. Thus, while a well-

designed, “random”-looking hash function can minimize the number of collisions,

we still need a method for resolving the collisions that do occur.

The remainder of this section presents the simplest collision resolution tech-

nique, called chaining. Section 11.4 introduces an alternative method for resolving

collisions, called open addressing.

Collision resolution by chaining

In chaining, we place all the elements that hash to the same slot into the same

linked list, as Figure 11.3 shows. Slot j contains a pointer to the head of the list of

all stored elements that hash to j ; if there are no such elements, slot j contains NIL.

258 Chapter 11 Hash Tables

The dictionary operations on a hash table T are easy to implement when colli-

sions are resolved by chaining:

CHAINED-HASH-INSERT.T; x/

1 insert x at the head of list T Œh.x:key/

CHAINED-HASH-SEARCH.T; k/

1 search for an element with key k in list T Œh.k/

CHAINED-HASH-DELETE.T; x/

1 delete x from the list T Œh.x:key/

The worst-case running time for insertion is O.1/. The insertion procedure is fast

in part because it assumes that the element x being inserted is not already present in

the table; if necessary, we can check this assumption (at additional cost) by search-

ing for an element whose key is x:key before we insert. For searching, the worst-

case running time is proportional to the length of the list; we shall analyze this

operation more closely below. We can delete an element in O.1/ time if the lists

are doubly linked, as Figure 11.3 depicts. (Note that CHAINED-HASH-DELETE

takes as input an element x and not its key k, so that we don’t have to search for x

ﬁrst. If the hash table supports deletion, then its linked lists should be doubly linked

so that we can delete an item quickly. If the lists were only singly linked, then to

delete element x, we would ﬁrst have to ﬁnd x in the list T Œh.x:key/ so that we

could update the next attribute of x’s predecessor. With singly linked lists, both

deletion and searching would have the same asymptotic running times.)

Analysis of hashing with chaining

How well does hashing with chaining perform? In particular, how long does it take

to search for an element with a given key?

Given a hash table T with m slots that stores n elements, we deﬁne the load

factor ˛ for T as n=m, that is, the average number of elements stored in a chain.

Our analysis will be in terms of ˛, which can be less than, equal to, or greater

than 1.

The worst-case behavior of hashing with chaining is terrible: all n keys hash

to the same slot, creating a list of length n. The worst-case time for searching is

thus ‚.n/ plus the time to compute the hash function—no better than if we used

one linked list for all the elements. Clearly, we do not use hash tables for their

worst-case performance. (Perfect hashing, described in Section 11.5, does provide

good worst-case performance when the set of keys is static, however.)

The average-case performance of hashing depends on how well the hash func-

tion h distributes the set of keys to be stored among the m slots, on the average.

11.2 Hash tables 259

Section 11.3 discusses these issues, but for now we shall assume that any given

element is equally likely to hash into any of the m slots, independently of where

any other element has hashed to. We call this the assumption of simple uniform

hashing.

For j D 0; 1; : : : ; m 1, let us denote the length of the list T Œj by n j , so that

n D n 0 C n 1 C C n m1 ; (11.1)

and the expected value of n j is E Œn j D ˛ D n=m.

We assume that O.1/ time sufﬁces to compute the hash value h.k/, so that

the time required to search for an element with key k depends linearly on the

length n h.k/ of the list T Œh.k/. Setting aside the O.1/ time required to compute

the hash function and to access slot h.k/, let us consider the expected number of

elements examined by the search algorithm, that is, the number of elements in the

list T Œh.k/ that the algorithm checks to see whether any have a key equal to k. We

shall consider two cases. In the ﬁrst, the search is unsuccessful: no element in the

table has key k. In the second, the search successfully ﬁnds an element with key k.

Theorem 11.1

In a hash table in which collisions are resolved by chaining, an unsuccessful search

takes average-case time ‚.1C˛/, under the assumption of simple uniform hashing.

Proof Under the assumption of simple uniform hashing, any key k not already

stored in the table is equally likely to hash to any of the m slots. The expected time

to search unsuccessfully for a key k is the expected time to search to the end of

list T Œh.k/, which has expected length E Œn h.k/ D ˛. Thus, the expected number

of elements examined in an unsuccessful search is ˛, and the total time required

(including the time for computing h.k/) is ‚.1 C ˛/.

The situation for a successful search is slightly different, since each list is not

equally likely to be searched. Instead, the probability that a list is searched is pro-

portional to the number of elements it contains. Nonetheless, the expected search

time still turns out to be ‚.1 C ˛/.

Theorem 11.2

In a hash table in which collisions are resolved by chaining, a successful search

takes average-case time ‚.1C˛/, under the assumption of simple uniform hashing.

Proof We assume that the element being searched for is equally likely to be any

of the n elements stored in the table. The number of elements examined during a

successful search for an element x is one more than the number of elements that

260 Chapter 11 Hash Tables

appear before x in x’s list. Because new elements are placed at the front of the

list, elements before x in the list were all inserted after x was inserted. To ﬁnd

the expected number of elements examined, we take the average, over the n ele-

ments x in the table, of 1 plus the expected number of elements added to x’s list

after x was added to the list. Let x i denote the ith element inserted into the ta-

ble, for i D 1; 2; : : : ; n, and let k i D x i :key. For keys k i and k j , we deﬁne the

indicator random variable X ij D I fh.k i / D h.k j /g. Under the assumption of sim-

ple uniform hashing, we have Pr fh.k i / D h.k j /g D 1=m, and so by Lemma 5.1,

E ŒX ij D 1=m. Thus, the expected number of elements examined in a successful

search is

E

"

1

n

n X

iD1

1 C

n X

j DiC1

X ij

!#

D

1

n

n X

iD1

1 C

n X

j DiC1

E ŒX ij

!

(by linearity of expectation)

D

1

n

n X

iD1

1 C

n X

j DiC1

1

m

!

D 1 C

1

nm

n X

iD1

.n i/

D 1 C

1

nm

n X

iD1

n

n X

iD1

i

!

D 1 C

1

nm

n

2

n.n C 1/

2

(by equation (A.1))

D 1 C

n 1

2m

D 1 C

˛

2

˛

2n

:

Thus, the total time required for a successful search (including the time for com-

puting the hash function) is ‚.2 C ˛=2 ˛=2n/ D ‚.1 C ˛/.

What does this analysis mean? If the number of hash-table slots is at least pro-

portional to the number of elements in the table, we have n D O.m/ and, con-

sequently, ˛ D n=m D O.m/=m D O.1/. Thus, searching takes constant time

on average. Since insertion takes O.1/ worst-case time and deletion takes O.1/

worst-case time when the lists are doubly linked, we can support all dictionary

operations in O.1/ time on average.

11.2 Hash tables 261

Exercises

11.2-1

Suppose we use a hash function h to hash n distinct keys into an array T of

length m. Assuming simple uniform hashing, what is the expected number of

collisions? More precisely, what is the expected cardinality of ffk; lg W k ¤ l and

h.k/ D h.l/g?

11.2-2

Demonstrate what happens when we insert the keys 5; 28; 19; 15; 20; 33; 12; 17; 10

into a hash table with collisions resolved by chaining. Let the table have 9 slots,

and let the hash function be h.k/ D k mod 9.

11.2-3

Professor Marley hypothesizes that he can obtain substantial performance gains by

modifying the chaining scheme to keep each list in sorted order. How does the pro-

fessor’s modiﬁcation affect the running time for successful searches, unsuccessful

searches, insertions, and deletions?

11.2-4

Suggest how to allocate and deallocate storage for elements within the hash table

itself by linking all unused slots into a free list. Assume that one slot can store

a ﬂag and either one element plus a pointer or two pointers. All dictionary and

free-list operations should run in O.1/ expected time. Does the free list need to be

doubly linked, or does a singly linked free list sufﬁce?

11.2-5

Suppose that we are storing a set of n keys into a hash table of size m. Show that if

the keys are drawn from a universe U with jU j > nm, then U has a subset of size n

consisting of keys that all hash to the same slot, so that the worst-case searching

time for hashing with chaining is ‚.n/.

11.2-6

Suppose we have stored n keys in a hash table of size m, with collisions resolved by

chaining, and that we know the length of each chain, including the length L of the

longest chain. Describe a procedure that selects a key uniformly at random from

among the keys in the hash table and returns it in expected time O.L .1 C 1=˛//.

262 Chapter 11 Hash Tables

11.3 Hash functions

In this section, we discuss some issues regarding the design of good hash functions

and then present three schemes for their creation. Two of the schemes, hashing by

division and hashing by multiplication, are heuristic in nature, whereas the third

scheme, universal hashing, uses randomization to provide provably good perfor-

mance.

What makes a good hash function?

A good hash function satisﬁes (approximately) the assumption of simple uniform

hashing: each key is equally likely to hash to any of the m slots, independently of

where any other key has hashed to. Unfortunately, we typically have no way to

check this condition, since we rarely know the probability distribution from which

the keys are drawn. Moreover, the keys might not be drawn independently.

Occasionally we do know the distribution. For example, if we know that the

keys are random real numbers k independently and uniformly distributed in the

range 0 k < 1, then the hash function

h.k/ D bkmc

satisﬁes the condition of simple uniform hashing.

In practice, we can often employ heuristic techniques to create a hash function

that performs well. Qualitative information about the distribution of keys may be

useful in this design process. For example, consider a compiler’s symbol table, in

which the keys are character strings representing identiﬁers in a program. Closely

related symbols, such as pt and pts, often occur in the same program. A good

hash function would minimize the chance that such variants hash to the same slot.

A good approach derives the hash value in a way that we expect to be indepen-

dent of any patterns that might exist in the data. For example, the “division method”

(discussed in Section 11.3.1) computes the hash value as the remainder when the

key is divided by a speciﬁed prime number. This method frequently gives good

results, assuming that we choose a prime number that is unrelated to any patterns

in the distribution of keys.

Finally, we note that some applications of hash functions might require stronger

properties than are provided by simple uniform hashing. For example, we might

want keys that are “close” in some sense to yield hash values that are far apart.

(This property is especially desirable when we are using linear probing, deﬁned in

Section 11.4.) Universal hashing, described in Section 11.3.3, often provides the

desired properties.

11.3 Hash functions 263

Interpreting keys as natural numbers

Most hash functions assume that the universe of keys is the set N D f0; 1; 2; : : :g

of natural numbers. Thus, if the keys are not natural numbers, we ﬁnd a way to

interpret them as natural numbers. For example, we can interpret a character string

as an integer expressed in suitable radix notation. Thus, we might interpret the

identiﬁer pt as the pair of decimal integers .112; 116/, since p D 112 and t D 116

in the ASCII character set; then, expressed as a radix-128 integer, pt becomes

.112 128/ C 116 D 14452. In the context of a given application, we can usually

devise some such method for interpreting each key as a (possibly large) natural

number. In what follows, we assume that the keys are natural numbers.

11.3.1 The division method

In the division method for creating hash functions, we map a key k into one of m

slots by taking the remainder of k divided by m. That is, the hash function is

h.k/ D k mod m :

For example, if the hash table has size m D 12 and the key is k D 100, then

h.k/ D 4. Since it requires only a single division operation, hashing by division is

quite fast.

When using the division method, we usually avoid certain values of m. For

example, m should not be a power of 2, since if m D 2 p

, then h.k/ is just the p

lowest-order bits of k. Unless we know that all low-order p-bit patterns are equally

likely, we are better off designing the hash function to depend on all the bits of the

key. As Exercise 11.3-3 asks you to show, choosing m D 2 p 1 when k is a

character string interpreted in radix 2 p

may be a poor choice, because permuting

the characters of k does not change its hash value.

A prime not too close to an exact power of 2 is often a good choice for m. For

example, suppose we wish to allocate a hash table, with collisions resolved by

chaining, to hold roughly n D 2000 character strings, where a character has 8 bits.

We don’t mind examining an average of 3 elements in an unsuccessful search, and

so we allocate a hash table of size m D 701. We could choose m D 701 because

it is a prime near 2000=3 but not near any power of 2. Treating each key k as an

integer, our hash function would be

h.k/ D k mod 701 :

11.3.2 The multiplication method

The multiplication method for creating hash functions operates in two steps. First,

we multiply the key k by a constant A in the range 0 < A < 1 and extract the

264 Chapter 11 Hash Tables

× s D A 2 w

w bits

k

r 0 r 1

h.k/

extract p bits

Figure 11.4 The multiplication method of hashing. The w-bit representation of the key k is multi-

plied by the w-bit value s D A 2

w

. The p highest-order bits of the lower w-bit half of the product

form the desired hash value h.k/.

fractional part of kA. Then, we multiply this value by m and take the ﬂoor of the

result. In short, the hash function is

h.k/ D bm .kA mod 1/c ;

where “kA mod 1” means the fractional part of kA, that is, kA bkAc.

An advantage of the multiplication method is that the value of m is not critical.

We typically choose it to be a power of 2 (m D 2 p

for some integer p), since we

can then easily implement the function on most computers as follows. Suppose

that the word size of the machine is w bits and that k ﬁts into a single word. We

restrict A to be a fraction of the form s=2 w

, where s is an integer in the range

0 < s < 2 w

. Referring to Figure 11.4, we ﬁrst multiply k by the w-bit integer

s D A 2 w

. The result is a 2w-bit value r 1 2 w C r 0 , where r 1 is the high-order word

of the product and r 0 is the low-order word of the product. The desired p-bit hash

value consists of the p most signiﬁcant bits of r 0 .

Although this method works with any value of the constant A, it works better

with some values than with others. The optimal choice depends on the character-

istics of the data being hashed. Knuth [211] suggests that

A .

p

5 1/=2 D 0:6180339887 : : : (11.2)

is likely to work reasonably well.

As an example, suppose we have k D 123456, p D 14, m D 2 14 D 16384,

and w D 32. Adapting Knuth’s suggestion, we choose A to be the fraction of the

form s=2 32

that is closest to .

p

5 1/=2, so that A D 2654435769=2 32

. Then

k s D 327706022297664 D .76300 2 32 / C 17612864, and so r 1 D 76300

and r 0 D 17612864. The 14 most signiﬁcant bits of r 0 yield the value h.k/ D 67.

11.3 Hash functions 265

? 11.3.3 Universal hashing

If a malicious adversary chooses the keys to be hashed by some ﬁxed hash function,

then the adversary can choose n keys that all hash to the same slot, yielding an av-

erage retrieval time of ‚.n/. Any ﬁxed hash function is vulnerable to such terrible

worst-case behavior; the only effective way to improve the situation is to choose

the hash function randomly in a way that is independent of the keys that are actually

going to be stored. This approach, called universal hashing, can yield provably

good performance on average, no matter which keys the adversary chooses.

In universal hashing, at the beginning of execution we select the hash function

at random from a carefully designed class of functions. As in the case of quick-

sort, randomization guarantees that no single input will always evoke worst-case

behavior. Because we randomly select the hash function, the algorithm can be-

have differently on each execution, even for the same input, guaranteeing good

average-case performance for any input. Returning to the example of a compiler’s

symbol table, we ﬁnd that the programmer’s choice of identiﬁers cannot now cause

consistently poor hashing performance. Poor performance occurs only when the

compiler chooses a random hash function that causes the set of identiﬁers to hash

poorly, but the probability of this situation occurring is small and is the same for

any set of identiﬁers of the same size.

Let H be a ﬁnite collection of hash functions that map a given universe U of

keys into the range f0; 1; : : : ; m 1g. Such a collection is said to be universal

if for each pair of distinct keys k; l 2 U , the number of hash functions h 2 H

for which h.k/ D h.l/ is at most j H j =m. In other words, with a hash function

randomly chosen from H , the chance of a collision between distinct keys k and l

is no more than the chance 1=m of a collision if h.k/ and h.l/ were randomly and

independently chosen from the set f0; 1; : : : ; m 1g.

The following theorem shows that a universal class of hash functions gives good

average-case behavior. Recall that n i denotes the length of list T Œi.

Theorem 11.3

Suppose that a hash function h is chosen randomly from a universal collection of

hash functions and has been used to hash n keys into a table T of size m, us-

ing chaining to resolve collisions. If key k is not in the table, then the expected

length E Œn h.k/ of the list that key k hashes to is at most the load factor ˛ D n=m.

If key k is in the table, then the expected length E Œn h.k/ of the list containing key k

is at most 1 C ˛.

Proof We note that the expectations here are over the choice of the hash func-

tion and do not depend on any assumptions about the distribution of the keys.

For each pair k and l of distinct keys, deﬁne the indicator random variable

266 Chapter 11 Hash Tables

X kl D I fh.k/ D h.l/g. Since by the deﬁnition of a universal collection of hash

functions, a single pair of keys collides with probability at most 1=m, we have

Pr fh.k/ D h.l/g 1=m. By Lemma 5.1, therefore, we have E ŒX kl 1=m.

Next we deﬁne, for each key k, the random variable Y k that equals the number

of keys other than k that hash to the same slot as k, so that

Y k D

X

l2T

l¤k

X kl :

Thus we have

E ŒY k D E

2

4

X

l2T

l¤k

X kl

3

5

D

X

l2T

l¤k

E ŒX kl (by linearity of expectation)

X

l2T

l¤k

1

m

:

The remainder of the proof depends on whether key k is in table T .

If k 62 T , then n h.k/ D Y k and jfl W l 2 T and l ¤ kgj D n. Thus E Œn h.k/ D

E ŒY k n=m D ˛.

If k 2 T , then because key k appears in list T Œh.k/ and the count Y k does not

include key k, we have n h.k/ D Y k C 1 and jfl W l 2 T and l ¤ kgj D n 1.

Thus E Œn h.k/ D E ŒY k C 1 .n 1/=m C 1 D 1 C ˛ 1=m < 1 C ˛.

The following corollary says universal hashing provides the desired payoff: it

has now become impossible for an adversary to pick a sequence of operations that

forces the worst-case running time. By cleverly randomizing the choice of hash

function at run time, we guarantee that we can process every sequence of operations

with a good average-case running time.

Corollary 11.4

Using universal hashing and collision resolution by chaining in an initially empty

table with m slots, it takes expected time ‚.n/ to handle any sequence of n INSERT,

SEARCH, and DELETE operations containing O.m/ INSERT operations.

Proof Since the number of insertions is O.m/, we have n D O.m/ and so

˛ D O.1/. The INSERT and DELETE operations take constant time and, by The-

orem 11.3, the expected time for each SEARCH operation is O.1/. By linearity of

11.3 Hash functions 267

expectation, therefore, the expected time for the entire sequence of n operations

is O.n/. Since each operation takes .1/ time, the ‚.n/ bound follows.

Designing a universal class of hash functions

It is quite easy to design a universal class of hash functions, as a little number

theory will help us prove. You may wish to consult Chapter 31 ﬁrst if you are

unfamiliar with number theory.

We begin by choosing a prime number p large enough so that every possible

key k is in the range 0 to p 1, inclusive. Let Z p denote the set f0; 1; : : : ; p 1g,

and let Z

p

denote the set f1; 2; : : : ; p 1g. Since p is prime, we can solve equa-

tions modulo p with the methods given in Chapter 31. Because we assume that the

size of the universe of keys is greater than the number of slots in the hash table, we

have p > m.

We now deﬁne the hash function h ab for any a 2 Z

p

and any b 2 Z p using a

linear transformation followed by reductions modulo p and then modulo m:

h ab .k/ D ..ak C b/ mod p/ mod m : (11.3)

For example, with p D 17 and m D 6, we have h 3;4 .8/ D 5. The family of all

such hash functions is

H pm D

˚

h ab W a 2 Z

p

and b 2 Z p

: (11.4)

Each hash function h ab maps Z p to Z m . This class of hash functions has the nice

property that the size m of the output range is arbitrary—not necessarily prime—a

feature which we shall use in Section 11.5. Since we have p 1 choices for a

and p choices for b, the collection H pm contains p.p 1/ hash functions.

Theorem 11.5

The class H pm of hash functions deﬁned by equations (11.3) and (11.4) is universal.

Proof Consider two distinct keys k and l from Z p , so that k ¤ l. For a given

hash function h ab we let

r D .ak C b/ mod p ;

s D .al C b/ mod p :

We ﬁrst note that r ¤ s. Why? Observe that

r s a.k l/ .mod p/ :

It follows that r ¤ s because p is prime and both a and .k l/ are nonzero

modulo p, and so their product must also be nonzero modulo p by Theorem 31.6.

Therefore, when computing any h ab 2 H pm , distinct inputs k and l map to distinct

268 Chapter 11 Hash Tables

values r and s modulo p; there are no collisions yet at the “mod p level.” Moreover,

each of the possible p.p1/ choices for the pair .a; b/ with a ¤ 0 yields a different

resulting pair .r; s/ with r ¤ s, since we can solve for a and b given r and s:

a D

.r s/..k l/

1

mod p/

mod p ;

b D .r ak/ mod p ;

where ..k l/ 1

mod p/ denotes the unique multiplicative inverse, modulo p,

of k l. Since there are only p.p 1/ possible pairs .r; s/ with r ¤ s, there

is a one-to-one correspondence between pairs .a; b/ with a ¤ 0 and pairs .r; s/

with r ¤ s. Thus, for any given pair of inputs k and l, if we pick .a; b/ uniformly

at random from Z

p

Z p , the resulting pair .r; s/ is equally likely to be any pair of

distinct values modulo p.

Therefore, the probability that distinct keys k and l collide is equal to the prob-

ability that r s .mod m/ when r and s are randomly chosen as distinct values

modulo p. For a given value of r, of the p 1 possible remaining values for s, the

number of values s such that s ¤ r and s r .mod m/ is at most

dp=me 1 ..p C m 1/=m/ 1 (by inequality (3.6))

D .p 1/=m :

The probability that s collides with r when reduced modulo m is at most

..p 1/=m/=.p 1/ D 1=m.

Therefore, for any pair of distinct values k; l 2 Z p ,

Pr fh ab .k/ D h ab .l/g 1=m ;

so that H pm is indeed universal.

Exercises

11.3-1

Suppose we wish to search a linked list of length n, where each element contains

a key k along with a hash value h.k/. Each key is a long character string. How

might we take advantage of the hash values when searching the list for an element

with a given key?

11.3-2

Suppose that we hash a string of r characters into m slots by treating it as a

radix-128 number and then using the division method. We can easily represent

the number m as a 32-bit computer word, but the string of r characters, treated as

a radix-128 number, takes many words. How can we apply the division method to

compute the hash value of the character string without using more than a constant

number of words of storage outside the string itself?

11.4 Open addressing 269

11.3-3

Consider a version of the division method in which h.k/ D k mod m, where

m D 2 p 1 and k is a character string interpreted in radix 2 p

. Show that if we

can derive string x from string y by permuting its characters, then x and y hash to

the same value. Give an example of an application in which this property would be

undesirable in a hash function.

11.3-4

Consider a hash table of size m D 1000 and a corresponding hash function h.k/ D

bm .kA mod 1/c for A D .

p

5 1/=2. Compute the locations to which the keys

61, 62, 63, 64, and 65 are mapped.

11.3-5 ?

Deﬁne a family H of hash functions from a ﬁnite set U to a ﬁnite set B to be

-universal if for all pairs of distinct elements k and l in U ,

Pr fh.k/ D h.l/g ;

where the probability is over the choice of the hash function h drawn at random

from the family H . Show that an -universal family of hash functions must have

1

jBj

1

jU j

:

11.3-6 ?

Let U be the set of n-tuples of values drawn from Z p , and let B D Z p , where p

is prime. Deﬁne the hash function h b W U ! B for b 2 Z p on an input n-tuple

ha 0 ; a 1 ; : : : ; a n1 i from U as

h b .ha 0 ; a 1 ; : : : ; a n1 i/ D

n1 X

j D0

a j b

j

!

mod p ;

and let H D fh b W b 2 Z p g. Argue that H is ..n 1/=p/-universal according to

the deﬁnition of -universal in Exercise 11.3-5. (Hint: See Exercise 31.4-4.)

11.4 Open addressing

In open addressing, all elements occupy the hash table itself. That is, each table

entry contains either an element of the dynamic set or NIL. When searching for

an element, we systematically examine table slots until either we ﬁnd the desired

element or we have ascertained that the element is not in the table. No lists and

270 Chapter 11 Hash Tables

no elements are stored outside the table, unlike in chaining. Thus, in open ad-

dressing, the hash table can “ﬁll up” so that no further insertions can be made; one

consequence is that the load factor ˛ can never exceed 1.

Of course, we could store the linked lists for chaining inside the hash table, in

the otherwise unused hash-table slots (see Exercise 11.2-4), but the advantage of

open addressing is that it avoids pointers altogether. Instead of following pointers,

we compute the sequence of slots to be examined. The extra memory freed by not

storing pointers provides the hash table with a larger number of slots for the same

amount of memory, potentially yielding fewer collisions and faster retrieval.

To perform insertion using open addressing, we successively examine, or probe,

the hash table until we ﬁnd an empty slot in which to put the key. Instead of being

ﬁxed in the order 0; 1; : : : ; m 1 (which requires ‚.n/ search time), the sequence

of positions probed depends upon the key being inserted. To determine which slots

to probe, we extend the hash function to include the probe number (starting from 0)

as a second input. Thus, the hash function becomes

h W U f0; 1; : : : ; m 1g ! f0; 1; : : : ; m 1g :

With open addressing, we require that for every key k, the probe sequence

hh.k; 0/; h.k; 1/; : : : ; h.k; m 1/i

be a permutation of h0;1;: : : ;m1i, so that every hash-table position is eventually

considered as a slot for a new key as the table ﬁlls up. In the following pseudocode,

we assume that the elements in the hash table T are keys with no satellite infor-

mation; the key k is identical to the element containing key k. Each slot contains

either a key or NIL (if the slot is empty). The HASH-INSERT procedure takes as

input a hash table T and a key k. It either returns the slot number where it stores

key k or ﬂags an error because the hash table is already full.

HASH-INSERT.T; k/

1 i D 0

2 repeat

3 j D h.k; i/

4 if T Œj == NIL

5 T Œj D k

6 return j

7 else i D i C 1

8 until i == m

9 error “hash table overﬂow”

The algorithm for searching for key k probes the same sequence of slots that the

insertion algorithm examined when key k was inserted. Therefore, the search can

11.4 Open addressing 271

terminate (unsuccessfully) when it ﬁnds an empty slot, since k would have been

inserted there and not later in its probe sequence. (This argument assumes that keys

are not deleted from the hash table.) The procedure HASH-SEARCH takes as input

a hash table T and a key k, returning j if it ﬁnds that slot j contains key k, or NIL

if key k is not present in table T .

HASH-SEARCH.T; k/

1 i D 0

2 repeat

3 j D h.k; i/

4 if T Œj == k

5 return j

6 i D i C 1

7 until T Œj == NIL or i == m

8 return NIL

Deletion from an open-address hash table is difﬁcult. When we delete a key

from slot i, we cannot simply mark that slot as empty by storing NIL in it. If

we did, we might be unable to retrieve any key k during whose insertion we had

probed slot i and found it occupied. We can solve this problem by marking the

slot, storing in it the special value DELETED instead of NIL. We would then modify

the procedure HASH-INSERT to treat such a slot as if it were empty so that we can

insert a new key there. We do not need to modify HASH-SEARCH, since it will pass

over DELETED values while searching. When we use the special value DELETED,

however, search times no longer depend on the load factor ˛, and for this reason

chaining is more commonly selected as a collision resolution technique when keys

must be deleted.

In our analysis, we assume uniform hashing: the probe sequence of each key

is equally likely to be any of the mŠ permutations of h0; 1; : : : ; m 1i. Uni-

form hashing generalizes the notion of simple uniform hashing deﬁned earlier to a

hash function that produces not just a single number, but a whole probe sequence.

True uniform hashing is difﬁcult to implement, however, and in practice suitable

approximations (such as double hashing, deﬁned below) are used.

We will examine three commonly used techniques to compute the probe se-

quences required for open addressing: linear probing, quadratic probing, and dou-

ble hashing. These techniques all guarantee that hh.k;0/;h.k;1/;: : : ;h.k;m 1/i

is a permutation of h0; 1; : : : ; m 1i for each key k. None of these techniques ful-

ﬁlls the assumption of uniform hashing, however, since none of them is capable of

generating more than m 2

different probe sequences (instead of the mŠ that uniform

hashing requires). Double hashing has the greatest number of probe sequences and,

as one might expect, seems to give the best results.

272 Chapter 11 Hash Tables

Linear probing

Given an ordinary hash function h 0 W U ! f0; 1; : : : ; m 1g, which we refer to as

an auxiliary hash function, the method of linear probing uses the hash function

h.k; i/ D .h

0

.k/ C i/ mod m

for i D 0; 1; : : : ; m 1. Given key k, we ﬁrst probe T Œh 0 .k/, i.e., the slot given

by the auxiliary hash function. We next probe slot T Œh 0 .k/ C 1, and so on up to

slot T Œm 1. Then we wrap around to slots T Œ0; T Œ1; : : : until we ﬁnally probe

slot T Œh 0 .k/ 1. Because the initial probe determines the entire probe sequence,

there are only m distinct probe sequences.

Linear probing is easy to implement, but it suffers from a problem known as

primary clustering. Long runs of occupied slots build up, increasing the average

search time. Clusters arise because an empty slot preceded by i full slots gets ﬁlled

next with probability .i C 1/=m. Long runs of occupied slots tend to get longer,

and the average search time increases.

Quadratic probing

Quadratic probing uses a hash function of the form

h.k; i/ D .h

0

.k/ C c 1 i C c 2 i

2

/ mod m ; (11.5)

where h 0

is an auxiliary hash function, c 1 and c 2 are positive auxiliary constants,

and i D 0; 1; : : : ; m 1. The initial position probed is T Œh 0 .k/; later positions

probed are offset by amounts that depend in a quadratic manner on the probe num-

ber i. This method works much better than linear probing, but to make full use of

the hash table, the values of c 1 , c 2 , and m are constrained. Problem 11-3 shows

one way to select these parameters. Also, if two keys have the same initial probe

position, then their probe sequences are the same, since h.k 1 ; 0/ D h.k 2 ; 0/ im-

plies h.k 1 ; i/ D h.k 2 ; i/. This property leads to a milder form of clustering, called

secondary clustering. As in linear probing, the initial probe determines the entire

sequence, and so only m distinct probe sequences are used.

Double hashing

Double hashing offers one of the best methods available for open addressing be-

cause the permutations produced have many of the characteristics of randomly

chosen permutations. Double hashing uses a hash function of the form

h.k; i/ D .h 1 .k/ C ih 2 .k// mod m ;

where both h 1 and h 2 are auxiliary hash functions. The initial probe goes to posi-

tion T Œh 1 .k/; successive probe positions are offset from previous positions by the

11.4 Open addressing 273

0

1

2

3

4

5

6

7

8

9

10

11

12

79

69

98

72

14

50

Figure 11.5 Insertion by double hashing. Here we have a hash table of size 13 with h1.k/ D

k mod 13 and h2.k/ D 1 C .k mod 11/. Since 14 1 .mod 13/ and 14 3 .mod 11/, we insert

the key 14 into empty slot 9, after examining slots 1 and 5 and ﬁnding them to be occupied.

amount h 2 .k/, modulo m. Thus, unlike the case of linear or quadratic probing, the

probe sequence here depends in two ways upon the key k, since the initial probe

position, the offset, or both, may vary. Figure 11.5 gives an example of insertion

by double hashing.

The value h 2 .k/ must be relatively prime to the hash-table size m for the entire

hash table to be searched. (See Exercise 11.4-4.) A convenient way to ensure this

condition is to let m be a power of 2 and to design h 2 so that it always produces an

odd number. Another way is to let m be prime and to design h 2 so that it always

returns a positive integer less than m. For example, we could choose m prime and

let

h 1 .k/ D k mod m ;

h 2 .k/ D 1 C .k mod m

0

/ ;

where m 0

is chosen to be slightly less than m (say, m 1). For example, if

k D 123456, m D 701, and m 0 D 700, we have h 1 .k/ D 80 and h 2 .k/ D 257, so

that we ﬁrst probe position 80, and then we examine every 257th slot (modulo m)

until we ﬁnd the key or have examined every slot.

When m is prime or a power of 2, double hashing improves over linear or qua-

dratic probing in that ‚.m 2 / probe sequences are used, rather than ‚.m/, since

each possible .h 1 .k/; h 2 .k// pair yields a distinct probe sequence. As a result, for

274 Chapter 11 Hash Tables

such values of m, the performance of double hashing appears to be very close to

the performance of the “ideal” scheme of uniform hashing.

Although values of m other than primes or powers of 2 could in principle be

used with double hashing, in practice it becomes more difﬁcult to efﬁciently gen-

erate h 2 .k/ in a way that ensures that it is relatively prime to m, in part because the

relative density .m/=m of such numbers may be small (see equation (31.24)).

Analysis of open-address hashing

As in our analysis of chaining, we express our analysis of open addressing in terms

of the load factor ˛ D n=m of the hash table. Of course, with open addressing, at

most one element occupies each slot, and thus n m, which implies ˛ 1.

We assume that we are using uniform hashing. In this idealized scheme, the

probe sequence hh.k; 0/; h.k; 1/; : : : ; h.k; m 1/i used to insert or search for

each key k is equally likely to be any permutation of h0; 1; : : : ; m 1i. Of course,

a given key has a unique ﬁxed probe sequence associated with it; what we mean

here is that, considering the probability distribution on the space of keys and the

operation of the hash function on the keys, each possible probe sequence is equally

likely.

We now analyze the expected number of probes for hashing with open address-

ing under the assumption of uniform hashing, beginning with an analysis of the

number of probes made in an unsuccessful search.

Theorem 11.6

Given an open-address hash table with load factor ˛ D n=m < 1, the expected

number of probes in an unsuccessful search is at most 1=.1˛/, assuming uniform

hashing.

Proof In an unsuccessful search, every probe but the last accesses an occupied

slot that does not contain the desired key, and the last slot probed is empty. Let us

deﬁne the random variable X to be the number of probes made in an unsuccessful

search, and let us also deﬁne the event A i , for i D 1; 2; : : :, to be the event that

an ith probe occurs and it is to an occupied slot. Then the event fX ig is the

intersection of events A 1 \A 2 \ \A i1 . We will bound Pr fX ig by bounding

Pr fA 1 \ A 2 \ \ A i1 g. By Exercise C.2-5,

Pr fA 1 \ A 2 \ \ A i1 g D Pr fA 1 g Pr fA 2 j A 1 g Pr fA 3 j A 1 \ A 2 g

Pr fA i1 j A 1 \ A 2 \ \ A i2 g :

Since there are n elements and m slots, Pr fA 1 g D n=m. For j > 1, the probability

that there is a j th probe and it is to an occupied slot, given that the ﬁrst j 1

probes were to occupied slots, is .nj C1/=.mj C1/. This probability follows

11.4 Open addressing 275

because we would be ﬁnding one of the remaining .n .j 1// elements in one

of the .m .j 1// unexamined slots, and by the assumption of uniform hashing,

the probability is the ratio of these quantities. Observing that n < m implies that

.n j /=.m j / n=m for all j such that 0 j < m, we have for all i such that

1 i m,

Pr fX ig D

n

m

n 1

m 1

n 2

m 2

n i C 2

m i C 2

n

m

i1

D ˛

i1

:

Now, we use equation (C.25) to bound the expected number of probes:

E ŒX D

1 X

iD1

Pr fX ig

1 X

iD1

˛

i1

D

1 X

iD0

˛

i

D

1

1 ˛

:

This bound of 1=.1 ˛/ D 1C ˛ C ˛ 2 C ˛ 3 C has an intuitive interpretation.

We always make the ﬁrst probe. With probability approximately ˛, the ﬁrst probe

ﬁnds an occupied slot, so that we need to probe a second time. With probability

approximately ˛ 2

, the ﬁrst two slots are occupied so that we make a third probe,

and so on.

If ˛ is a constant, Theorem 11.6 predicts that an unsuccessful search runs in O.1/

time. For example, if the hash table is half full, the average number of probes in an

unsuccessful search is at most 1=.1 :5/ D 2. If it is 90 percent full, the average

number of probes is at most 1=.1 :9/ D 10.

Theorem 11.6 gives us the performance of the HASH-INSERT procedure almost

immediately.

Corollary 11.7

Inserting an element into an open-address hash table with load factor ˛ requires at

most 1=.1 ˛/ probes on average, assuming uniform hashing.

276 Chapter 11 Hash Tables

Proof An element is inserted only if there is room in the table, and thus ˛ < 1.

Inserting a key requires an unsuccessful search followed by placing the key into the

ﬁrst empty slot found. Thus, the expected number of probes is at most 1=.1˛/.

We have to do a little more work to compute the expected number of probes for

a successful search.

Theorem 11.8

Given an open-address hash table with load factor ˛ < 1, the expected number of

probes in a successful search is at most

1

˛

ln

1

1 ˛

;

assuming uniform hashing and assuming that each key in the table is equally likely

to be searched for.

Proof A search for a key k reproduces the same probe sequence as when the

element with key k was inserted. By Corollary 11.7, if k was the .i C 1/st key

inserted into the hash table, the expected number of probes made in a search for k

is at most 1=.1 i=m/ D m=.m i/. Averaging over all n keys in the hash table

gives us the expected number of probes in a successful search:

1

n

n1 X

iD0

m

m i

D

m

n

n1 X

iD0

1

m i

D

1

˛

m X

kDmnC1

1

k

1

˛

Z m

mn

.1=x/ dx (by inequality (A.12))

D

1

˛

ln

m

m n

D

1

˛

ln

1

1 ˛

:

If the hash table is half full, the expected number of probes in a successful search

is less than 1:387. If the hash table is 90 percent full, the expected number of probes

is less than 2:559.

11.5 Perfect hashing 277

Exercises

11.4-1

Consider inserting the keys 10; 22; 31; 4; 15; 28; 17; 88; 59 into a hash table of

length m D 11 using open addressing with the auxiliary hash function h 0 .k/ D k.

Illustrate the result of inserting these keys using linear probing, using quadratic

probing with c 1 D 1 and c 2 D 3, and using double hashing with h 1 .k/ D k and

h 2 .k/ D 1 C .k mod .m 1//.

11.4-2

Write pseudocode for HASH-DELETE as outlined in the text, and modify HASH-

INSERT to handle the special value DELETED.

11.4-3

Consider an open-address hash table with uniform hashing. Give upper bounds

on the expected number of probes in an unsuccessful search and on the expected

number of probes in a successful search when the load factor is 3=4 and when it

is 7=8.

11.4-4 ?

Suppose that we use double hashing to resolve collisions—that is, we use the hash

function h.k; i/ D .h 1 .k/ C ih 2 .k// mod m. Show that if m and h 2 .k/ have

greatest common divisor d 1 for some key k, then an unsuccessful search for

key k examines .1=d/th of the hash table before returning to slot h 1 .k/. Thus,

when d D 1, so that m and h 2 .k/ are relatively prime, the search may examine the

entire hash table. (Hint: See Chapter 31.)

11.4-5 ?

Consider an open-address hash table with a load factor ˛. Find the nonzero value ˛

for which the expected number of probes in an unsuccessful search equals twice

the expected number of probes in a successful search. Use the upper bounds given

by Theorems 11.6 and 11.8 for these expected numbers of probes.

? 11.5 Perfect hashing

Although hashing is often a good choice for its excellent average-case perfor-

mance, hashing can also provide excellent worst-case performance when the set of

keys is static: once the keys are stored in the table, the set of keys never changes.

Some applications naturally have static sets of keys: consider the set of reserved

words in a programming language, or the set of ﬁle names on a CD-ROM. We

278 Chapter 11 Hash Tables

0

1

2

3

4

5

6

7

8

1 0 0 10

9 10 18 60 75

0 1 2 3

1 0 0 70

0

0

16 23 88 40 37

0 1 2 3 4 5 6 7 8

52

m

2

S

2 a

2

b

2

m

0

S

0 a

0

b

0

m

5

S

5 a

5

b

5

m

7

S

7 a

7

b

7

T

4 5 6 7 8

72

9 10 11 12 13 14 15

22

Figure 11.6 Using perfect hashing to store the set K D f10; 22; 37; 40; 52; 60; 70; 72; 75g. The

outer hash function is h.k/ D ..ak C b/ mod p/ mod m, where a D 3, b D 42, p D 101, and

m D 9. For example, h.75/ D 2, and so key 75 hashes to slot 2 of table T . A secondary hash

table Sj stores all keys hashing to slot j . The size of hash table Sj is mj D n

2

j

, and the associated

hash function is hj .k/ D ..aj k C bj / mod p/ mod mj . Since h2.75/ D 7, key 75 is stored in slot 7

of secondary hash table S2. No collisions occur in any of the secondary hash tables, and so searching

takes constant time in the worst case.

call a hashing technique perfect hashing if O.1/ memory accesses are required to

perform a search in the worst case.

To create a perfect hashing scheme, we use two levels of hashing, with universal

hashing at each level. Figure 11.6 illustrates the approach.

The ﬁrst level is essentially the same as for hashing with chaining: we hash

the n keys into m slots using a hash function h carefully selected from a family of

universal hash functions.

Instead of making a linked list of the keys hashing to slot j , however, we use a

small secondary hash table S j with an associated hash function h j . By choosing

the hash functions h j carefully, we can guarantee that there are no collisions at the

secondary level.

In order to guarantee that there are no collisions at the secondary level, however,

we will need to let the size m j of hash table S j be the square of the number n j of

keys hashing to slot j . Although you might think that the quadratic dependence

of m j on n j may seem likely to cause the overall storage requirement to be exces-

sive, we shall show that by choosing the ﬁrst-level hash function well, we can limit

the expected total amount of space used to O.n/.

We use hash functions chosen from the universal classes of hash functions of

Section 11.3.3. The ﬁrst-level hash function comes from the class H pm , where as

in Section 11.3.3, p is a prime number greater than any key value. Those keys

11.5 Perfect hashing 279

hashing to slot j are re-hashed into a secondary hash table S j of size m j using a

hash function h j chosen from the class H p;m j . 1

We shall proceed in two steps. First, we shall determine how to ensure that

the secondary tables have no collisions. Second, we shall show that the expected

amount of memory used overall—for the primary hash table and all the secondary

hash tables—is O.n/.

Theorem 11.9

Suppose that we store n keys in a hash table of size m D n 2

using a hash function h

randomly chosen from a universal class of hash functions. Then, the probability is

less than 1=2 that there are any collisions.

Proof There are

n

2

pairs of keys that may collide; each pair collides with prob-

ability 1=m if h is chosen at random from a universal family H of hash functions.

Let X be a random variable that counts the number of collisions. When m D n 2

,

the expected number of collisions is

E ŒX D

n

2

!

1

n 2

D

n 2 n

2

1

n 2

< 1=2 :

(This analysis is similar to the analysis of the birthday paradox in Section 5.4.1.)

Applying Markov’s inequality (C.30), Pr fX tg E ŒX =t, with t D 1, com-

pletes the proof.

In the situation described in Theorem 11.9, where m D n 2

, it follows that a hash

function h chosen at random from H is more likely than not to have no collisions.

Given the set K of n keys to be hashed (remember that K is static), it is thus easy

to ﬁnd a collision-free hash function h with a few random trials.

When n is large, however, a hash table of size m D n 2

is excessive. Therefore,

we adopt the two-level hashing approach, and we use the approach of Theorem 11.9

only to hash the entries within each slot. We use an outer, or ﬁrst-level, hash

function h to hash the keys into m D n slots. Then, if n j keys hash to slot j , we

use a secondary hash table S j of size m j D n 2

j

to provide collision-free constant-

time lookup.

1

When nj D mj D 1, we don’t really need a hash function for slot j ; when we choose a hash

function hab.k/ D ..ak C b/ mod p/ mod mj for such a slot, we just use a D b D 0.

280 Chapter 11 Hash Tables

We now turn to the issue of ensuring that the overall memory used is O.n/.

Since the size m j of the j th secondary hash table grows quadratically with the

number n j of keys stored, we run the risk that the overall amount of storage could

be excessive.

If the ﬁrst-level table size is m D n, then the amount of memory used is O.n/

for the primary hash table, for the storage of the sizes m j of the secondary hash

tables, and for the storage of the parameters a j and b j deﬁning the secondary hash

functions h j drawn from the class H p;m j of Section 11.3.3 (except when n j D 1

and we use a D b D 0). The following theorem and a corollary provide a bound on

the expected combined sizes of all the secondary hash tables. A second corollary

bounds the probability that the combined size of all the secondary hash tables is

superlinear (actually, that it equals or exceeds 4n).

Theorem 11.10

Suppose that we store n keys in a hash table of size m D n using a hash function h

randomly chosen from a universal class of hash functions. Then, we have

E

"

m1 X

j D0

n

2

j

#

< 2n ;

where n j is the number of keys hashing to slot j .

Proof We start with the following identity, which holds for any nonnegative inte-

ger a:

a

2

D a C 2

a

2

!

: (11.6)

We have

E

"

m1 X

j D0

n

2

j

#

D E

"

m1 X

j D0

n j C 2

n j

2

!!#

(by equation (11.6))

D E

"

m1 X

j D0

n j

#

C 2 E

"

m1 X

j D0

n j

2

!#

(by linearity of expectation)

D E Œn C 2 E

"

m1 X

j D0

n j

2

!#

(by equation (11.1))

11.5 Perfect hashing 281

D n C 2 E

"

m1 X

j D0

n j

2

!#

(since n is not a random variable) .

To evaluate the summation

P m1

j D0

n j

2

, we observe that it is just the total number

of pairs of keys in the hash table that collide. By the properties of universal hashing,

the expected value of this summation is at most

n

2

!

1

m

D

n.n 1/

2m

D

n 1

2

;

since m D n. Thus,

E

"

m1 X

j D0

n

2

j

#

n C 2

n 1

2

D 2n 1

< 2n :

Corollary 11.11

Suppose that we store n keys in a hash table of size m D n using a hash func-

tion h randomly chosen from a universal class of hash functions, and we set the

size of each secondary hash table to m j D n 2

j

for j D 0; 1; : : : ; m 1. Then,

the expected amount of storage required for all secondary hash tables in a perfect

hashing scheme is less than 2n.

Proof Since m j D n 2

j

for j D 0; 1; : : : ; m 1, Theorem 11.10 gives

E

"

m1 X

j D0

m j

#

D E

"

m1 X

j D0

n

2

j

#

< 2n ; (11.7)

which completes the proof.

Corollary 11.12

Suppose that we store n keys in a hash table of size m D n using a hash function h

randomly chosen from a universal class of hash functions, and we set the size

of each secondary hash table to m j D n 2

j

for j D 0; 1; : : : ; m 1. Then, the

probability is less than 1=2 that the total storage used for secondary hash tables

equals or exceeds 4n.

282 Chapter 11 Hash Tables

Proof Again we apply Markov’s inequality (C.30), Pr fX tg E ŒX =t, this

time to inequality (11.7), with X D

P m1

j D0

m j and t D 4n:

Pr

(

m1 X

j D0

m j 4n

)

E

P m1

j D0

m j

4n

<

2n

4n

D 1=2 :

From Corollary 11.12, we see that if we test a few randomly chosen hash func-

tions from the universal family, we will quickly ﬁnd one that uses a reasonable

amount of storage.

Exercises

11.5-1 ?

Suppose that we insert n keys into a hash table of size m using open addressing

and uniform hashing. Let p.n; m/ be the probability that no collisions occur. Show

that p.n; m/ e n.n1/=2m

. (Hint: See equation (3.12).) Argue that when n ex-

ceeds

p

m, the probability of avoiding collisions goes rapidly to zero.

Problems

11-1 Longest-probe bound for hashing

Suppose that we use an open-addressed hash table of size m to store n m=2

items.

a. Assuming uniform hashing, show that for i D 1; 2; : : : ; n, the probability is at

most 2 k

that the ith insertion requires strictly more than k probes.

b. Show that for i D 1; 2; : : : ; n, the probability is O.1=n 2 / that the ith insertion

requires more than 2 lg n probes.

Let the random variable X i denote the number of probes required by the ith inser-

tion. You have shown in part (b) that Pr fX i > 2 lg ng D O.1=n 2 /. Let the random

variable X D max 1in X i denote the maximum number of probes required by

any of the n insertions.

c. Show that Pr fX > 2 lg ng D O.1=n/.

d. Show that the expected length E ŒX of the longest probe sequence is O.lg n/.

Problems for Chapter 11 283

11-2 Slot-size bound for chaining

Suppose that we have a hash table with n slots, with collisions resolved by chain-

ing, and suppose that n keys are inserted into the table. Each key is equally likely

to be hashed to each slot. Let M be the maximum number of keys in any slot after

all the keys have been inserted. Your mission is to prove an O.lg n= lg lg n/ upper

bound on E ŒM , the expected value of M .

a. Argue that the probability Q k that exactly k keys hash to a particular slot is

given by

Q k D

1

n

k

1

1

n

nk

n

k

!

:

b. Let P k be the probability that M D k, that is, the probability that the slot

containing the most keys contains k keys. Show that P k nQ k .

c. Use Stirling’s approximation, equation (3.18), to show that Q k < e k =k k

.

d. Show that there exists a constant c > 1 such that Q k 0 < 1=n 3

for k 0 D

c lg n= lg lg n. Conclude that P k < 1=n 2

for k k 0 D c lg n= lg lg n.

e. Argue that

E ŒM Pr

M >

c lg n

lg lg n

n C Pr

M

c lg n

lg lg n

c lg n

lg lg n

:

Conclude that E ŒM D O.lg n= lg lg n/.

11-3 Quadratic probing

Suppose that we are given a key k to search for in a hash table with positions

0; 1; : : : ; m 1, and suppose that we have a hash function h mapping the key space

into the set f0; 1; : : : ; m 1g. The search scheme is as follows:

1. Compute the value j D h.k/, and set i D 0.

2. Probe in position j for the desired key k. If you ﬁnd it, or if this position is

empty, terminate the search.

3. Set i D i C 1. If i now equals m, the table is full, so terminate the search.

Otherwise, set j D .i C j / mod m, and return to step 2.

Assume that m is a power of 2.

a. Show that this scheme is an instance of the general “quadratic probing” scheme

by exhibiting the appropriate constants c 1 and c 2 for equation (11.5).

b. Prove that this algorithm examines every table position in the worst case.

284 Chapter 11 Hash Tables

11-4 Hashing and authentication

Let H be a class of hash functions in which each hash function h 2 H maps the

universe U of keys to f0; 1; : : : ; m 1g. We say that H is k-universal if, for every

ﬁxed sequence of k distinct keys hx .1/ ; x .2/ ; : : : ; x .k/ i and for any h chosen at

random from H , the sequence hh.x .1/ /;h.x .2/ /;: : : ;h.x .k/ /i is equally likely to be

any of the m k

sequences of length k with elements drawn from f0; 1; : : : ; m 1g.

a. Show that if the family H of hash functions is 2-universal, then it is universal.

b. Suppose that the universe U is the set of n-tuples of values drawn from

Z p D f0; 1; : : : ; p 1g, where p is prime. Consider an element x D

hx 0 ; x 1 ; : : : ; x n1 i 2 U . For any n-tuple a D ha 0 ; a 1 ; : : : ; a n1 i 2 U , de-

ﬁne the hash function h a by

h a .x/ D

n1 X

j D0

a j x j

!

mod p :

Let H D fh a g. Show that H is universal, but not 2-universal. (Hint: Find a key

for which all hash functions in H produce the same value.)

c. Suppose that we modify H slightly from part (b): for any a 2 U and for any

b 2 Z p , deﬁne

h

0

ab

.x/ D

n1 X

j D0

a j x j C b

!

mod p

and H 0 D fh 0

ab

g. Argue that H 0

is 2-universal. (Hint: Consider ﬁxed n-tuples

x 2 U and y 2 U , with x i ¤ y i for some i. What happens to h 0

ab

.x/

and h 0

ab

.y/ as a i and b range over Z p ?)

d. Suppose that Alice and Bob secretly agree on a hash function h from a

2-universal family H of hash functions. Each h 2 H maps from a universe of

keys U to Z p , where p is prime. Later, Alice sends a message m to Bob over the

Internet, where m 2 U . She authenticates this message to Bob by also sending

an authentication tag t D h.m/, and Bob checks that the pair .m; t/ he receives

indeed satisﬁes t D h.m/. Suppose that an adversary intercepts .m; t/ en route

and tries to fool Bob by replacing the pair .m; t/ with a different pair .m 0 ; t 0 /.

Argue that the probability that the adversary succeeds in fooling Bob into ac-

cepting .m 0 ; t 0 / is at most 1=p, no matter how much computing power the ad-

versary has, and even if the adversary knows the family H of hash functions

used.

Notes for Chapter 11 285

Chapter notes

Knuth [211] and Gonnet [145] are excellent references for the analysis of hash-

ing algorithms. Knuth credits H. P. Luhn (1953) for inventing hash tables, along

with the chaining method for resolving collisions. At about the same time, G. M.

Amdahl originated the idea of open addressing.

Carter and Wegman introduced the notion of universal classes of hash functions

in 1979 [58].

Fredman, Koml´os, and Szemer´edi [112] developed the perfect hashing scheme

for static sets presented in Section 11.5. An extension of their method to dynamic

sets, handling insertions and deletions in amortized expected time O.1/, has been

given by Dietzfelbinger et al. [86].

12 Binary Search Trees

The search tree data structure supports many dynamic-set operations, including

SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and

DELETE. Thus, we can use a search tree both as a dictionary and as a priority

queue.

Basic operations on a binary search tree take time proportional to the height of

the tree. For a complete binary tree with n nodes, such operations run in ‚.lg n/

worst-case time. If the tree is a linear chain of n nodes, however, the same oper-

ations take ‚.n/ worst-case time. We shall see in Section 12.4 that the expected

height of a randomly built binary search tree is O.lg n/, so that basic dynamic-set

operations on such a tree take ‚.lg n/ time on average.

In practice, we can’t always guarantee that binary search trees are built ran-

domly, but we can design variations of binary search trees with good guaranteed

worst-case performance on basic operations. Chapter 13 presents one such vari-

ation, red-black trees, which have height O.lg n/. Chapter 18 introduces B-trees,

which are particularly good for maintaining databases on secondary (disk) storage.

After presenting the basic properties of binary search trees, the following sec-

tions show how to walk a binary search tree to print its values in sorted order, how

to search for a value in a binary search tree, how to ﬁnd the minimum or maximum

element, how to ﬁnd the predecessor or successor of an element, and how to insert

into or delete from a binary search tree. The basic mathematical properties of trees

appear in Appendix B.

12.1 What is a binary search tree?

A binary search tree is organized, as the name suggests, in a binary tree, as shown

in Figure 12.1. We can represent such a tree by a linked data structure in which

each node is an object. In addition to a key and satellite data, each node contains

attributes left, right, and p that point to the nodes corresponding to its left child,

12.1 What is a binary search tree? 287

5

2 5

5

8

7

6

(a)

6 8

7

5

2

(b)

Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most x:key,

and the keys in the right subtree of x are at least x:key. Different binary search trees can represent

the same set of values. The worst-case running time for most search-tree operations is proportional

to the height of the tree. (a) A binary search tree on 6 nodes with height 2. (b) A less efﬁcient binary

search tree with height 4 that contains the same keys.

its right child, and its parent, respectively. If a child or the parent is missing, the

appropriate attribute contains the value NIL. The root node is the only node in the

tree whose parent is NIL.

The keys in a binary search tree are always stored in such a way as to satisfy the

binary-search-tree property:

Let x be a node in a binary search tree. If y is a node in the left subtree

of x, then y:key x:key. If y is a node in the right subtree of x, then

y:key x:key.

Thus, in Figure 12.1(a), the key of the root is 6, the keys 2, 5, and 5 in its left

subtree are no larger than 6, and the keys 7 and 8 in its right subtree are no smaller

than 6. The same property holds for every node in the tree. For example, the key 5

in the root’s left child is no smaller than the key 2 in that node’s left subtree and no

larger than the key 5 in the right subtree.

The binary-search-tree property allows us to print out all the keys in a binary

search tree in sorted order by a simple recursive algorithm, called an inorder tree

walk. This algorithm is so named because it prints the key of the root of a subtree

between printing the values in its left subtree and printing those in its right subtree.

(Similarly, a preorder tree walk prints the root before the values in either subtree,

and a postorder tree walk prints the root after the values in its subtrees.) To use

the following procedure to print all the elements in a binary search tree T , we call

INORDER-TREE-WALK.T:root/.

288 Chapter 12 Binary Search Trees

INORDER-TREE-WALK.x/

1 if x ¤ NIL

2 INORDER-TREE-WALK.x:left/

3 print x:key

4 INORDER-TREE-WALK.x:right/

As an example, the inorder tree walk prints the keys in each of the two binary

search trees from Figure 12.1 in the order 2; 5; 5; 6; 7; 8. The correctness of the

algorithm follows by induction directly from the binary-search-tree property.

It takes ‚.n/ time to walk an n-node binary search tree, since after the ini-

tial call, the procedure calls itself recursively exactly twice for each node in the

tree—once for its left child and once for its right child. The following theorem

gives a formal proof that it takes linear time to perform an inorder tree walk.

Theorem 12.1

If x is the root of an n-node subtree, then the call INORDER-TREE-WALK.x/

takes ‚.n/ time.

Proof Let T .n/ denote the time taken by INORDER-TREE-WALK when it is

called on the root of an n-node subtree. Since INORDER-TREE-WALK visits all n

nodes of the subtree, we have T .n/ D .n/. It remains to show that T .n/ D O.n/.

Since INORDER-TREE-WALK takes a small, constant amount of time on an

empty subtree (for the test x ¤ NIL), we have T .0/ D c for some constant c > 0.

For n > 0, suppose that INORDER-TREE-WALK is called on a node x whose

left subtree has k nodes and whose right subtree has n k 1 nodes. The time to

perform INORDER-TREE-WALK.x/ is bounded by T .n/ T .k/CT .nk1/Cd

for some constant d > 0 that reﬂects an upper bound on the time to execute the

body of INORDER-TREE-WALK.x/, exclusive of the time spent in recursive calls.

We use the substitution method to show that T .n/ D O.n/ by proving that

T .n/ .c C d/n C c. For n D 0, we have .c C d/ 0C c D c D T .0/. For n > 0,

we have

T .n/ T .k/ C T .n k 1/ C d

D ..c C d/k C c/ C ..c C d/.n k 1/ C c/ C d

D .c C d/n C c .c C d/ C c C d

D .c C d/n C c ;

which completes the proof.

12.2 Querying a binary search tree 289

Exercises

12.1-1

For the set of f1; 4; 5; 10; 16; 17; 21g of keys, draw binary search trees of heights 2,

3, 4, 5, and 6.

12.1-2

What is the difference between the binary-search-tree property and the min-heap

property (see page 153)? Can the min-heap property be used to print out the keys

of an n-node tree in sorted order in O.n/ time? Show how, or explain why not.

12.1-3

Give a nonrecursive algorithm that performs an inorder tree walk. (Hint: An easy

solution uses a stack as an auxiliary data structure. A more complicated, but ele-

gant, solution uses no stack but assumes that we can test two pointers for equality.)

12.1-4

Give recursive algorithms that perform preorder and postorder tree walks in ‚.n/

time on a tree of n nodes.

12.1-5

Argue that since sorting n elements takes .n lg n/ time in the worst case in

the comparison model, any comparison-based algorithm for constructing a binary

search tree from an arbitrary list of n elements takes .n lg n/ time in the worst

case.

12.2 Querying a binary search tree

We often need to search for a key stored in a binary search tree. Besides the

SEARCH operation, binary search trees can support such queries as MINIMUM,

MAXIMUM, SUCCESSOR, and PREDECESSOR. In this section, we shall examine

these operations and show how to support each one in time O.h/ on any binary

search tree of height h.

Searching

We use the following procedure to search for a node with a given key in a binary

search tree. Given a pointer to the root of the tree and a key k, TREE-SEARCH

returns a pointer to a node with key k if one exists; otherwise, it returns NIL.

290 Chapter 12 Binary Search Trees

2 4

3

13

7

6

17 20

18

15

9

Figure 12.2 Queries on a binary search tree. To search for the key 13 in the tree, we follow the path

15 ! 6 ! 7 ! 13 from the root. The minimum key in the tree is 2, which is found by following

left pointers from the root. The maximum key 20 is found by following right pointers from the root.

The successor of the node with key 15 is the node with key 17, since it is the minimum key in the

right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest

ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

TREE-SEARCH.x; k/

1 if x == NIL or k == x:key

2 return x

3 if k < x:key

4 return TREE-SEARCH.x:left; k/

5 else return TREE-SEARCH.x:right; k/

The procedure begins its search at the root and traces a simple path downward in

the tree, as shown in Figure 12.2. For each node x it encounters, it compares the

key k with x:key. If the two keys are equal, the search terminates. If k is smaller

than x:key, the search continues in the left subtree of x, since the binary-search-

tree property implies that k could not be stored in the right subtree. Symmetrically,

if k is larger than x:key, the search continues in the right subtree. The nodes

encountered during the recursion form a simple path downward from the root of

the tree, and thus the running time of TREE-SEARCH is O.h/, where h is the height

of the tree.

We can rewrite this procedure in an iterative fashion by “unrolling” the recursion

into a while loop. On most computers, the iterative version is more efﬁcient.

12.2 Querying a binary search tree 291

ITERATIVE-TREE-SEARCH.x; k/

1 while x ¤ NIL and k ¤ x:key

2 if k < x:key

3 x D x:left

4 else x D x:right

5 return x

Minimum and maximum

We can always ﬁnd an element in a binary search tree whose key is a minimum by

following left child pointers from the root until we encounter a NIL, as shown in

Figure 12.2. The following procedure returns a pointer to the minimum element in

the subtree rooted at a given node x, which we assume to be non-NIL:

TREE-MINIMUM.x/

1 while x:left ¤ NIL

2 x D x:left

3 return x

The binary-search-tree property guarantees that TREE-MINIMUM is correct. If a

node x has no left subtree, then since every key in the right subtree of x is at least as

large as x:key, the minimum key in the subtree rooted at x is x:key. If node x has

a left subtree, then since no key in the right subtree is smaller than x:key and every

key in the left subtree is not larger than x:key, the minimum key in the subtree

rooted at x resides in the subtree rooted at x:left.

The pseudocode for TREE-MAXIMUM is symmetric:

TREE-MAXIMUM.x/

1 while x:right ¤ NIL

2 x D x:right

3 return x

Both of these procedures run in O.h/ time on a tree of height h since, as in TREE-

SEARCH, the sequence of nodes encountered forms a simple path downward from

the root.

Successor and predecessor

Given a node in a binary search tree, sometimes we need to ﬁnd its successor in

the sorted order determined by an inorder tree walk. If all keys are distinct, the

292 Chapter 12 Binary Search Trees

successor of a node x is the node with the smallest key greater than x:key. The

structure of a binary search tree allows us to determine the successor of a node

without ever comparing keys. The following procedure returns the successor of a

node x in a binary search tree if it exists, and NIL if x has the largest key in the

tree:

TREE-SUCCESSOR.x/

1 if x:right ¤ NIL

2 return TREE-MINIMUM.x:right/

3 y D x:p

4 while y ¤ NIL and x == y:right

5 x D y

6 y D y:p

7 return y

We break the code for TREE-SUCCESSOR into two cases. If the right subtree

of node x is nonempty, then the successor of x is just the leftmost node in x’s

right subtree, which we ﬁnd in line 2 by calling TREE-MINIMUM.x:right/. For

example, the successor of the node with key 15 in Figure 12.2 is the node with

key 17.

On the other hand, as Exercise 12.2-6 asks you to show, if the right subtree of

node x is empty and x has a successor y, then y is the lowest ancestor of x whose

left child is also an ancestor of x. In Figure 12.2, the successor of the node with

key 13 is the node with key 15. To ﬁnd y, we simply go up the tree from x until we

encounter a node that is the left child of its parent; lines 3–7 of TREE-SUCCESSOR

handle this case.

The running time of TREE-SUCCESSOR on a tree of height h is O.h/, since we

either follow a simple path up the tree or follow a simple path down the tree. The

procedure TREE-PREDECESSOR, which is symmetric to TREE-SUCCESSOR, also

runs in time O.h/.

Even if keys are not distinct, we deﬁne the successor and predecessor of any

node x as the node returned by calls made to TREE-SUCCESSOR.x/ and TREE-

PREDECESSOR.x/, respectively.

In summary, we have proved the following theorem.

Theorem 12.2

We can implement the dynamic-set operations SEARCH, MINIMUM, MAXIMUM,

SUCCESSOR, and PREDECESSOR so that each one runs in O.h/ time on a binary

search tree of height h.

12.2 Querying a binary search tree 293

Exercises

12.2-1

Suppose that we have numbers between 1 and 1000 in a binary search tree, and we

want to search for the number 363. Which of the following sequences could not be

the sequence of nodes examined?

a. 2, 252, 401, 398, 330, 344, 397, 363.

b. 924, 220, 911, 244, 898, 258, 362, 363.

c. 925, 202, 911, 240, 912, 245, 363.

d. 2, 399, 387, 219, 266, 382, 381, 278, 363.

e. 935, 278, 347, 621, 299, 392, 358, 363.

12.2-2

Write recursive versions of TREE-MINIMUM and TREE-MAXIMUM.

12.2-3

Write the TREE-PREDECESSOR procedure.

12.2-4

Professor Bunyan thinks he has discovered a remarkable property of binary search

trees. Suppose that the search for key k in a binary search tree ends up in a leaf.

Consider three sets: A, the keys to the left of the search path; B, the keys on the

search path; and C , the keys to the right of the search path. Professor Bunyan

claims that any three keys a 2 A, b 2 B, and c 2 C must satisfy a b c. Give

a smallest possible counterexample to the professor’s claim.

12.2-5

Show that if a node in a binary search tree has two children, then its successor has

no left child and its predecessor has no right child.

12.2-6

Consider a binary search tree T whose keys are distinct. Show that if the right

subtree of a node x in T is empty and x has a successor y, then y is the lowest

ancestor of x whose left child is also an ancestor of x. (Recall that every node is

its own ancestor.)

12.2-7

An alternative method of performing an inorder tree walk of an n-node binary

search tree ﬁnds the minimum element in the tree by calling TREE-MINIMUM and

then making n 1 calls to TREE-SUCCESSOR. Prove that this algorithm runs

in ‚.n/ time.

294 Chapter 12 Binary Search Trees

12.2-8

Prove that no matter what node we start at in a height-h binary search tree, k

successive calls to TREE-SUCCESSOR take O.k C h/ time.

12.2-9

Let T be a binary search tree whose keys are distinct, let x be a leaf node, and let y

be its parent. Show that y:key is either the smallest key in T larger than x:key or

the largest key in T smaller than x:key.

12.3 Insertion and deletion

The operations of insertion and deletion cause the dynamic set represented by a

binary search tree to change. The data structure must be modiﬁed to reﬂect this

change, but in such a way that the binary-search-tree property continues to hold.

As we shall see, modifying the tree to insert a new element is relatively straight-

forward, but handling deletion is somewhat more intricate.

Insertion

To insert a new value into a binary search tree T , we use the procedure TREE-

INSERT. The procedure takes a node ´ for which ´:key D , ´:left D NIL,

and ´:right D NIL. It modiﬁes T and some of the attributes of ´ in such a way that

it inserts ´ into an appropriate position in the tree.

TREE-INSERT.T; ´/

1 y D NIL

2 x D T:root

3 while x ¤ NIL

4 y D x

5 if ´:key < x:key

6 x D x:left

7 else x D x:right

8 ´:p D y

9 if y == NIL

10 T:root D ´ // tree T was empty

11 elseif ´:key < y:key

12 y:left D ´

13 else y:right D ´

12.3 Insertion and deletion 295

2 9

5

13 17

15 19

18

12

Figure 12.3 Inserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate

the simple path from the root down to the position where the item is inserted. The dashed line

indicates the link in the tree that is added to insert the item.

Figure 12.3 shows how TREE-INSERT works. Just like the procedures TREE-

SEARCH and ITERATIVE-TREE-SEARCH, TREE-INSERT begins at the root of the

tree and the pointer x traces a simple path downward looking for a NIL to replace

with the input item ´. The procedure maintains the trailing pointer y as the parent

of x. After initialization, the while loop in lines 3–7 causes these two pointers

to move down the tree, going left or right depending on the comparison of ´:key

with x:key, until x becomes NIL. This NIL occupies the position where we wish to

place the input item ´. We need the trailing pointer y, because by the time we ﬁnd

the NIL where ´ belongs, the search has proceeded one step beyond the node that

needs to be changed. Lines 8–13 set the pointers that cause ´ to be inserted.

Like the other primitive operations on search trees, the procedure TREE-INSERT

runs in O.h/ time on a tree of height h.

Deletion

The overall strategy for deleting a node ´ from a binary search tree T has three

basic cases but, as we shall see, one of the cases is a bit tricky.

If ´ has no children, then we simply remove it by modifying its parent to re-

place ´ with NIL as its child.

If ´ has just one child, then we elevate that child to take ´’s position in the tree

by modifying ´’s parent to replace ´ by ´’s child.

If ´ has two children, then we ﬁnd ´’s successor y—which must be in ´’s right

subtree—and have y take ´’s position in the tree. The rest of ´’s original right

subtree becomes y’s new right subtree, and ´’s left subtree becomes y’s new

left subtree. This case is the tricky one because, as we shall see, it matters

whether y is ´’s right child.

296 Chapter 12 Binary Search Trees

The procedure for deleting a given node ´ from a binary search tree T takes as

arguments pointers to T and ´. It organizes its cases a bit differently from the three

cases outlined previously by considering the four cases shown in Figure 12.4.

If ´ has no left child (part (a) of the ﬁgure), then we replace ´ by its right child,

which may or may not be NIL. When ´’s right child is NIL, this case deals with

the situation in which ´ has no children. When ´’s right child is non-NIL, this

case handles the situation in which ´ has just one child, which is its right child.

If ´ has just one child, which is its left child (part (b) of the ﬁgure), then we

replace ´ by its left child.

Otherwise, ´ has both a left and a right child. We ﬁnd ´’s successor y, which

lies in ´’s right subtree and has no left child (see Exercise 12.2-5). We want to

splice y out of its current location and have it replace ´ in the tree.

If y is ´’s right child (part (c)), then we replace ´ by y, leaving y’s right

child alone.

Otherwise, y lies within ´’s right subtree but is not ´’s right child (part (d)).

In this case, we ﬁrst replace y by its own right child, and then we replace ´

by y.

In order to move subtrees around within the binary search tree, we deﬁne a

subroutine TRANSPLANT, which replaces one subtree as a child of its parent with

another subtree. When TRANSPLANT replaces the subtree rooted at node u with

the subtree rooted at node , node u’s parent becomes node ’s parent, and u’s

parent ends up having as its appropriate child.

TRANSPLANT.T; u; /

1 if u:p == NIL

2 T:root D

3 elseif u == u:p:left

4 u:p:left D

5 else u:p:right D

6 if ¤ NIL

7 :p D u:p

Lines 1–2 handle the case in which u is the root of T . Otherwise, u is either a left

child or a right child of its parent. Lines 3–4 take care of updating u:p:left if u

is a left child, and line 5 updates u:p:right if u is a right child. We allow to be

NIL, and lines 6–7 update :p if is non-NIL. Note that TRANSPLANT does not

attempt to update :left and :right; doing so, or not doing so, is the responsibility

of TRANSPLANT’s caller.

12.3 Insertion and deletion 297

q q

z (a) r

q q

z

l

(b)

q

z

l

(c)

q

y

l y

q

z

l

(d)

r

q

z

l r

y

q

l r

y

r

l

x

x

x y

x

x

NIL

NIL

NIL

NIL

NIL

Figure 12.4 Deleting a node ´ from a binary search tree. Node ´ may be the root, a left child of

node q, or a right child of q. (a) Node ´ has no left child. We replace ´ by its right child r, which

may or may not be NIL. (b) Node ´ has a left child l but no right child. We replace ´ by l. (c) Node ´

has two children; its left child is node l, its right child is its successor y, and y’s right child is node x.

We replace ´ by y, updating y’s left child to become l, but leaving x as y’s right child. (d) Node ´

has two children (left child l and right child r), and its successor y ¤ r lies within the subtree rooted

at r. We replace y by its own right child x, and we set y to be r’s parent. Then, we set y to be q’s

child and the parent of l.

298 Chapter 12 Binary Search Trees

With the TRANSPLANT procedure in hand, here is the procedure that deletes

node ´ from binary search tree T :

TREE-DELETE.T; ´/

1 if ´:left == NIL

2 TRANSPLANT.T; ´; ´:right/

3 elseif ´:right == NIL

4 TRANSPLANT.T; ´; ´:left/

5 else y D TREE-MINIMUM.´:right/

6 if y:p ¤ ´

7 TRANSPLANT.T; y; y:right/

8 y:right D ´:right

9 y:right:p D y

10 TRANSPLANT.T; ´; y/

11 y:left D ´:left

12 y:left:p D y

The TREE-DELETE procedure executes the four cases as follows. Lines 1–2

handle the case in which node ´ has no left child, and lines 3–4 handle the case in

which ´ has a left child but no right child. Lines 5–12 deal with the remaining two

cases, in which ´ has two children. Line 5 ﬁnds node y, which is the successor

of ´. Because ´ has a nonempty right subtree, its successor must be the node in

that subtree with the smallest key; hence the call to TREE-MINIMUM.´:right/. As

we noted before, y has no left child. We want to splice y out of its current location,

and it should replace ´ in the tree. If y is ´’s right child, then lines 10–12 replace ´

as a child of its parent by y and replace y’s left child by ´’s left child. If y is

not ´’s left child, lines 7–9 replace y as a child of its parent by y’s right child and

turn ´’s right child into y’s right child, and then lines 10–12 replace ´ as a child of

its parent by y and replace y’s left child by ´’s left child.

Each line of TREE-DELETE, including the calls to TRANSPLANT, takes constant

time, except for the call to TREE-MINIMUM in line 5. Thus, TREE-DELETE runs

in O.h/ time on a tree of height h.

In summary, we have proved the following theorem.

Theorem 12.3

We can implement the dynamic-set operations INSERT and DELETE so that each

one runs in O.h/ time on a binary search tree of height h.

12.4 Randomly built binary search trees 299

Exercises

12.3-1

Give a recursive version of the TREE-INSERT procedure.

12.3-2

Suppose that we construct a binary search tree by repeatedly inserting distinct val-

ues into the tree. Argue that the number of nodes examined in searching for a

value in the tree is one plus the number of nodes examined when the value was

ﬁrst inserted into the tree.

12.3-3

We can sort a given set of n numbers by ﬁrst building a binary search tree contain-

ing these numbers (using TREE-INSERT repeatedly to insert the numbers one by

one) and then printing the numbers by an inorder tree walk. What are the worst-

case and best-case running times for this sorting algorithm?

12.3-4

Is the operation of deletion “commutative” in the sense that deleting x and then y

from a binary search tree leaves the same tree as deleting y and then x? Argue why

it is or give a counterexample.

12.3-5

Suppose that instead of each node x keeping the attribute x:p, pointing to x’s

parent, it keeps x:succ, pointing to x’s successor. Give pseudocode for SEARCH,

INSERT, and DELETE on a binary search tree T using this representation. These

procedures should operate in time O.h/, where h is the height of the tree T . (Hint:

You may wish to implement a subroutine that returns the parent of a node.)

12.3-6

When node ´ in TREE-DELETE has two children, we could choose node y as

its predecessor rather than its successor. What other changes to TREE-DELETE

would be necessary if we did so? Some have argued that a fair strategy, giving

equal priority to predecessor and successor, yields better empirical performance.

How might TREE-DELETE be changed to implement such a fair strategy?

? 12.4 Randomly built binary search trees

We have shown that each of the basic operations on a binary search tree runs

in O.h/ time, where h is the height of the tree. The height of a binary search

300 Chapter 12 Binary Search Trees

tree varies, however, as items are inserted and deleted. If, for example, the n items

are inserted in strictly increasing order, the tree will be a chain with height n 1.

On the other hand, Exercise B.5-4 shows that h blg nc. As with quicksort, we

can show that the behavior of the average case is much closer to the best case than

to the worst case.

Unfortunately, little is known about the average height of a binary search tree

when both insertion and deletion are used to create it. When the tree is created

by insertion alone, the analysis becomes more tractable. Let us therefore deﬁne a

randomly built binary search tree on n keys as one that arises from inserting the

keys in random order into an initially empty tree, where each of the nŠ permutations

of the input keys is equally likely. (Exercise 12.4-3 asks you to show that this notion

is different from assuming that every binary search tree on n keys is equally likely.)

In this section, we shall prove the following theorem.

Theorem 12.4

The expected height of a randomly built binary search tree on n distinct keys is

O.lg n/.

Proof We start by deﬁning three random variables that help measure the height

of a randomly built binary search tree. We denote the height of a randomly built

binary search on n keys by X n , and we deﬁne the exponential height Y n D 2 X n .

When we build a binary search tree on n keys, we choose one key as that of the

root, and we let R n denote the random variable that holds this key’s rank within

the set of n keys; that is, R n holds the position that this key would occupy if the

set of keys were sorted. The value of R n is equally likely to be any element of the

set f1; 2; : : : ; ng. If R n D i, then the left subtree of the root is a randomly built

binary search tree on i 1 keys, and the right subtree is a randomly built binary

search tree on n i keys. Because the height of a binary tree is 1 more than the

larger of the heights of the two subtrees of the root, the exponential height of a

binary tree is twice the larger of the exponential heights of the two subtrees of the

root. If we know that R n D i, it follows that

Y n D 2 max.Y i1 ; Y ni / :

As base cases, we have that Y 1 D 1, because the exponential height of a tree with 1

node is 2 0 D 1 and, for convenience, we deﬁne Y 0 D 0.

Next, deﬁne indicator random variables Z n;1 ; Z n;2 ; : : : ; Z n;n , where

Z n;i D I fR n D ig :

Because R n is equally likely to be any element of f1; 2; : : : ; ng, it follows that

Pr fR n D ig D 1=n for i D 1; 2; : : : ; n, and hence, by Lemma 5.1, we have

E ŒZ n;i D 1=n ; (12.1)

12.4 Randomly built binary search trees 301

for i D 1; 2; : : : ; n. Because exactly one value of Z n;i is 1 and all others are 0, we

also have

Y n D

n X

iD1

Z n;i .2 max.Y i1 ; Y ni // :

We shall show that E ŒY n is polynomial in n, which will ultimately imply that

E ŒX n D O.lg n/.

We claim that the indicator random variable Z n;i D I fR n D ig is independent

of the values of Y i1 and Y ni . Having chosen R n D i, the left subtree (whose

exponential height is Y i1 ) is randomly built on the i 1 keys whose ranks are

less than i. This subtree is just like any other randomly built binary search tree

on i 1 keys. Other than the number of keys it contains, this subtree’s structure

is not affected at all by the choice of R n D i, and hence the random variables

Y i1 and Z n;i are independent. Likewise, the right subtree, whose exponential

height is Y ni , is randomly built on the n i keys whose ranks are greater than i.

Its structure is independent of the value of R n , and so the random variables Y ni

and Z n;i are independent. Hence, we have

E ŒY n D E

"

n X

iD1

Z n;i .2 max.Y i1 ; Y ni //

#

D

n X

iD1

E ŒZ n;i .2 max.Y i1 ; Y ni // (by linearity of expectation)

D

n X

iD1

E ŒZ n;i E Œ2 max.Y i1 ; Y ni / (by independence)

D

n X

iD1

1

n

E Œ2 max.Y i1 ; Y ni / (by equation (12.1))

D

2

n

n X

iD1

E Œmax.Y i1 ; Y ni / (by equation (C.22))

2

n

n X

iD1

.E ŒY i1 C E ŒY ni / (by Exercise C.3-4) .

Since each term E ŒY 0 ; E ŒY 1 ; : : : ; E ŒY n1 appears twice in the last summation,

once as E ŒY i1 and once as E ŒY ni , we have the recurrence

E ŒY n

4

n

n1 X

iD0

E ŒY i : (12.2)

302 Chapter 12 Binary Search Trees

Using the substitution method, we shall show that for all positive integers n, the

recurrence (12.2) has the solution

E ŒY n

1

4

n C 3

3

!

:

In doing so, we shall use the identity

n1 X

iD0

i C 3

3

!

D

n C 3

4

!

: (12.3)

(Exercise 12.4-1 asks you to prove this identity.)

For the base cases, we note that the bounds 0 D Y 0 D E ŒY 0 .1=4/

3

3

D 1=4

and 1 D Y 1 D E ŒY 1 .1=4/

1C3

3

D 1 hold. For the inductive case, we have that

E ŒY n

4

n

n1 X

iD0

E ŒY i

4

n

n1 X

iD0

1

4

i C 3

3

!

(by the inductive hypothesis)

D

1

n

n1 X

iD0

i C 3

3

!

D

1

n

n C 3

4

!

(by equation (12.3))

D

1

n

.n C 3/Š

4Š .n 1/Š

D

1

4

.n C 3/Š

3Š nŠ

D

1

4

n C 3

3

!

:

We have bounded E ŒY n , but our ultimate goal is to bound E ŒX n . As Exer-

cise 12.4-4 asks you to show, the function f .x/ D 2 x

is convex (see page 1199).

Therefore, we can employ Jensen’s inequality (C.26), which says that

2 EŒX n

E

2

X n

D E ŒY n ;

as follows:

2 EŒX n

1

4

n C 3

3

!

Problems for Chapter 12 303

D

1

4

.n C 3/.n C 2/.n C 1/

6

D

n 3 C 6n 2 C 11n C 6

24

:

Taking logarithms of both sides gives E ŒX n D O.lg n/.

Exercises

12.4-1

Prove equation (12.3).

12.4-2

Describe a binary search tree on n nodes such that the average depth of a node in

the tree is ‚.lg n/ but the height of the tree is !.lg n/. Give an asymptotic upper

bound on the height of an n-node binary search tree in which the average depth of

a node is ‚.lg n/.

12.4-3

Show that the notion of a randomly chosen binary search tree on n keys, where

each binary search tree of n keys is equally likely to be chosen, is different from

the notion of a randomly built binary search tree given in this section. (Hint: List

the possibilities when n D 3.)

12.4-4

Show that the function f .x/ D 2 x

is convex.

12.4-5 ?

Consider RANDOMIZED-QUICKSORT operating on a sequence of n distinct input

numbers. Prove that for any constant k > 0, all but O.1=n k / of the nŠ input

permutations yield an O.n lg n/ running time.

Problems

12-1 Binary search trees with equal keys

Equal keys pose a problem for the implementation of binary search trees.

a. What is the asymptotic performance of TREE-INSERT when used to insert n

items with identical keys into an initially empty binary search tree?

We propose to improve TREE-INSERT by testing before line 5 to determine whether

´:key D x:key and by testing before line 11 to determine whether ´:key D y:key.

304 Chapter 12 Binary Search Trees

If equality holds, we implement one of the following strategies. For each strategy,

ﬁnd the asymptotic performance of inserting n items with identical keys into an

initially empty binary search tree. (The strategies are described for line 5, in which

we compare the keys of ´ and x. Substitute y for x to arrive at the strategies for

line 11.)

b. Keep a boolean ﬂag x:b at node x, and set x to either x:left or x:right based

on the value of x:b, which alternates between FALSE and TRUE each time we

visit x while inserting a node with the same key as x.

c. Keep a list of nodes with equal keys at x, and insert ´ into the list.

d. Randomly set x to either x:left or x:right. (Give the worst-case performance

and informally derive the expected running time.)

12-2 Radix trees

Given two strings a D a 0 a 1 : : : a p and b D b 0 b 1 : : : b q , where each a i and each b j

is in some ordered set of characters, we say that string a is lexicographically less

than string b if either

1. there exists an integer j , where 0 j min.p; q/, such that a i D b i for all

i D 0; 1; : : : ; j 1 and a j < b j , or

2. p < q and a i D b i for all i D 0; 1; : : : ; p.

For example, if a and b are bit strings, then 10100 < 10110 by rule 1 (letting

j D 3) and 10100 < 101000 by rule 2. This ordering is similar to that used in

English-language dictionaries.

The radix tree data structure shown in Figure 12.5 stores the bit strings 1011,

10, 011, 100, and 0. When searching for a key a D a 0 a 1 : : : a p , we go left at a

node of depth i if a i D 0 and right if a i D 1. Let S be a set of distinct bit strings

whose lengths sum to n. Show how to use a radix tree to sort S lexicographically

in ‚.n/ time. For the example in Figure 12.5, the output of the sort should be the

sequence 0, 011, 10, 100, 1011.

12-3 Average node depth in a randomly built binary search tree

In this problem, we prove that the average depth of a node in a randomly built

binary search tree with n nodes is O.lg n/. Although this result is weaker than

that of Theorem 12.4, the technique we shall use reveals a surprising similarity

between the building of a binary search tree and the execution of RANDOMIZED-

QUICKSORT from Section 7.3.

We deﬁne the total path length P.T / of a binary tree T as the sum, over all

nodes x in T , of the depth of node x, which we denote by d.x; T /.

Problems for Chapter 12 305

011

0

100

10

1011

0 1

1 0

1 0 1

1

Figure 12.5 A radix tree storing the bit strings 1011, 10, 011, 100, and 0. We can determine each

node’s key by traversing the simple path from the root to that node. There is no need, therefore, to

store the keys in the nodes; the keys appear here for illustrative purposes only. Nodes are heavily

shaded if the keys corresponding to them are not in the tree; such nodes are present only to establish

a path to other nodes.

a. Argue that the average depth of a node in T is

1

n

X

x2T

d.x; T / D

1

n

P.T / :

Thus, we wish to show that the expected value of P.T / is O.n lg n/.

b. Let T L and T R denote the left and right subtrees of tree T , respectively. Argue

that if T has n nodes, then

P.T / D P.T L / C P.T R / C n 1 :

c. Let P.n/ denote the average total path length of a randomly built binary search

tree with n nodes. Show that

P.n/ D

1

n

n1 X

iD0

.P.i/ C P.n i 1/ C n 1/ :

d. Show how to rewrite P.n/ as

P.n/ D

2

n

n1 X

kD1

P.k/ C ‚.n/ :

e. Recalling the alternative analysis of the randomized version of quicksort given

in Problem 7-3, conclude that P.n/ D O.n lg n/.

306 Chapter 12 Binary Search Trees

At each recursive invocation of quicksort, we choose a random pivot element to

partition the set of elements being sorted. Each node of a binary search tree parti-

tions the set of elements that fall into the subtree rooted at that node.

f. Describe an implementation of quicksort in which the comparisons to sort a set

of elements are exactly the same as the comparisons to insert the elements into

a binary search tree. (The order in which comparisons are made may differ, but

the same comparisons must occur.)

12-4 Number of different binary trees

Let b n denote the number of different binary trees with n nodes. In this problem,

you will ﬁnd a formula for b n , as well as an asymptotic estimate.

a. Show that b 0 D 1 and that, for n 1,

b n D

n1 X

kD0

b k b n1k :

b. Referring to Problem 4-4 for the deﬁnition of a generating function, let B.x/

be the generating function

B.x/ D

1 X

nD0

b n x

n

:

Show that B.x/ D xB.x/ 2 C 1, and hence one way to express B.x/ in closed

form is

B.x/ D

1

2x

1

p

1 4x

:

The Taylor expansion of f .x/ around the point x D a is given by

f .x/ D

1 X

kD0

f .k/ .a/

kŠ

.x a/

k

;

where f .k/ .x/ is the kth derivative of f evaluated at x.

c. Show that

b n D

1

n C 1

2n

n

!

Notes for Chapter 12 307

(the nth Catalan number) by using the Taylor expansion of

p

1 4x around

x D 0. (If you wish, instead of using the Taylor expansion, you may use

the generalization of the binomial expansion (C.4) to nonintegral exponents n,

where for any real number n and for any integer k, we interpret

n

k

to be

n.n 1/ .n k C 1/=kŠ if k 0, and 0 otherwise.)

d. Show that

b n D

4 n

p

n 3=2

.1 C O.1=n// :

Chapter notes

Knuth [211] contains a good discussion of simple binary search trees as well as

many variations. Binary search trees seem to have been independently discovered

by a number of people in the late 1950s. Radix trees are often called “tries,” which

comes from the middle letters in the word retrieval. Knuth [211] also discusses

them.

Many texts, including the ﬁrst two editions of this book, have a somewhat sim-

pler method of deleting a node from a binary search tree when both of its children

are present. Instead of replacing node ´ by its successor y, we delete node y but

copy its key and satellite data into node ´. The downside of this approach is that

the node actually deleted might not be the node passed to the delete procedure. If

other components of a program maintain pointers to nodes in the tree, they could

mistakenly end up with “stale” pointers to nodes that have been deleted. Although

the deletion method presented in this edition of this book is a bit more complicated,

it guarantees that a call to delete node ´ deletes node ´ and only node ´.

Section 15.5 will show how to construct an optimal binary search tree when

we know the search frequencies before constructing the tree. That is, given the

frequencies of searching for each key and the frequencies of searching for values

that fall between keys in the tree, we construct a binary search tree for which a

set of searches that follows these frequencies examines the minimum number of

nodes.

The proof in Section 12.4 that bounds the expected height of a randomly built

binary search tree is due to Aslam [24]. Mart´ınez and Roura [243] give randomized

algorithms for insertion into and deletion from binary search trees in which the

result of either operation is a random binary search tree. Their deﬁnition of a

random binary search tree differs—only slightly—from that of a randomly built

binary search tree in this chapter, however.

13 Red-Black Trees

Chapter 12 showed that a binary search tree of height h can support any of the basic

dynamic-set operations—such as SEARCH, PREDECESSOR, SUCCESSOR, MINI-

MUM, MAXIMUM, INSERT, and DELETE—in O.h/ time. Thus, the set operations

are fast if the height of the search tree is small. If its height is large, however, the

set operations may run no faster than with a linked list. Red-black trees are one

of many search-tree schemes that are “balanced” in order to guarantee that basic

dynamic-set operations take O.lg n/ time in the worst case.

13.1 Properties of red-black trees

A red-black tree is a binary search tree with one extra bit of storage per node: its

color, which can be either RED or BLACK. By constraining the node colors on any

simple path from the root to a leaf, red-black trees ensure that no such path is more

than twice as long as any other, so that the tree is approximately balanced.

Each node of the tree now contains the attributes color, key, left, right, and p. If

a child or the parent of a node does not exist, the corresponding pointer attribute

of the node contains the value NIL. We shall regard these NILs as being pointers to

leaves (external nodes) of the binary search tree and the normal, key-bearing nodes

as being internal nodes of the tree.

A red-black tree is a binary tree that satisﬁes the following red-black properties:

1. Every node is either red or black.

2. The root is black.

3. Every leaf (NIL) is black.

4. If a node is red, then both its children are black.

5. For each node, all simple paths from the node to descendant leaves contain the

same number of black nodes.

13.1 Properties of red-black trees 309

Figure 13.1(a) shows an example of a red-black tree.

As a matter of convenience in dealing with boundary conditions in red-black

tree code, we use a single sentinel to represent NIL (see page 238). For a red-black

tree T , the sentinel T:nil is an object with the same attributes as an ordinary node

in the tree. Its color attribute is BLACK, and its other attributes—p, left, right,

and key—can take on arbitrary values. As Figure 13.1(b) shows, all pointers to NIL

are replaced by pointers to the sentinel T:nil.

We use the sentinel so that we can treat a NIL child of a node x as an ordinary

node whose parent is x. Although we instead could add a distinct sentinel node

for each NIL in the tree, so that the parent of each NIL is well deﬁned, that ap-

proach would waste space. Instead, we use the one sentinel T:nil to represent all

the NILs—all leaves and the root’s parent. The values of the attributes p, left, right,

and key of the sentinel are immaterial, although we may set them during the course

of a procedure for our convenience.

We generally conﬁne our interest to the internal nodes of a red-black tree, since

they hold the key values. In the remainder of this chapter, we omit the leaves when

we draw red-black trees, as shown in Figure 13.1(c).

We call the number of black nodes on any simple path from, but not including, a

node x down to a leaf the black-height of the node, denoted bh.x/. By property 5,

the notion of black-height is well deﬁned, since all descending simple paths from

the node have the same number of black nodes. We deﬁne the black-height of a

red-black tree to be the black-height of its root.

The following lemma shows why red-black trees make good search trees.

Lemma 13.1

A red-black tree with n internal nodes has height at most 2 lg.n C 1/.

Proof We start by showing that the subtree rooted at any node x contains at least

2 bh.x/ 1 internal nodes. We prove this claim by induction on the height of x. If

the height of x is 0, then x must be a leaf (T:nil), and the subtree rooted at x indeed

contains at least 2 bh.x/ 1 D 2 0 1 D 0 internal nodes. For the inductive step,

consider a node x that has positive height and is an internal node with two children.

Each child has a black-height of either bh.x/ or bh.x/ 1, depending on whether

its color is red or black, respectively. Since the height of a child of x is less than

the height of x itself, we can apply the inductive hypothesis to conclude that each

child has at least 2 bh.x/1 1 internal nodes. Thus, the subtree rooted at x contains

at least .2 bh.x/1 1/C.2 bh.x/1 1/C1 D 2 bh.x/ 1 internal nodes, which proves

the claim.

To complete the proof of the lemma, let h be the height of the tree. According

to property 4, at least half the nodes on any simple path from the root to a leaf, not

310 Chapter 13 Red-Black Trees

NIL NIL

NIL NIL NIL NIL NIL

NIL NIL

NIL NIL

NIL NIL NIL NIL

NIL NIL NIL NIL

NIL NIL

26

41

47 30

28 38

35 39

17

21

23 19

20

14

16

15

10

12 7

3 1

1 1

2

1

1

2

1

1 1

2

3

1 1

1 1

2 1

2

3

(a)

26

41

47 30

28 38

35 39

17

21

23 19

20

14

16

15

10

12 7

3

(b)

26

41

47 30

28 38

35 39

17

21

23 19

20

14

16

15

10

12 7

3

(c)

T:nil

Figure 13.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a

red-black tree is either red or black, the children of a red node are both black, and every simple path

from a node to a descendant leaf contains the same number of black nodes. (a) Every leaf, shown

as a NIL, is black. Each non-NIL node is marked with its black-height; NILs have black-height 0.

(b) The same red-black tree but with each NIL replaced by the single sentinel T:nil, which is always

black, and with black-heights omitted. The root’s parent is also the sentinel. (c) The same red-black

tree but with leaves and the root’s parent omitted entirely. We shall use this drawing style in the

remainder of this chapter.

13.1 Properties of red-black trees 311

including the root, must be black. Consequently, the black-height of the root must

be at least h=2; thus,

n 2

h=2

1 :

Moving the 1 to the left-hand side and taking logarithms on both sides yields

lg.n C 1/ h=2, or h 2 lg.n C 1/.

As an immediate consequence of this lemma, we can implement the dynamic-set

operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR

in O.lg n/ time on red-black trees, since each can run in O.h/ time on a binary

search tree of height h (as shown in Chapter 12) and any red-black tree on n nodes

is a binary search tree with height O.lg n/. (Of course, references to NIL in the

algorithms of Chapter 12 would have to be replaced by T:nil.) Although the al-

gorithms TREE-INSERT and TREE-DELETE from Chapter 12 run in O.lg n/ time

when given a red-black tree as input, they do not directly support the dynamic-set

operations INSERT and DELETE, since they do not guarantee that the modiﬁed bi-

nary search tree will be a red-black tree. We shall see in Sections 13.3 and 13.4,

however, how to support these two operations in O.lg n/ time.

Exercises

13.1-1

In the style of Figure 13.1(a), draw the complete binary search tree of height 3 on

the keys f1; 2; : : : ; 15g. Add the NIL leaves and color the nodes in three different

ways such that the black-heights of the resulting red-black trees are 2, 3, and 4.

13.1-2

Draw the red-black tree that results after TREE-INSERT is called on the tree in

Figure 13.1 with key 36. If the inserted node is colored red, is the resulting tree a

red-black tree? What if it is colored black?

13.1-3

Let us deﬁne a relaxed red-black tree as a binary search tree that satisﬁes red-

black properties 1, 3, 4, and 5. In other words, the root may be either red or black.

Consider a relaxed red-black tree T whose root is red. If we color the root of T

black but make no other changes to T , is the resulting tree a red-black tree?

13.1-4

Suppose that we “absorb” every red node in a red-black tree into its black parent,

so that the children of the red node become children of the black parent. (Ignore

what happens to the keys.) What are the possible degrees of a black node after all

312 Chapter 13 Red-Black Trees

its red children are absorbed? What can you say about the depths of the leaves of

the resulting tree?

13.1-5

Show that the longest simple path from a node x in a red-black tree to a descendant

leaf has length at most twice that of the shortest simple path from node x to a

descendant leaf.

13.1-6

What is the largest possible number of internal nodes in a red-black tree with black-

height k? What is the smallest possible number?

13.1-7

Describe a red-black tree on n keys that realizes the largest possible ratio of red in-

ternal nodes to black internal nodes. What is this ratio? What tree has the smallest

possible ratio, and what is the ratio?

13.2 Rotations

The search-tree operations TREE-INSERT and TREE-DELETE, when run on a red-

black tree with n keys, take O.lg n/ time. Because they modify the tree, the result

may violate the red-black properties enumerated in Section 13.1. To restore these

properties, we must change the colors of some of the nodes in the tree and also

change the pointer structure.

We change the pointer structure through rotation, which is a local operation in

a search tree that preserves the binary-search-tree property. Figure 13.2 shows the

two kinds of rotations: left rotations and right rotations. When we do a left rotation

on a node x, we assume that its right child y is not T:nil; x may be any node in

the tree whose right child is not T:nil. The left rotation “pivots” around the link

from x to y. It makes y the new root of the subtree, with x as y’s left child and y’s

left child as x’s right child.

The pseudocode for LEFT-ROTATE assumes that x:right ¤ T:nil and that the

root’s parent is T:nil.

13.2 Rotations 313

y

x

α β

γ

x

yα

β γ

LEFT-ROTATE(T, x)

RIGHT-ROTATE(T, y)

Figure 13.2 The rotation operations on a binary search tree. The operation LEFT-ROTATE.T; x/

transforms the conﬁguration of the two nodes on the right into the conﬁguration on the left by chang-

ing a constant number of pointers. The inverse operation RIGHT-ROTATE.T; y/ transforms the con-

ﬁguration on the left into the conﬁguration on the right. The letters ˛, ˇ, and represent arbitrary

subtrees. A rotation operation preserves the binary-search-tree property: the keys in ˛ precede x:key,

which precedes the keys in ˇ, which precede y:key, which precedes the keys in .

LEFT-ROTATE.T; x/

1 y D x:right // set y

2 x:right D y:left // turn y’s left subtree into x’s right subtree

3 if y:left ¤ T:nil

4 y:left:p D x

5 y:p D x:p // link x’s parent to y

6 if x:p == T:nil

7 T:root D y

8 elseif x == x:p:left

9 x:p:left D y

10 else x:p:right D y

11 y:left D x // put x on y’s left

12 x:p D y

Figure 13.3 shows an example of how LEFT-ROTATE modiﬁes a binary search

tree. The code for RIGHT-ROTATE is symmetric. Both LEFT-ROTATE and RIGHT-

ROTATE run in O.1/ time. Only pointers are changed by a rotation; all other

attributes in a node remain the same.

Exercises

13.2-1

Write pseudocode for RIGHT-ROTATE.

13.2-2

Argue that in every n-node binary search tree, there are exactly n 1 possible

rotations.

314 Chapter 13 Red-Black Trees

2

3

4

6

7

11

9 18

14

12 17

19

22

20

x

y

2

3

4

6

7

18

19

14

12 17

22

20

x

y

11

9

LEFT-ROTATE(T, x)

Figure 13.3 An example of how the procedure LEFT-ROTATE.T; x/ modiﬁes a binary search tree.

Inorder tree walks of the input tree and the modiﬁed tree produce the same listing of key values.

13.2-3

Let a, b, and c be arbitrary nodes in subtrees ˛, ˇ, and , respectively, in the left

tree of Figure 13.2. How do the depths of a, b, and c change when a left rotation

is performed on node x in the ﬁgure?

13.2-4

Show that any arbitrary n-node binary search tree can be transformed into any other

arbitrary n-node binary search tree using O.n/ rotations. (Hint: First show that at

most n 1 right rotations sufﬁce to transform the tree into a right-going chain.)

13.2-5 ?

We say that a binary search tree T 1 can be right-converted to binary search tree T 2

if it is possible to obtain T 2 from T 1 via a series of calls to RIGHT-ROTATE. Give

an example of two trees T 1 and T 2 such that T 1 cannot be right-converted to T 2 .

Then, show that if a tree T 1 can be right-converted to T 2 , it can be right-converted

using O.n 2 / calls to RIGHT-ROTATE.

13.3 Insertion 315

13.3 Insertion

We can insert a node into an n-node red-black tree in O.lg n/ time. To do so, we

use a slightly modiﬁed version of the TREE-INSERT procedure (Section 12.3) to

insert node ´ into the tree T as if it were an ordinary binary search tree, and then we

color ´ red. (Exercise 13.3-1 asks you to explain why we choose to make node ´

red rather than black.) To guarantee that the red-black properties are preserved, we

then call an auxiliary procedure RB-INSERT-FIXUP to recolor nodes and perform

rotations. The call RB-INSERT.T; ´/ inserts node ´, whose key is assumed to have

already been ﬁlled in, into the red-black tree T .

RB-INSERT.T; ´/

1 y D T:nil

2 x D T:root

3 while x ¤ T:nil

4 y D x

5 if ´:key < x:key

6 x D x:left

7 else x D x:right

8 ´:p D y

9 if y == T:nil

10 T:root D ´

11 elseif ´:key < y:key

12 y:left D ´

13 else y:right D ´

14 ´:left D T:nil

15 ´:right D T:nil

16 ´:color D RED

17 RB-INSERT-FIXUP.T; ´/

The procedures TREE-INSERT and RB-INSERT differ in four ways. First, all

instances of NIL in TREE-INSERT are replaced by T:nil. Second, we set ´:left

and ´:right to T:nil in lines 14–15 of RB-INSERT, in order to maintain the

proper tree structure. Third, we color ´ red in line 16. Fourth, because col-

oring ´ red may cause a violation of one of the red-black properties, we call

RB-INSERT-FIXUP.T; ´/ in line 17 of RB-INSERT to restore the red-black prop-

erties.

316 Chapter 13 Red-Black Trees

RB-INSERT-FIXUP.T; ´/

1 while ´:p:color == RED

2 if ´:p == ´:p:p:left

3 y D ´:p:p:right

4 if y:color == RED

5 ´:p:color D BLACK // case 1

6 y:color D BLACK // case 1

7 ´:p:p:color D RED // case 1

8 ´ D ´:p:p // case 1

9 else if ´ == ´:p:right

10 ´ D ´:p // case 2

11 LEFT-ROTATE.T; ´/ // case 2

12 ´:p:color D BLACK // case 3

13 ´:p:p:color D RED // case 3

14 RIGHT-ROTATE.T; ´:p:p/ // case 3

15 else (same as then clause

with “right” and “left” exchanged)

16 T:root:color D BLACK

To understand how RB-INSERT-FIXUP works, we shall break our examination

of the code into three major steps. First, we shall determine what violations of

the red-black properties are introduced in RB-INSERT when node ´ is inserted

and colored red. Second, we shall examine the overall goal of the while loop in

lines 1–15. Finally, we shall explore each of the three cases 1 within the while

loop’s body and see how they accomplish the goal. Figure 13.4 shows how RB-

INSERT-FIXUP operates on a sample red-black tree.

Which of the red-black properties might be violated upon the call to RB-

INSERT-FIXUP? Property 1 certainly continues to hold, as does property 3, since

both children of the newly inserted red node are the sentinel T:nil. Property 5,

which says that the number of black nodes is the same on every simple path from

a given node, is satisﬁed as well, because node ´ replaces the (black) sentinel, and

node ´ is red with sentinel children. Thus, the only properties that might be vi-

olated are property 2, which requires the root to be black, and property 4, which

says that a red node cannot have a red child. Both possible violations are due to ´

being colored red. Property 2 is violated if ´ is the root, and property 4 is violated

if ´’s parent is red. Figure 13.4(a) shows a violation of property 4 after the node ´

has been inserted.

1

Case 2 falls through into case 3, and so these two cases are not mutually exclusive.

13.3 Insertion 317

z

y

11

2

1 7

5

4

8

14

15

z

y

11

2

1 7

5

4

8

14

15

(a)

(b)

Case 1

z

y

11

7

2 8

4

14

15 (c)

Case 2

1 5

4

z

7

2

1 5

11

14 (d)

Case 3

4

8

15

Figure 13.4 The operation of RB-INSERT-FIXUP. (a) A node ´ after insertion. Because both ´

and its parent ´:p are red, a violation of property 4 occurs. Since ´’s uncle y is red, case 1 in the

code applies. We recolor nodes and move the pointer ´ up the tree, resulting in the tree shown in (b).

Once again, ´ and its parent are both red, but ´’s uncle y is black. Since ´ is the right child of ´:p,

case 2 applies. We perform a left rotation, and the tree that results is shown in (c). Now, ´ is the left

child of its parent, and case 3 applies. Recoloring and right rotation yield the tree in (d), which is a

legal red-black tree.

318 Chapter 13 Red-Black Trees

The while loop in lines 1–15 maintains the following three-part invariant at the

start of each iteration of the loop:

a. Node ´ is red.

b. If ´:p is the root, then ´:p is black.

c. If the tree violates any of the red-black properties, then it violates at most

one of them, and the violation is of either property 2 or property 4. If the

tree violates property 2, it is because ´ is the root and is red. If the tree

violates property 4, it is because both ´ and ´:p are red.

Part (c), which deals with violations of red-black properties, is more central to

showing that RB-INSERT-FIXUP restores the red-black properties than parts (a)

and (b), which we use along the way to understand situations in the code. Because

we’ll be focusing on node ´ and nodes near it in the tree, it helps to know from

part (a) that ´ is red. We shall use part (b) to show that the node ´:p:p exists when

we reference it in lines 2, 3, 7, 8, 13, and 14.

Recall that we need to show that a loop invariant is true prior to the ﬁrst itera-

tion of the loop, that each iteration maintains the loop invariant, and that the loop

invariant gives us a useful property at loop termination.

We start with the initialization and termination arguments. Then, as we exam-

ine how the body of the loop works in more detail, we shall argue that the loop

maintains the invariant upon each iteration. Along the way, we shall also demon-

strate that each iteration of the loop has two possible outcomes: either the pointer ´

moves up the tree, or we perform some rotations and then the loop terminates.

Initialization: Prior to the ﬁrst iteration of the loop, we started with a red-black

tree with no violations, and we added a red node ´. We show that each part of

the invariant holds at the time RB-INSERT-FIXUP is called:

a. When RB-INSERT-FIXUP is called, ´ is the red node that was added.

b. If ´:p is the root, then ´:p started out black and did not change prior to the

call of RB-INSERT-FIXUP.

c. We have already seen that properties 1, 3, and 5 hold when RB-INSERT-

FIXUP is called.

If the tree violates property 2, then the red root must be the newly added

node ´, which is the only internal node in the tree. Because the parent and

both children of ´ are the sentinel, which is black, the tree does not also

violate property 4. Thus, this violation of property 2 is the only violation of

red-black properties in the entire tree.

If the tree violates property 4, then, because the children of node ´ are black

sentinels and the tree had no other violations prior to ´ being added, the

13.3 Insertion 319

violation must be because both ´ and ´:p are red. Moreover, the tree violates

no other red-black properties.

Termination: When the loop terminates, it does so because ´:p is black. (If ´ is

the root, then ´:p is the sentinel T:nil, which is black.) Thus, the tree does not

violate property 4 at loop termination. By the loop invariant, the only property

that might fail to hold is property 2. Line 16 restores this property, too, so that

when RB-INSERT-FIXUP terminates, all the red-black properties hold.

Maintenance: We actually need to consider six cases in the while loop, but three

of them are symmetric to the other three, depending on whether line 2 deter-

mines ´’s parent ´:p to be a left child or a right child of ´’s grandparent ´:p:p.

We have given the code only for the situation in which ´:p is a left child. The

node ´:p:p exists, since by part (b) of the loop invariant, if ´:p is the root,

then ´:p is black. Since we enter a loop iteration only if ´:p is red, we know

that ´:p cannot be the root. Hence, ´:p:p exists.

We distinguish case 1 from cases 2 and 3 by the color of ´’s parent’s sibling,

or “uncle.” Line 3 makes y point to ´’s uncle ´:p:p:right, and line 4 tests y’s

color. If y is red, then we execute case 1. Otherwise, control passes to cases 2

and 3. In all three cases, ´’s grandparent ´:p:p is black, since its parent ´:p is

red, and property 4 is violated only between ´ and ´:p.

Case 1: ´’s uncle y is red

Figure 13.5 shows the situation for case 1 (lines 5–8), which occurs when

both ´:p and y are red. Because ´:p:p is black, we can color both ´:p and y

black, thereby ﬁxing the problem of ´ and ´:p both being red, and we can

color ´:p:p red, thereby maintaining property 5. We then repeat the while loop

with ´:p:p as the new node ´. The pointer ´ moves up two levels in the tree.

Now, we show that case 1 maintains the loop invariant at the start of the next

iteration. We use ´ to denote node ´ in the current iteration, and ´ 0 D ´:p:p

to denote the node that will be called node ´ at the test in line 1 upon the next

iteration.

a. Because this iteration colors ´:p:p red, node ´ 0

is red at the start of the next

iteration.

b. The node ´ 0 :p is ´:p:p:p in this iteration, and the color of this node does not

change. If this node is the root, it was black prior to this iteration, and it

remains black at the start of the next iteration.

c. We have already argued that case 1 maintains property 5, and it does not

introduce a violation of properties 1 or 3.

320 Chapter 13 Red-Black Trees

z

y

C

D A

Bα

β γ

δ ε

(a)

C

D A

Bα

β γ

δ ε

new z

y

C

D B

δ ε

C

D B

A

α β

γ δ ε

new z

(b)

A

α β

γ

z

Figure 13.5 Case 1 of the procedure RB-INSERT-FIXUP. Property 4 is violated, since ´ and its

parent ´:p are both red. We take the same action whether (a) ´ is a right child or (b) ´ is a left

child. Each of the subtrees ˛, ˇ, , ı, and " has a black root, and each has the same black-height.

The code for case 1 changes the colors of some nodes, preserving property 5: all downward simple

paths from a node to a leaf have the same number of blacks. The while loop continues with node ´’s

grandparent ´:p:p as the new ´. Any violation of property 4 can now occur only between the new ´,

which is red, and its parent, if it is red as well.

If node ´ 0

is the root at the start of the next iteration, then case 1 corrected

the lone violation of property 4 in this iteration. Since ´ 0

is red and it is the

root, property 2 becomes the only one that is violated, and this violation is

due to ´ 0

.

If node ´ 0

is not the root at the start of the next iteration, then case 1 has

not created a violation of property 2. Case 1 corrected the lone violation

of property 4 that existed at the start of this iteration. It then made ´ 0

red

and left ´ 0 :p alone. If ´ 0 :p was black, there is no violation of property 4.

If ´ 0 :p was red, coloring ´ 0

red created one violation of property 4 between ´ 0

and ´ 0 :p.

Case 2: ´’s uncle y is black and ´ is a right child

Case 3: ´’s uncle y is black and ´ is a left child

In cases 2 and 3, the color of ´’s uncle y is black. We distinguish the two cases

according to whether ´ is a right or left child of ´:p. Lines 10–11 constitute

case 2, which is shown in Figure 13.6 together with case 3. In case 2, node ´

is a right child of its parent. We immediately use a left rotation to transform

the situation into case 3 (lines 12–14), in which node ´ is a left child. Because

13.3 Insertion 321

C

A

Bα

β γ

δ

Case 2

z

y B

A

α β

γ

δ

Case 3

z

y

z A

B

C

α β γ δ

C

Figure 13.6 Cases 2 and 3 of the procedure RB-INSERT-FIXUP. As in case 1, property 4 is violated

in either case 2 or case 3 because ´ and its parent ´:p are both red. Each of the subtrees ˛, ˇ, , and ı

has a black root (˛, ˇ, and from property 4, and ı because otherwise we would be in case 1), and

each has the same black-height. We transform case 2 into case 3 by a left rotation, which preserves

property 5: all downward simple paths from a node to a leaf have the same number of blacks. Case 3

causes some color changes and a right rotation, which also preserve property 5. The while loop then

terminates, because property 4 is satisﬁed: there are no longer two red nodes in a row.

both ´ and ´:p are red, the rotation affects neither the black-height of nodes

nor property 5. Whether we enter case 3 directly or through case 2, ´’s uncle y

is black, since otherwise we would have executed case 1. Additionally, the

node ´:p:p exists, since we have argued that this node existed at the time that

lines 2 and 3 were executed, and after moving ´ up one level in line 10 and then

down one level in line 11, the identity of ´:p:p remains unchanged. In case 3,

we execute some color changes and a right rotation, which preserve property 5,

and then, since we no longer have two red nodes in a row, we are done. The

while loop does not iterate another time, since ´:p is now black.

We now show that cases 2 and 3 maintain the loop invariant. (As we have just

argued, ´:p will be black upon the next test in line 1, and the loop body will not

execute again.)

a. Case 2 makes ´ point to ´:p, which is red. No further change to ´ or its color

occurs in cases 2 and 3.

b. Case 3 makes ´:p black, so that if ´:p is the root at the start of the next

iteration, it is black.

c. As in case 1, properties 1, 3, and 5 are maintained in cases 2 and 3.

Since node ´ is not the root in cases 2 and 3, we know that there is no viola-

tion of property 2. Cases 2 and 3 do not introduce a violation of property 2,

since the only node that is made red becomes a child of a black node by the

rotation in case 3.

Cases 2 and 3 correct the lone violation of property 4, and they do not intro-

duce another violation.

322 Chapter 13 Red-Black Trees

Having shown that each iteration of the loop maintains the invariant, we have

shown that RB-INSERT-FIXUP correctly restores the red-black properties.

Analysis

What is the running time of RB-INSERT? Since the height of a red-black tree on n

nodes is O.lg n/, lines 1–16 of RB-INSERT take O.lg n/ time. In RB-INSERT-

FIXUP, the while loop repeats only if case 1 occurs, and then the pointer ´ moves

two levels up the tree. The total number of times the while loop can be executed

is therefore O.lg n/. Thus, RB-INSERT takes a total of O.lg n/ time. Moreover, it

never performs more than two rotations, since the while loop terminates if case 2

or case 3 is executed.

Exercises

13.3-1

In line 16 of RB-INSERT, we set the color of the newly inserted node ´ to red.

Observe that if we had chosen to set ´’s color to black, then property 4 of a red-

black tree would not be violated. Why didn’t we choose to set ´’s color to black?

13.3-2

Show the red-black trees that result after successively inserting the keys 41; 38; 31;

12; 19; 8 into an initially empty red-black tree.

13.3-3

Suppose that the black-height of each of the subtrees ˛; ˇ; ; ı; " in Figures 13.5

and 13.6 is k. Label each node in each ﬁgure with its black-height to verify that

the indicated transformation preserves property 5.

13.3-4

Professor Teach is concerned that RB-INSERT-FIXUP might set T:nil:color to

RED, in which case the test in line 1 would not cause the loop to terminate when ´

is the root. Show that the professor’s concern is unfounded by arguing that RB-

INSERT-FIXUP never sets T:nil:color to RED.

13.3-5

Consider a red-black tree formed by inserting n nodes with RB-INSERT. Argue

that if n > 1, the tree has at least one red node.

13.3-6

Suggest how to implement RB-INSERT efﬁciently if the representation for red-

black trees includes no storage for parent pointers.

13.4 Deletion 323

13.4 Deletion

Like the other basic operations on an n-node red-black tree, deletion of a node takes

time O.lg n/. Deleting a node from a red-black tree is a bit more complicated than

inserting a node.

The procedure for deleting a node from a red-black tree is based on the TREE-

DELETE procedure (Section 12.3). First, we need to customize the TRANSPLANT

subroutine that TREE-DELETE calls so that it applies to a red-black tree:

RB-TRANSPLANT.T; u; /

1 if u:p == T:nil

2 T:root D

3 elseif u == u:p:left

4 u:p:left D

5 else u:p:right D

6 :p D u:p

The procedure RB-TRANSPLANT differs from TRANSPLANT in two ways. First,

line 1 references the sentinel T:nil instead of NIL. Second, the assignment to :p in

line 6 occurs unconditionally: we can assign to :p even if points to the sentinel.

In fact, we shall exploit the ability to assign to :p when D T:nil.

The procedure RB-DELETE is like the TREE-DELETE procedure, but with ad-

ditional lines of pseudocode. Some of the additional lines keep track of a node y

that might cause violations of the red-black properties. When we want to delete

node ´ and ´ has fewer than two children, then ´ is removed from the tree, and we

want y to be ´. When ´ has two children, then y should be ´’s successor, and y

moves into ´’s position in the tree. We also remember y’s color before it is re-

moved from or moved within the tree, and we keep track of the node x that moves

into y’s original position in the tree, because node x might also cause violations

of the red-black properties. After deleting node ´, RB-DELETE calls an auxiliary

procedure RB-DELETE-FIXUP, which changes colors and performs rotations to

restore the red-black properties.

324 Chapter 13 Red-Black Trees

RB-DELETE.T; ´/

1 y D ´

2 y-original-color D y:color

3 if ´:left == T:nil

4 x D ´:right

5 RB-TRANSPLANT.T; ´; ´:right/

6 elseif ´:right == T:nil

7 x D ´:left

8 RB-TRANSPLANT.T; ´; ´:left/

9 else y D TREE-MINIMUM.´:right/

10 y-original-color D y:color

11 x D y:right

12 if y:p == ´

13 x:p D y

14 else RB-TRANSPLANT.T; y; y:right/

15 y:right D ´:right

16 y:right:p D y

17 RB-TRANSPLANT.T; ´; y/

18 y:left D ´:left

19 y:left:p D y

20 y:color D ´:color

21 if y-original-color == BLACK

22 RB-DELETE-FIXUP.T; x/

Although RB-DELETE contains almost twice as many lines of pseudocode as

TREE-DELETE, the two procedures have the same basic structure. You can ﬁnd

each line of TREE-DELETE within RB-DELETE (with the changes of replacing

NIL by T:nil and replacing calls to TRANSPLANT by calls to RB-TRANSPLANT),

executed under the same conditions.

Here are the other differences between the two procedures:

We maintain node y as the node either removed from the tree or moved within

the tree. Line 1 sets y to point to node ´ when ´ has fewer than two children

and is therefore removed. When ´ has two children, line 9 sets y to point to ´’s

successor, just as in TREE-DELETE, and y will move into ´’s position in the

tree.

Because node y’s color might change, the variable y-original-color stores y’s

color before any changes occur. Lines 2 and 10 set this variable immediately

after assignments to y. When ´ has two children, then y ¤ ´ and node y

moves into node ´’s original position in the red-black tree; line 20 gives y the

same color as ´. We need to save y’s original color in order to test it at the

13.4 Deletion 325

end of RB-DELETE; if it was black, then removing or moving y could cause

violations of the red-black properties.

As discussed, we keep track of the node x that moves into node y’s original

position. The assignments in lines 4, 7, and 11 set x to point to either y’s only

child or, if y has no children, the sentinel T:nil. (Recall from Section 12.3

that y has no left child.)

Since node x moves into node y’s original position, the attribute x:p is always

set to point to the original position in the tree of y’s parent, even if x is, in fact,

the sentinel T:nil. Unless ´ is y’s original parent (which occurs only when ´ has

two children and its successor y is ´’s right child), the assignment to x:p takes

place in line 6 of RB-TRANSPLANT. (Observe that when RB-TRANSPLANT

is called in lines 5, 8, or 14, the second parameter passed is the same as x.)

When y’s original parent is ´, however, we do not want x:p to point to y’s orig-

inal parent, since we are removing that node from the tree. Because node y will

move up to take ´’s position in the tree, setting x:p to y in line 13 causes x:p

to point to the original position of y’s parent, even if x D T:nil.

Finally, if node y was black, we might have introduced one or more violations

of the red-black properties, and so we call RB-DELETE-FIXUP in line 22 to

restore the red-black properties. If y was red, the red-black properties still hold

when y is removed or moved, for the following reasons:

1. No black-heights in the tree have changed.

2. No red nodes have been made adjacent. Because y takes ´’s place in the

tree, along with ´’s color, we cannot have two adjacent red nodes at y’s new

position in the tree. In addition, if y was not ´’s right child, then y’s original

right child x replaces y in the tree. If y is red, then x must be black, and so

replacing y by x cannot cause two red nodes to become adjacent.

3. Since y could not have been the root if it was red, the root remains black.

If node y was black, three problems may arise, which the call of RB-DELETE-

FIXUP will remedy. First, if y had been the root and a red child of y becomes the

new root, we have violated property 2. Second, if both x and x:p are red, then

we have violated property 4. Third, moving y within the tree causes any simple

path that previously contained y to have one fewer black node. Thus, property 5

is now violated by any ancestor of y in the tree. We can correct the violation

of property 5 by saying that node x, now occupying y’s original position, has an

“extra” black. That is, if we add 1 to the count of black nodes on any simple path

that contains x, then under this interpretation, property 5 holds. When we remove

or move the black node y, we “push” its blackness onto node x. The problem is

that now node x is neither red nor black, thereby violating property 1. Instead,

326 Chapter 13 Red-Black Trees

node x is either “doubly black” or “red-and-black,” and it contributes either 2 or 1,

respectively, to the count of black nodes on simple paths containing x. The color

attribute of x will still be either RED (if x is red-and-black) or BLACK (if x is

doubly black). In other words, the extra black on a node is reﬂected in x’s pointing

to the node rather than in the color attribute.

We can now see the procedure RB-DELETE-FIXUP and examine how it restores

the red-black properties to the search tree.

RB-DELETE-FIXUP.T; x/

1 while x ¤ T:root and x:color == BLACK

2 if x == x:p:left

3 w D x:p:right

4 if w:color == RED

5 w:color D BLACK // case 1

6 x:p:color D RED // case 1

7 LEFT-ROTATE.T; x:p/ // case 1

8 w D x:p:right // case 1

9 if w:left:color == BLACK and w:right:color == BLACK

10 w:color D RED // case 2

11 x D x:p // case 2

12 else if w:right:color == BLACK

13 w:left:color D BLACK // case 3

14 w:color D RED // case 3

15 RIGHT-ROTATE.T; w/ // case 3

16 w D x:p:right // case 3

17 w:color D x:p:color // case 4

18 x:p:color D BLACK // case 4

19 w:right:color D BLACK // case 4

20 LEFT-ROTATE.T; x:p/ // case 4

21 x D T:root // case 4

22 else (same as then clause with “right” and “left” exchanged)

23 x:color D BLACK

The procedure RB-DELETE-FIXUP restores properties 1, 2, and 4. Exercises

13.4-1 and 13.4-2 ask you to show that the procedure restores properties 2 and 4,

and so in the remainder of this section, we shall focus on property 1. The goal of

the while loop in lines 1–22 is to move the extra black up the tree until

1. x points to a red-and-black node, in which case we color x (singly) black in

line 23;

2. x points to the root, in which case we simply “remove” the extra black; or

3. having performed suitable rotations and recolorings, we exit the loop.

13.4 Deletion 327

Within the while loop, x always points to a nonroot doubly black node. We

determine in line 2 whether x is a left child or a right child of its parent x:p. (We

have given the code for the situation in which x is a left child; the situation in

which x is a right child—line 22—is symmetric.) We maintain a pointer w to

the sibling of x. Since node x is doubly black, node w cannot be T:nil, because

otherwise, the number of blacks on the simple path from x:p to the (singly black)

leaf w would be smaller than the number on the simple path from x:p to x.

The four cases 2 in the code appear in Figure 13.7. Before examining each case

in detail, let’s look more generally at how we can verify that the transformation

in each of the cases preserves property 5. The key idea is that in each case, the

transformation applied preserves the number of black nodes (including x’s extra

black) from (and including) the root of the subtree shown to each of the subtrees

˛; ˇ; : : : ;   
. Thus, if property 5 holds prior to the transformation, it continues to

hold afterward. For example, in Figure 13.7(a), which illustrates case 1, the num-

ber of black nodes from the root to either subtree ˛ or ˇ is 3, both before and after

the transformation. (Again, remember that node x adds an extra black.) Similarly,

the number of black nodes from the root to any of , ı, ", and   
 is 2, both be-

fore and after the transformation. In Figure 13.7(b), the counting must involve the

value c of the color attribute of the root of the subtree shown, which can be either

RED or BLACK. If we deﬁne count.RED/ D 0 and count.BLACK/ D 1, then the

number of black nodes from the root to ˛ is 2 C count.c/, both before and after

the transformation. In this case, after the transformation, the new node x has color

attribute c, but this node is really either red-and-black (if c D RED) or doubly black

(if c D BLACK). You can verify the other cases similarly (see Exercise 13.4-5).

Case 1: x’s sibling w is red

Case 1 (lines 5–8 of RB-DELETE-FIXUP and Figure 13.7(a)) occurs when node w,

the sibling of node x, is red. Since w must have black children, we can switch the

colors of w and x:p and then perform a left-rotation on x:p without violating any

of the red-black properties. The new sibling of x, which is one of w’s children

prior to the rotation, is now black, and thus we have converted case 1 into case 2,

3, or 4.

Cases 2, 3, and 4 occur when node w is black; they are distinguished by the

colors of w’s children.

2

As in RB-INSERT-FIXUP, the cases in RB-DELETE-FIXUP are not mutually exclusive.

328 Chapter 13 Red-Black Trees

Case 2: x’s sibling w is black, and both of w’s children are black

In case 2 (lines 10–11 of RB-DELETE-FIXUP and Figure 13.7(b)), both of w’s

children are black. Since w is also black, we take one black off both x and w,

leaving x with only one black and leaving w red. To compensate for removing

one black from x and w, we would like to add an extra black to x:p, which was

originally either red or black. We do so by repeating the while loop with x:p as

the new node x. Observe that if we enter case 2 through case 1, the new node x

is red-and-black, since the original x:p was red. Hence, the value c of the color

attribute of the new node x is RED, and the loop terminates when it tests the loop

condition. We then color the new node x (singly) black in line 23.

Case 3: x’s sibling w is black, w’s left child is red, and w’s right child is black

Case 3 (lines 13–16 and Figure 13.7(c)) occurs when w is black, its left child

is red, and its right child is black. We can switch the colors of w and its left

child w:left and then perform a right rotation on w without violating any of the

red-black properties. The new sibling w of x is now a black node with a red right

child, and thus we have transformed case 3 into case 4.

Case 4: x’s sibling w is black, and w’s right child is red

Case 4 (lines 17–21 and Figure 13.7(d)) occurs when node x’s sibling w is black

and w’s right child is red. By making some color changes and performing a left ro-

tation on x:p, we can remove the extra black on x, making it singly black, without

violating any of the red-black properties. Setting x to be the root causes the while

loop to terminate when it tests the loop condition.

Analysis

What is the running time of RB-DELETE? Since the height of a red-black tree of n

nodes is O.lg n/, the total cost of the procedure without the call to RB-DELETE-

FIXUP takes O.lg n/ time. Within RB-DELETE-FIXUP, each of cases 1, 3, and 4

lead to termination after performing a constant number of color changes and at

most three rotations. Case 2 is the only case in which the while loop can be re-

peated, and then the pointer x moves up the tree at most O.lg n/ times, performing

no rotations. Thus, the procedure RB-DELETE-FIXUP takes O.lg n/ time and per-

forms at most three rotations, and the overall time for RB-DELETE is therefore

also O.lg n/.

13.4 Deletion 329

A

B

D

C E

α β

γ δ ε ζ

x w

A

B

C

D

E

x new w

α β γ δ

ε ζ

A

B

D

C E

α β

γ δ ε ζ

x w

c

A

B

D

C E

α β

γ δ ε ζ

c new x

A

B

D

C E

α β

γ δ ε ζ

x w

c

A

B

C

Dαβγ

δ

ε ζ

x

c

new w

A

B

D

C E

α β

γ δ

ε ζ

x w

c c

α β

A

B

C

D

E (d)

(c)

(b)

(a)

γ δ ε ζ

Case 4

Case 3

Case 2

Case 1

E

c ′ c ′

new x D T:root

Figure 13.7 The cases in the while loop of the procedure RB-DELETE-FIXUP. Darkened nodes

have color attributes BLACK, heavily shaded nodes have color attributes RED, and lightly shaded

nodes have color attributes represented by c and c

0

, which may be either RED or BLACK. The letters

˛; ˇ; : : : ;   
 represent arbitrary subtrees. Each case transforms the conﬁguration on the left into the

conﬁguration on the right by changing some colors and/or performing a rotation. Any node pointed

to by x has an extra black and is either doubly black or red-and-black. Only case 2 causes the loop to

repeat. (a) Case 1 is transformed to case 2, 3, or 4 by exchanging the colors of nodes B and D and

performing a left rotation. (b) In case 2, the extra black represented by the pointer x moves up the

tree by coloring node D red and setting x to point to node B. If we enter case 2 through case 1, the

while loop terminates because the new node x is red-and-black, and therefore the value c of its color

attribute is RED. (c) Case 3 is transformed to case 4 by exchanging the colors of nodes C and D and

performing a right rotation. (d) Case 4 removes the extra black represented by x by changing some

colors and performing a left rotation (without violating the red-black properties), and then the loop

terminates.

330 Chapter 13 Red-Black Trees

Exercises

13.4-1

Argue that after executing RB-DELETE-FIXUP, the root of the tree must be black.

13.4-2

Argue that if in RB-DELETE both x and x:p are red, then property 4 is restored by

the call to RB-DELETE-FIXUP.T; x/.

13.4-3

In Exercise 13.3-2, you found the red-black tree that results from successively

inserting the keys 41; 38; 31; 12; 19; 8 into an initially empty tree. Now show the

red-black trees that result from the successive deletion of the keys in the order

8; 12; 19; 31; 38; 41.

13.4-4

In which lines of the code for RB-DELETE-FIXUP might we examine or modify

the sentinel T:nil?

13.4-5

In each of the cases of Figure 13.7, give the count of black nodes from the root of

the subtree shown to each of the subtrees ˛; ˇ; : : : ;   
, and verify that each count

remains the same after the transformation. When a node has a color attribute c

or c 0

, use the notation count.c/ or count.c 0 / symbolically in your count.

13.4-6

Professors Skelton and Baron are concerned that at the start of case 1 of RB-

DELETE-FIXUP, the node x:p might not be black. If the professors are correct,

then lines 5–6 are wrong. Show that x:p must be black at the start of case 1, so that

the professors have nothing to worry about.

13.4-7

Suppose that a node x is inserted into a red-black tree with RB-INSERT and then

is immediately deleted with RB-DELETE. Is the resulting red-black tree the same

as the initial red-black tree? Justify your answer.

Problems for Chapter 13 331

Problems

13-1 Persistent dynamic sets

During the course of an algorithm, we sometimes ﬁnd that we need to maintain past

versions of a dynamic set as it is updated. We call such a set persistent. One way to

implement a persistent set is to copy the entire set whenever it is modiﬁed, but this

approach can slow down a program and also consume much space. Sometimes, we

can do much better.

Consider a persistent set S with the operations INSERT, DELETE, and SEARCH,

which we implement using binary search trees as shown in Figure 13.8(a). We

maintain a separate root for every version of the set. In order to insert the key 5

into the set, we create a new node with key 5. This node becomes the left child

of a new node with key 7, since we cannot modify the existing node with key 7.

Similarly, the new node with key 7 becomes the left child of a new node with

key 8 whose right child is the existing node with key 10. The new node with key 8

becomes, in turn, the right child of a new root r 0

with key 4 whose left child is the

existing node with key 3. We thus copy only part of the tree and share some of the

nodes with the original tree, as shown in Figure 13.8(b).

Assume that each tree node has the attributes key, left, and right but no parent.

(See also Exercise 13.3-6.)

4

3

2

8

7 10

4

3

2

8

7 10

4

8

7

5

(b) (a)

r r r ′

Figure 13.8 (a) A binary search tree with keys 2; 3; 4; 7; 8; 10. (b) The persistent binary search

tree that results from the insertion of key 5. The most recent version of the set consists of the nodes

reachable from the root r

0

, and the previous version consists of the nodes reachable from r. Heavily

shaded nodes are added when key 5 is inserted.

332 Chapter 13 Red-Black Trees

a. For a general persistent binary search tree, identify the nodes that we need to

change to insert a key k or delete a node y.

b. Write a procedure PERSISTENT-TREE-INSERT that, given a persistent tree T

and a key k to insert, returns a new persistent tree T 0

that is the result of insert-

ing k into T .

c. If the height of the persistent binary search tree T is h, what are the time and

space requirements of your implementation of PERSISTENT-TREE-INSERT?

(The space requirement is proportional to the number of new nodes allocated.)

d. Suppose that we had included the parent attribute in each node. In this case,

PERSISTENT-TREE-INSERT would need to perform additional copying. Prove

that PERSISTENT-TREE-INSERT would then require .n/ time and space,

where n is the number of nodes in the tree.

e. Show how to use red-black trees to guarantee that the worst-case running time

and space are O.lg n/ per insertion or deletion.

13-2 Join operation on red-black trees

The join operation takes two dynamic sets S 1 and S 2 and an element x such that

for any x 1 2 S 1 and x 2 2 S 2 , we have x 1 :key x:key x 2 :key. It returns a set

S D S 1 [ fxg [ S 2 . In this problem, we investigate how to implement the join

operation on red-black trees.

a. Given a red-black tree T , let us store its black-height as the new attribute T:bh.

Argue that RB-INSERT and RB-DELETE can maintain the bh attribute with-

out requiring extra storage in the nodes of the tree and without increasing the

asymptotic running times. Show that while descending through T , we can de-

termine the black-height of each node we visit in O.1/ time per node visited.

We wish to implement the operation RB-JOIN.T 1 ; x; T 2 /, which destroys T 1 and T 2

and returns a red-black tree T D T 1 [ fxg[ T 2 . Let n be the total number of nodes

in T 1 and T 2 .

b. Assume that T 1 :bh T 2 :bh. Describe an O.lg n/-time algorithm that ﬁnds a

black node y in T 1 with the largest key from among those nodes whose black-

height is T 2 :bh.

c. Let T y be the subtree rooted at y. Describe how T y [ fxg [ T 2 can replace T y

in O.1/ time without destroying the binary-search-tree property.

d. What color should we make x so that red-black properties 1, 3, and 5 are main-

tained? Describe how to enforce properties 2 and 4 in O.lg n/ time.

Problems for Chapter 13 333

e. Argue that no generality is lost by making the assumption in part (b). Describe

the symmetric situation that arises when T 1 :bh T 2 :bh.

f. Argue that the running time of RB-JOIN is O.lg n/.

13-3 AVL trees

An AVL tree is a binary search tree that is height balanced: for each node x, the

heights of the left and right subtrees of x differ by at most 1. To implement an AVL

tree, we maintain an extra attribute in each node: x:h is the height of node x. As

for any other binary search tree T , we assume that T:root points to the root node.

a. Prove that an AVL tree with n nodes has height O.lg n/. (Hint: Prove that

an AVL tree of height h has at least F h nodes, where F h is the hth Fibonacci

number.)

b. To insert into an AVL tree, we ﬁrst place a node into the appropriate place in bi-

nary search tree order. Afterward, the tree might no longer be height balanced.

Speciﬁcally, the heights of the left and right children of some node might differ

by 2. Describe a procedure BALANCE.x/, which takes a subtree rooted at x

whose left and right children are height balanced and have heights that differ

by at most 2, i.e., jx:right:h x:left:hj 2, and alters the subtree rooted at x

to be height balanced. (Hint: Use rotations.)

c. Using part (b), describe a recursive procedure AVL-INSERT.x; ´/ that takes

a node x within an AVL tree and a newly created node ´ (whose key has al-

ready been ﬁlled in), and adds ´ to the subtree rooted at x, maintaining the

property that x is the root of an AVL tree. As in TREE-INSERT from Sec-

tion 12.3, assume that ´:key has already been ﬁlled in and that ´:left D NIL

and ´:right D NIL; also assume that ´:h D 0. Thus, to insert the node ´ into

the AVL tree T , we call AVL-INSERT.T:root; ´/.

d. Show that AVL-INSERT, run on an n-node AVL tree, takes O.lg n/ time and

performs O.1/ rotations.

13-4 Treaps

If we insert a set of n items into a binary search tree, the resulting tree may be

horribly unbalanced, leading to long search times. As we saw in Section 12.4,

however, randomly built binary search trees tend to be balanced. Therefore, one

strategy that, on average, builds a balanced tree for a ﬁxed set of items would be to

randomly permute the items and then insert them in that order into the tree.

What if we do not have all the items at once? If we receive the items one at a

time, can we still randomly build a binary search tree out of them?

334 Chapter 13 Red-Black Trees

G: 4

B: 7 H: 5

A: 10 E: 23 K: 65

I: 73

Figure 13.9 A treap. Each node x is labeled with x:key: x:priority. For example, the root has

key G and priority 4.

We will examine a data structure that answers this question in the afﬁrmative. A

treap is a binary search tree with a modiﬁed way of ordering the nodes. Figure 13.9

shows an example. As usual, each node x in the tree has a key value x:key. In

addition, we assign x:priority, which is a random number chosen independently

for each node. We assume that all priorities are distinct and also that all keys are

distinct. The nodes of the treap are ordered so that the keys obey the binary-search-

tree property and the priorities obey the min-heap order property:

If is a left child of u, then :key < u:key.

If is a right child of u, then :key > u:key.

If is a child of u, then :priority > u:priority.

(This combination of properties is why the tree is called a “treap”: it has features

of both a binary search tree and a heap.)

It helps to think of treaps in the following way. Suppose that we insert nodes

x 1 ; x 2 ; : : : ; x n , with associated keys, into a treap. Then the resulting treap is the

tree that would have been formed if the nodes had been inserted into a normal

binary search tree in the order given by their (randomly chosen) priorities, i.e.,

x i :priority < x j :priority means that we had inserted x i before x j .

a. Show that given a set of nodes x 1 ; x 2 ; : : : ; x n , with associated keys and priori-

ties, all distinct, the treap associated with these nodes is unique.

b. Show that the expected height of a treap is ‚.lg n/, and hence the expected time

to search for a value in the treap is ‚.lg n/.

Let us see how to insert a new node into an existing treap. The ﬁrst thing we do

is assign to the new node a random priority. Then we call the insertion algorithm,

which we call TREAP-INSERT, whose operation is illustrated in Figure 13.10.

Problems for Chapter 13 335

G: 4

B: 7 H: 5

A: 10 E: 23 K: 65

I: 73

G: 4

B: 7 H: 5

A: 10 E: 23 K: 65

I: 73

C: 25

C: 25

(a) (b)

G: 4

B: 7 H: 5

A: 10 E: 23 K: 65

I: 73 C: 25

(c)

D: 9

D: 9

G: 4

B: 7 H: 5

A: 10 E: 23 K: 65

I: 73

(d)

D: 9

C: 25

G: 4

B: 7 H: 5

A: 10 K: 65

I: 73

(e)

D: 9

C: 25 E: 23

B: 7

A: 10

(f)

D: 9

C: 25 E: 23

F: 2

I: 73

K: 65

H: 5

G: 4

F: 2

…

Figure 13.10 The operation of TREAP-INSERT. (a) The original treap, prior to insertion. (b) The

treap after inserting a node with key C and priority 25. (c)–(d) Intermediate stages when inserting a

node with key D and priority 9. (e) The treap after the insertion of parts (c) and (d) is done. (f) The

treap after inserting a node with key F and priority 2.

336 Chapter 13 Red-Black Trees

15

9 18

3 12 25

21 6

(a)

15

9 18

3 12 25

21 6

(b)

Figure 13.11 Spines of a binary search tree. The left spine is shaded in (a), and the right spine is

shaded in (b).

c. Explain how TREAP-INSERT works. Explain the idea in English and give pseu-

docode. (Hint: Execute the usual binary-search-tree insertion procedure and

then perform rotations to restore the min-heap order property.)

d. Show that the expected running time of TREAP-INSERT is ‚.lg n/.

TREAP-INSERT performs a search and then a sequence of rotations. Although

these two operations have the same expected running time, they have different

costs in practice. A search reads information from the treap without modifying it.

In contrast, a rotation changes parent and child pointers within the treap. On most

computers, read operations are much faster than write operations. Thus we would

like TREAP-INSERT to perform few rotations. We will show that the expected

number of rotations performed is bounded by a constant.

In order to do so, we will need some deﬁnitions, which Figure 13.11 depicts.

The left spine of a binary search tree T is the simple path from the root to the node

with the smallest key. In other words, the left spine is the simple path from the

root that consists of only left edges. Symmetrically, the right spine of T is the

simple path from the root consisting of only right edges. The length of a spine is

the number of nodes it contains.

e. Consider the treap T immediately after TREAP-INSERT has inserted node x.

Let C be the length of the right spine of the left subtree of x. Let D be the

length of the left spine of the right subtree of x. Prove that the total number of

rotations that were performed during the insertion of x is equal to C C D.

We will now calculate the expected values of C and D. Without loss of generality,

we assume that the keys are 1; 2; : : : ; n, since we are comparing them only to one

another.

Notes for Chapter 13 337

For nodes x and y in treap T , where y ¤ x, let k D x:key and i D y:key. We

deﬁne indicator random variables

X ik D I fy is in the right spine of the left subtree of xg :

f. Show that X ik D 1 if and only if y:priority > x:priority, y:key < x:key, and,

for every ´ such that y:key < ´:key < x:key, we have y:priority < ´:priority.

g. Show that

Pr fX ik D 1g D

.k i 1/Š

.k i C 1/Š

D

1

.k i C 1/.k i/

:

h. Show that

E ŒC D

k1 X

j D1

1

j.j C 1/

D 1

1

k

:

i. Use a symmetry argument to show that

E ŒD D 1

1

n k C 1

:

j. Conclude that the expected number of rotations performed when inserting a

node into a treap is less than 2.

Chapter notes

The idea of balancing a search tree is due to Adel’son-Vel’ski˘ı and Landis [2], who

introduced a class of balanced search trees called “AVL trees” in 1962, described in

Problem 13-3. Another class of search trees, called “2-3 trees,” was introduced by

J. E. Hopcroft (unpublished) in 1970. A 2-3 tree maintains balance by manipulating

the degrees of nodes in the tree. Chapter 18 covers a generalization of 2-3 trees

introduced by Bayer and McCreight [35], called “B-trees.”

Red-black trees were invented by Bayer [34] under the name “symmetric binary

B-trees.” Guibas and Sedgewick [155] studied their properties at length and in-

troduced the red/black color convention. Andersson [15] gives a simpler-to-code

338 Chapter 13 Red-Black Trees

variant of red-black trees. Weiss [351] calls this variant AA-trees. An AA-tree is

similar to a red-black tree except that left children may never be red.

Treaps, the subject of Problem 13-4, were proposed by Seidel and Aragon [309].

They are the default implementation of a dictionary in LEDA [253], which is a

well-implemented collection of data structures and algorithms.

There are many other variations on balanced binary trees, including weight-

balanced trees [264], k-neighbor trees [245], and scapegoat trees [127]. Perhaps

the most intriguing are the “splay trees” introduced by Sleator and Tarjan [320],

which are “self-adjusting.” (See Tarjan [330] for a good description of splay trees.)

Splay trees maintain balance without any explicit balance condition such as color.

Instead, “splay operations” (which involve rotations) are performed within the tree

every time an access is made. The amortized cost (see Chapter 17) of each opera-

tion on an n-node tree is O.lg n/.

Skip lists [286] provide an alternative to balanced binary trees. A skip list is a

linked list that is augmented with a number of additional pointers. Each dictionary

operation runs in expected time O.lg n/ on a skip list of n items.

14 Augmenting Data Structures

Some engineering situations require no more than a “textbook” data struc-

ture—such as a doubly linked list, a hash table, or a binary search tree—but many

others require a dash of creativity. Only in rare situations will you need to cre-

ate an entirely new type of data structure, though. More often, it will sufﬁce to

augment a textbook data structure by storing additional information in it. You can

then program new operations for the data structure to support the desired applica-

tion. Augmenting a data structure is not always straightforward, however, since the

added information must be updated and maintained by the ordinary operations on

the data structure.

This chapter discusses two data structures that we construct by augmenting red-

black trees. Section 14.1 describes a data structure that supports general order-

statistic operations on a dynamic set. We can then quickly ﬁnd the ith smallest

number in a set or the rank of a given element in the total ordering of the set.

Section 14.2 abstracts the process of augmenting a data structure and provides a

theorem that can simplify the process of augmenting red-black trees. Section 14.3

uses this theorem to help design a data structure for maintaining a dynamic set of

intervals, such as time intervals. Given a query interval, we can then quickly ﬁnd

an interval in the set that overlaps it.

14.1 Dynamic order statistics

Chapter 9 introduced the notion of an order statistic. Speciﬁcally, the ith order

statistic of a set of n elements, where i 2 f1; 2; : : : ; ng, is simply the element in the

set with the ith smallest key. We saw how to determine any order statistic in O.n/

time from an unordered set. In this section, we shall see how to modify red-black

trees so that we can determine any order statistic for a dynamic set in O.lg n/ time.

We shall also see how to compute the rank of an element—its position in the linear

order of the set—in O.lg n/ time.

340 Chapter 14 Augmenting Data Structures

1

3

7 12

10

14

16

14

2 1 1

2 4

7

20

19 21

21

17

28

35 39

38

47 30

41

26

1

2 1

4

12

1

1 1

3

5 1

7

20

key

size

Figure 14.1 An order-statistic tree, which is an augmented red-black tree. Shaded nodes are red,

and darkened nodes are black. In addition to its usual attributes, each node x has an attribute x:size,

which is the number of nodes, other than the sentinel, in the subtree rooted at x.

Figure 14.1 shows a data structure that can support fast order-statistic operations.

An order-statistic tree T is simply a red-black tree with additional information

stored in each node. Besides the usual red-black tree attributes x:key, x:color, x:p,

x:left, and x:right in a node x, we have another attribute, x:size. This attribute

contains the number of (internal) nodes in the subtree rooted at x (including x

itself), that is, the size of the subtree. If we deﬁne the sentinel’s size to be 0—that

is, we set T:nil:size to be 0—then we have the identity

x:size D x:left:size C x:right:size C 1 :

We do not require keys to be distinct in an order-statistic tree. (For example, the

tree in Figure 14.1 has two keys with value 14 and two keys with value 21.) In the

presence of equal keys, the above notion of rank is not well deﬁned. We remove

this ambiguity for an order-statistic tree by deﬁning the rank of an element as the

position at which it would be printed in an inorder walk of the tree. In Figure 14.1,

for example, the key 14 stored in a black node has rank 5, and the key 14 stored in

a red node has rank 6.

Retrieving an element with a given rank

Before we show how to maintain this size information during insertion and dele-

tion, let us examine the implementation of two order-statistic queries that use this

additional information. We begin with an operation that retrieves an element with

a given rank. The procedure OS-SELECT.x; i/ returns a pointer to the node con-

taining the ith smallest key in the subtree rooted at x. To ﬁnd the node with the ith

smallest key in an order-statistic tree T , we call OS-SELECT.T:root; i/.

14.1 Dynamic order statistics 341

OS-SELECT.x; i/

1 r D x:left:size C 1

2 if i == r

3 return x

4 elseif i < r

5 return OS-SELECT.x:left; i/

6 else return OS-SELECT.x:right; i r/

In line 1 of OS-SELECT, we compute r, the rank of node x within the subtree

rooted at x. The value of x:left:size is the number of nodes that come before x

in an inorder tree walk of the subtree rooted at x. Thus, x:left:size C 1 is the

rank of x within the subtree rooted at x. If i D r, then node x is the ith smallest

element, and so we return x in line 3. If i < r, then the ith smallest element

resides in x’s left subtree, and so we recurse on x:left in line 5. If i > r, then

the ith smallest element resides in x’s right subtree. Since the subtree rooted at x

contains r elements that come before x’s right subtree in an inorder tree walk, the

ith smallest element in the subtree rooted at x is the .i r/th smallest element in

the subtree rooted at x:right. Line 6 determines this element recursively.

To see how OS-SELECT operates, consider a search for the 17th smallest ele-

ment in the order-statistic tree of Figure 14.1. We begin with x as the root, whose

key is 26, and with i D 17. Since the size of 26’s left subtree is 12, its rank is 13.

Thus, we know that the node with rank 17 is the 17 13 D 4th smallest element

in 26’s right subtree. After the recursive call, x is the node with key 41, and i D 4.

Since the size of 41’s left subtree is 5, its rank within its subtree is 6. Thus, we

know that the node with rank 4 is the 4th smallest element in 41’s left subtree. Af-

ter the recursive call, x is the node with key 30, and its rank within its subtree is 2.

Thus, we recurse once again to ﬁnd the 42 D 2nd smallest element in the subtree

rooted at the node with key 38. We now ﬁnd that its left subtree has size 1, which

means it is the second smallest element. Thus, the procedure returns a pointer to

the node with key 38.

Because each recursive call goes down one level in the order-statistic tree, the

total time for OS-SELECT is at worst proportional to the height of the tree. Since

the tree is a red-black tree, its height is O.lg n/, where n is the number of nodes.

Thus, the running time of OS-SELECT is O.lg n/ for a dynamic set of n elements.

Determining the rank of an element

Given a pointer to a node x in an order-statistic tree T , the procedure OS-RANK

returns the position of x in the linear order determined by an inorder tree walk

of T .

342 Chapter 14 Augmenting Data Structures

OS-RANK.T; x/

1 r D x:left:size C 1

2 y D x

3 while y ¤ T:root

4 if y == y:p:right

5 r D r C y:p:left:size C 1

6 y D y:p

7 return r

The procedure works as follows. We can think of node x’s rank as the number of

nodes preceding x in an inorder tree walk, plus 1 for x itself. OS-RANK maintains

the following loop invariant:

At the start of each iteration of the while loop of lines 3–6, r is the rank

of x:key in the subtree rooted at node y.

We use this loop invariant to show that OS-RANK works correctly as follows:

Initialization: Prior to the ﬁrst iteration, line 1 sets r to be the rank of x:key within

the subtree rooted at x. Setting y D x in line 2 makes the invariant true the

ﬁrst time the test in line 3 executes.

Maintenance: At the end of each iteration of the while loop, we set y D y:p.

Thus we must show that if r is the rank of x:key in the subtree rooted at y at the

start of the loop body, then r is the rank of x:key in the subtree rooted at y:p

at the end of the loop body. In each iteration of the while loop, we consider

the subtree rooted at y:p. We have already counted the number of nodes in the

subtree rooted at node y that precede x in an inorder walk, and so we must add

the nodes in the subtree rooted at y’s sibling that precede x in an inorder walk,

plus 1 for y:p if it, too, precedes x. If y is a left child, then neither y:p nor any

node in y:p’s right subtree precedes x, and so we leave r alone. Otherwise, y is

a right child and all the nodes in y:p’s left subtree precede x, as does y:p itself.

Thus, in line 5, we add y:p:left:size C 1 to the current value of r.

Termination: The loop terminates when y D T:root, so that the subtree rooted

at y is the entire tree. Thus, the value of r is the rank of x:key in the entire tree.

As an example, when we run OS-RANK on the order-statistic tree of Figure 14.1

to ﬁnd the rank of the node with key 38, we get the following sequence of values

of y:key and r at the top of the while loop:

iteration y:key r

1 38 2

2 30 4

3 41 4

4 26 17

14.1 Dynamic order statistics 343

The procedure returns the rank 17.

Since each iteration of the while loop takes O.1/ time, and y goes up one level in

the tree with each iteration, the running time of OS-RANK is at worst proportional

to the height of the tree: O.lg n/ on an n-node order-statistic tree.

Maintaining subtree sizes

Given the size attribute in each node, OS-SELECT and OS-RANK can quickly

compute order-statistic information. But unless we can efﬁciently maintain these

attributes within the basic modifying operations on red-black trees, our work will

have been for naught. We shall now show how to maintain subtree sizes for both

insertion and deletion without affecting the asymptotic running time of either op-

eration.

We noted in Section 13.3 that insertion into a red-black tree consists of two

phases. The ﬁrst phase goes down the tree from the root, inserting the new node

as a child of an existing node. The second phase goes up the tree, changing colors

and performing rotations to maintain the red-black properties.

To maintain the subtree sizes in the ﬁrst phase, we simply increment x:size for

each node x on the simple path traversed from the root down toward the leaves. The

new node added gets a size of 1. Since there are O.lg n/ nodes on the traversed

path, the additional cost of maintaining the size attributes is O.lg n/.

In the second phase, the only structural changes to the underlying red-black tree

are caused by rotations, of which there are at most two. Moreover, a rotation is

a local operation: only two nodes have their size attributes invalidated. The link

around which the rotation is performed is incident on these two nodes. Referring

to the code for LEFT-ROTATE.T; x/ in Section 13.2, we add the following lines:

13 y:size D x:size

14 x:size D x:left:size C x:right:size C 1

Figure 14.2 illustrates how the attributes are updated. The change to RIGHT-

ROTATE is symmetric.

Since at most two rotations are performed during insertion into a red-black tree,

we spend only O.1/ additional time updating size attributes in the second phase.

Thus, the total time for insertion into an n-node order-statistic tree is O.lg n/,

which is asymptotically the same as for an ordinary red-black tree.

Deletion from a red-black tree also consists of two phases: the ﬁrst operates

on the underlying search tree, and the second causes at most three rotations and

otherwise performs no structural changes. (See Section 13.4.) The ﬁrst phase

either removes one node y from the tree or moves upward it within the tree. To

update the subtree sizes, we simply traverse a simple path from node y (starting

from its original position within the tree) up to the root, decrementing the size

344 Chapter 14 Augmenting Data Structures

LEFT-ROTATE(T, x)

RIGHT-ROTATE(T, y)

93

19

y

42

11

x

6 4

7

93

42

19

12

6

4 7

x

y

Figure 14.2 Updating subtree sizes during rotations. The link around which we rotate is incident

on the two nodes whose size attributes need to be updated. The updates are local, requiring only the

size information stored in x, y, and the roots of the subtrees shown as triangles.

attribute of each node on the path. Since this path has length O.lg n/ in an n-

node red-black tree, the additional time spent maintaining size attributes in the ﬁrst

phase is O.lg n/. We handle the O.1/ rotations in the second phase of deletion

in the same manner as for insertion. Thus, both insertion and deletion, including

maintaining the size attributes, take O.lg n/ time for an n-node order-statistic tree.

Exercises

14.1-1

Show how OS-SELECT.T:root; 10/ operates on the red-black tree T of Fig-

ure 14.1.

14.1-2

Show how OS-RANK.T; x/ operates on the red-black tree T of Figure 14.1 and

the node x with x:key D 35.

14.1-3

Write a nonrecursive version of OS-SELECT.

14.1-4

Write a recursive procedure OS-KEY-RANK.T; k/ that takes as input an order-

statistic tree T and a key k and returns the rank of k in the dynamic set represented

by T . Assume that the keys of T are distinct.

14.1-5

Given an element x in an n-node order-statistic tree and a natural number i, how

can we determine the ith successor of x in the linear order of the tree in O.lg n/

time?

14.2 How to augment a data structure 345

14.1-6

Observe that whenever we reference the size attribute of a node in either OS-

SELECT or OS-RANK, we use it only to compute a rank. Accordingly, suppose

we store in each node its rank in the subtree of which it is the root. Show how to

maintain this information during insertion and deletion. (Remember that these two

operations can cause rotations.)

14.1-7

Show how to use an order-statistic tree to count the number of inversions (see

Problem 2-4) in an array of size n in time O.n lg n/.

14.1-8 ?

Consider n chords on a circle, each deﬁned by its endpoints. Describe an O.n lg n/-

time algorithm to determine the number of pairs of chords that intersect inside the

circle. (For example, if the n chords are all diameters that meet at the center, then

the correct answer is

n

2

.) Assume that no two chords share an endpoint.

14.2 How to augment a data structure

The process of augmenting a basic data structure to support additional functionality

occurs quite frequently in algorithm design. We shall use it again in the next section

to design a data structure that supports operations on intervals. In this section, we

examine the steps involved in such augmentation. We shall also prove a theorem

that allows us to augment red-black trees easily in many cases.

We can break the process of augmenting a data structure into four steps:

1. Choose an underlying data structure.

2. Determine additional information to maintain in the underlying data structure.

3. Verify that we can maintain the additional information for the basic modifying

operations on the underlying data structure.

4. Develop new operations.

As with any prescriptive design method, you should not blindly follow the steps

in the order given. Most design work contains an element of trial and error, and

progress on all steps usually proceeds in parallel. There is no point, for example, in

determining additional information and developing new operations (steps 2 and 4)

if we will not be able to maintain the additional information efﬁciently. Neverthe-

less, this four-step method provides a good focus for your efforts in augmenting

a data structure, and it is also a good way to organize the documentation of an

augmented data structure.

346 Chapter 14 Augmenting Data Structures

We followed these steps in Section 14.1 to design our order-statistic trees. For

step 1, we chose red-black trees as the underlying data structure. A clue to the

suitability of red-black trees comes from their efﬁcient support of other dynamic-

set operations on a total order, such as MINIMUM, MAXIMUM, SUCCESSOR, and

PREDECESSOR.

For step 2, we added the size attribute, in which each node x stores the size of the

subtree rooted at x. Generally, the additional information makes operations more

efﬁcient. For example, we could have implemented OS-SELECT and OS-RANK

using just the keys stored in the tree, but they would not have run in O.lg n/ time.

Sometimes, the additional information is pointer information rather than data, as

in Exercise 14.2-1.

For step 3, we ensured that insertion and deletion could maintain the size at-

tributes while still running in O.lg n/ time. Ideally, we should need to update only

a few elements of the data structure in order to maintain the additional information.

For example, if we simply stored in each node its rank in the tree, the OS-SELECT

and OS-RANK procedures would run quickly, but inserting a new minimum ele-

ment would cause a change to this information in every node of the tree. When we

store subtree sizes instead, inserting a new element causes information to change

in only O.lg n/ nodes.

For step 4, we developed the operations OS-SELECT and OS-RANK. After all,

the need for new operations is why we bother to augment a data structure in the ﬁrst

place. Occasionally, rather than developing new operations, we use the additional

information to expedite existing ones, as in Exercise 14.2-1.

Augmenting red-black trees

When red-black trees underlie an augmented data structure, we can prove that in-

sertion and deletion can always efﬁciently maintain certain kinds of additional in-

formation, thereby making step 3 very easy. The proof of the following theorem is

similar to the argument from Section 14.1 that we can maintain the size attribute

for order-statistic trees.

Theorem 14.1 (Augmenting a red-black tree)

Let f be an attribute that augments a red-black tree T of n nodes, and suppose that

the value of f for each node x depends on only the information in nodes x, x:left,

and x:right, possibly including x:left:f and x:right:f . Then, we can maintain the

values of f in all nodes of T during insertion and deletion without asymptotically

affecting the O.lg n/ performance of these operations.

Proof The main idea of the proof is that a change to an f attribute in a node x

propagates only to ancestors of x in the tree. That is, changing x:f may re-

14.2 How to augment a data structure 347

quire x:p:f to be updated, but nothing else; updating x:p:f may require x:p:p:f

to be updated, but nothing else; and so on up the tree. Once we have updated

T:root:f , no other node will depend on the new value, and so the process termi-

nates. Since the height of a red-black tree is O.lg n/, changing an f attribute in a

node costs O.lg n/ time in updating all nodes that depend on the change.

Insertion of a node x into T consists of two phases. (See Section 13.3.) The

ﬁrst phase inserts x as a child of an existing node x:p. We can compute the value

of x:f in O.1/ time since, by supposition, it depends only on information in the

other attributes of x itself and the information in x’s children, but x’s children are

both the sentinel T:nil. Once we have computed x:f , the change propagates up

the tree. Thus, the total time for the ﬁrst phase of insertion is O.lg n/. During the

second phase, the only structural changes to the tree come from rotations. Since

only two nodes change in a rotation, the total time for updating the f attributes

is O.lg n/ per rotation. Since the number of rotations during insertion is at most

two, the total time for insertion is O.lg n/.

Like insertion, deletion has two phases. (See Section 13.4.) In the ﬁrst phase,

changes to the tree occur when the deleted node is removed from the tree. If the

deleted node had two children at the time, then its successor moves into the position

of the deleted node. Propagating the updates to f caused by these changes costs

at most O.lg n/, since the changes modify the tree locally. Fixing up the red-black

tree during the second phase requires at most three rotations, and each rotation

requires at most O.lg n/ time to propagate the updates to f . Thus, like insertion,

the total time for deletion is O.lg n/.

In many cases, such as maintaining the size attributes in order-statistic trees, the

cost of updating after a rotation is O.1/, rather than the O.lg n/ derived in the proof

of Theorem 14.1. Exercise 14.2-3 gives an example.

Exercises

14.2-1

Show, by adding pointers to the nodes, how to support each of the dynamic-set

queries MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in O.1/ worst-

case time on an augmented order-statistic tree. The asymptotic performance of

other operations on order-statistic trees should not be affected.

14.2-2

Can we maintain the black-heights of nodes in a red-black tree as attributes in the

nodes of the tree without affecting the asymptotic performance of any of the red-

black tree operations? Show how, or argue why not. How about maintaining the

depths of nodes?

348 Chapter 14 Augmenting Data Structures

14.2-3 ?

Let ˝ be an associative binary operator, and let a be an attribute maintained in each

node of a red-black tree. Suppose that we want to include in each node x an addi-

tional attribute f such that x:f D x 1 :a ˝ x 2 :a ˝ ˝ x m :a, where x 1 ; x 2 ; : : : ; x m

is the inorder listing of nodes in the subtree rooted at x. Show how to update the f

attributes in O.1/ time after a rotation. Modify your argument slightly to apply it

to the size attributes in order-statistic trees.

14.2-4 ?

We wish to augment red-black trees with an operation RB-ENUMERATE.x; a; b/

that outputs all the keys k such that a k b in a red-black tree rooted at x.

Describe how to implement RB-ENUMERATE in ‚.mClg n/ time, where m is the

number of keys that are output and n is the number of internal nodes in the tree.

(Hint: You do not need to add new attributes to the red-black tree.)

14.3 Interval trees

In this section, we shall augment red-black trees to support operations on dynamic

sets of intervals. A closed interval is an ordered pair of real numbers Œt 1 ; t 2 , with

t 1 t 2 . The interval Œt 1 ; t 2 represents the set ft 2 R W t 1 t t 2 g. Open and

half-open intervals omit both or one of the endpoints from the set, respectively. In

this section, we shall assume that intervals are closed; extending the results to open

and half-open intervals is conceptually straightforward.

Intervals are convenient for representing events that each occupy a continuous

period of time. We might, for example, wish to query a database of time intervals

to ﬁnd out what events occurred during a given interval. The data structure in this

section provides an efﬁcient means for maintaining such an interval database.

We can represent an interval Œt 1 ; t 2 as an object i, with attributes i:low D t 1

(the low endpoint) and i:high D t 2 (the high endpoint). We say that intervals i

and i 0

overlap if i \ i 0 ¤ ;, that is, if i:low i 0 :high and i 0 :low i:high. As

Figure 14.3 shows, any two intervals i and i 0

satisfy the interval trichotomy; that

is, exactly one of the following three properties holds:

a. i and i 0

overlap,

b. i is to the left of i 0

(i.e., i:high < i 0 :low),

c. i is to the right of i 0

(i.e., i 0 :high < i:low).

An interval tree is a red-black tree that maintains a dynamic set of elements, with

each element x containing an interval x:int. Interval trees support the following

operations:

14.3 Interval trees 349

i i i i

(a)

i

(b)

i

(c)

i ′ i ′ i ′ i ′

i ′ i ′

Figure 14.3 The interval trichotomy for two closed intervals i and i

0

. (a) If i and i

0

overlap, there

are four situations; in each, i:low i

0

:high and i

0

:low i:high. (b) The intervals do not overlap,

and i:high < i

0

:low. (c) The intervals do not overlap, and i

0

:high < i:low.

INTERVAL-INSERT.T; x/ adds the element x, whose int attribute is assumed to

contain an interval, to the interval tree T .

INTERVAL-DELETE.T; x/ removes the element x from the interval tree T .

INTERVAL-SEARCH.T; i/ returns a pointer to an element x in the interval tree T

such that x:int overlaps interval i, or a pointer to the sentinel T:nil if no such

element is in the set.

Figure 14.4 shows how an interval tree represents a set of intervals. We shall track

the four-step method from Section 14.2 as we review the design of an interval tree

and the operations that run on it.

Step 1: Underlying data structure

We choose a red-black tree in which each node x contains an interval x:int and the

key of x is the low endpoint, x:int:low, of the interval. Thus, an inorder tree walk

of the data structure lists the intervals in sorted order by low endpoint.

Step 2: Additional information

In addition to the intervals themselves, each node x contains a value x:max, which

is the maximum value of any interval endpoint stored in the subtree rooted at x.

Step 3: Maintaining the information

We must verify that insertion and deletion take O.lg n/ time on an interval tree

of n nodes. We can determine x:max given interval x:int and the max values of

node x’s children:

350 Chapter 14 Augmenting Data Structures

0 5 10 15 20 25 30

0

5

6

8

15

16

17

19

25

26 26

30

20

19

21

23

9

10

8

3

(a)

[0,3]

3

[6,10]

10

[5,8]

10

[8,9]

23

[15,23]

23

[16,21]

30

[17,19]

20

[26,26]

26

[19,20]

20

(b)

[25,30]

30

int

max

Figure 14.4 An interval tree. (a) A set of 10 intervals, shown sorted bottom to top by left endpoint.

(b) The interval tree that represents them. Each node x contains an interval, shown above the dashed

line, and the maximum value of any interval endpoint in the subtree rooted at x, shown below the

dashed line. An inorder tree walk of the tree lists the nodes in sorted order by left endpoint.

x:max D max.x:int:high; x:left:max; x:right:max/ :

Thus, by Theorem 14.1, insertion and deletion run in O.lg n/ time. In fact, we

can update the max attributes after a rotation in O.1/ time, as Exercises 14.2-3

and 14.3-1 show.

Step 4: Developing new operations

The only new operation we need is INTERVAL-SEARCH.T; i/, which ﬁnds a node

in tree T whose interval overlaps interval i. If there is no interval that overlaps i in

the tree, the procedure returns a pointer to the sentinel T:nil.

14.3 Interval trees 351

INTERVAL-SEARCH.T; i/

1 x D T:root

2 while x ¤ T:nil and i does not overlap x:int

3 if x:left ¤ T:nil and x:left:max i:low

4 x D x:left

5 else x D x:right

6 return x

The search for an interval that overlaps i starts with x at the root of the tree and

proceeds downward. It terminates when either it ﬁnds an overlapping interval or x

points to the sentinel T:nil. Since each iteration of the basic loop takes O.1/ time,

and since the height of an n-node red-black tree is O.lg n/, the INTERVAL-SEARCH

procedure takes O.lg n/ time.

Before we see why INTERVAL-SEARCH is correct, let’s examine how it works

on the interval tree in Figure 14.4. Suppose we wish to ﬁnd an interval that overlaps

the interval i D Œ22; 25. We begin with x as the root, which contains Œ16; 21 and

does not overlap i. Since x:left:max D 23 is greater than i:low D 22, the loop

continues with x as the left child of the root—the node containing Œ8; 9, which also

does not overlap i. This time, x:left:max D 10 is less than i:low D 22, and so the

loop continues with the right child of x as the new x. Because the interval Œ15; 23

stored in this node overlaps i, the procedure returns this node.

As an example of an unsuccessful search, suppose we wish to ﬁnd an interval

that overlaps i D Œ11; 14 in the interval tree of Figure 14.4. We once again be-

gin with x as the root. Since the root’s interval Œ16; 21 does not overlap i, and

since x:left:max D 23 is greater than i:low D 11, we go left to the node con-

taining Œ8; 9. Interval Œ8; 9 does not overlap i, and x:left:max D 10 is less than

i:low D 11, and so we go right. (Note that no interval in the left subtree over-

laps i.) Interval Œ15; 23 does not overlap i, and its left child is T:nil, so again we

go right, the loop terminates, and we return the sentinel T:nil.

To see why INTERVAL-SEARCH is correct, we must understand why it sufﬁces

to examine a single path from the root. The basic idea is that at any node x,

if x:int does not overlap i, the search always proceeds in a safe direction: the

search will deﬁnitely ﬁnd an overlapping interval if the tree contains one. The

following theorem states this property more precisely.

Theorem 14.2

Any execution of INTERVAL-SEARCH.T; i/ either returns a node whose interval

overlaps i, or it returns T:nil and the tree T contains no node whose interval over-

laps i.

352 Chapter 14 Augmenting Data Structures

i

(a) (b)

i ′

i ′ i i ′

i ′′

i ′′

i ′′

Figure 14.5 Intervals in the proof of Theorem 14.2. The value of x:left:max is shown in each case

as a dashed line. (a) The search goes right. No interval i

0

in x’s left subtree can overlap i. (b) The

search goes left. The left subtree of x contains an interval that overlaps i (situation not shown),

or x’s left subtree contains an interval i

0

such that i

0

:high D x:left:max. Since i does not overlap i

0

,

neither does it overlap any interval i

00

in x’s right subtree, since i

0

:low i

00

:low.

Proof The while loop of lines 2–5 terminates either when x D T:nil or i over-

laps x:int. In the latter case, it is certainly correct to return x. Therefore, we focus

on the former case, in which the while loop terminates because x D T:nil.

We use the following invariant for the while loop of lines 2–5:

If tree T contains an interval that overlaps i, then the subtree rooted at x

contains such an interval.

We use this loop invariant as follows:

Initialization: Prior to the ﬁrst iteration, line 1 sets x to be the root of T , so that

the invariant holds.

Maintenance: Each iteration of the while loop executes either line 4 or line 5. We

shall show that both cases maintain the loop invariant.

If line 5 is executed, then because of the branch condition in line 3, we

have x:left D T:nil, or x:left:max < i:low. If x:left D T:nil, the subtree

rooted at x:left clearly contains no interval that overlaps i, and so setting x

to x:right maintains the invariant. Suppose, therefore, that x:left ¤ T:nil and

x:left:max < i:low. As Figure 14.5(a) shows, for each interval i 0

in x’s left

subtree, we have

i

0

:high x:left:max

< i:low :

By the interval trichotomy, therefore, i 0

and i do not overlap. Thus, the left

subtree of x contains no intervals that overlap i, so that setting x to x:right

maintains the invariant.

14.3 Interval trees 353

If, on the other hand, line 4 is executed, then we will show that the contrapos-

itive of the loop invariant holds. That is, if the subtree rooted at x:left con-

tains no interval overlapping i, then no interval anywhere in the tree overlaps i.

Since line 4 is executed, then because of the branch condition in line 3, we

have x:left:max i:low. Moreover, by deﬁnition of the max attribute, x’s left

subtree must contain some interval i 0

such that

i

0

:high D x:left:max

i:low :

(Figure 14.5(b) illustrates the situation.) Since i and i 0

do not overlap, and

since it is not true that i 0 :high < i:low, it follows by the interval trichotomy

that i:high < i 0 :low. Interval trees are keyed on the low endpoints of intervals,

and thus the search-tree property implies that for any interval i 00

in x’s right

subtree,

i:high < i

0

:low

i

00

:low :

By the interval trichotomy, i and i 00

do not overlap. We conclude that whether

or not any interval in x’s left subtree overlaps i, setting x to x:left maintains

the invariant.

Termination: If the loop terminates when x D T:nil, then the subtree rooted at x

contains no interval overlapping i. The contrapositive of the loop invariant

implies that T contains no interval that overlaps i. Hence it is correct to return

x D T:nil.

Thus, the INTERVAL-SEARCH procedure works correctly.

Exercises

14.3-1

Write pseudocode for LEFT-ROTATE that operates on nodes in an interval tree and

updates the max attributes in O.1/ time.

14.3-2

Rewrite the code for INTERVAL-SEARCH so that it works properly when all inter-

vals are open.

14.3-3

Describe an efﬁcient algorithm that, given an interval i, returns an interval over-

lapping i that has the minimum low endpoint, or T:nil if no such interval exists.

354 Chapter 14 Augmenting Data Structures

14.3-4

Given an interval tree T and an interval i, describe how to list all intervals in T

that overlap i in O.min.n; k lg n// time, where k is the number of intervals in the

output list. (Hint: One simple method makes several queries, modifying the tree

between queries. A slightly more complicated method does not modify the tree.)

14.3-5

Suggest modiﬁcations to the interval-tree procedures to support the new opera-

tion INTERVAL-SEARCH-EXACTLY.T; i/, where T is an interval tree and i is

an interval. The operation should return a pointer to a node x in T such that

x:int:low D i:low and x:int:high D i:high, or T:nil if T contains no such node.

All operations, including INTERVAL-SEARCH-EXACTLY, should run in O.lg n/

time on an n-node interval tree.

14.3-6

Show how to maintain a dynamic set Q of numbers that supports the operation

MIN-GAP, which gives the magnitude of the difference of the two closest num-

bers in Q. For example, if Q D f1; 5; 9; 15; 18; 22g, then MIN-GAP.Q/ returns

18 15 D 3, since 15 and 18 are the two closest numbers in Q. Make the op-

erations INSERT, DELETE, SEARCH, and MIN-GAP as efﬁcient as possible, and

analyze their running times.

14.3-7 ?

VLSI databases commonly represent an integrated circuit as a list of rectan-

gles. Assume that each rectangle is rectilinearly oriented (sides parallel to the

x- and y-axes), so that we represent a rectangle by its minimum and maximum x-

and y-coordinates. Give an O.n lg n/-time algorithm to decide whether or not a set

of n rectangles so represented contains two rectangles that overlap. Your algorithm

need not report all intersecting pairs, but it must report that an overlap exists if one

rectangle entirely covers another, even if the boundary lines do not intersect. (Hint:

Move a “sweep” line across the set of rectangles.)

Problems

14-1 Point of maximum overlap

Suppose that we wish to keep track of a point of maximum overlap in a set of

intervals—a point with the largest number of intervals in the set that overlap it.

a. Show that there will always be a point of maximum overlap that is an endpoint

of one of the segments.

Notes for Chapter 14 355

b. Design a data structure that efﬁciently supports the operations INTERVAL-

INSERT, INTERVAL-DELETE, and FIND-POM, which returns a point of max-

imum overlap. (Hint: Keep a red-black tree of all the endpoints. Associate

a value of C1 with each left endpoint, and associate a value of 1 with each

right endpoint. Augment each node of the tree with some extra information to

maintain the point of maximum overlap.)

14-2 Josephus permutation

We deﬁne the Josephus problem as follows. Suppose that n people form a circle

and that we are given a positive integer m n. Beginning with a designated

ﬁrst person, we proceed around the circle, removing every mth person. After each

person is removed, counting continues around the circle that remains. This process

continues until we have removed all n people. The order in which the people are

removed from the circle deﬁnes the .n; m/-Josephus permutation of the integers

1; 2; : : : ; n. For example, the .7; 3/-Josephus permutation is h3; 6; 2; 7; 5; 1; 4i.

a. Suppose that m is a constant. Describe an O.n/-time algorithm that, given an

integer n, outputs the .n; m/-Josephus permutation.

b. Suppose that m is not a constant. Describe an O.n lg n/-time algorithm that,

given integers n and m, outputs the .n; m/-Josephus permutation.

Chapter notes

In their book, Preparata and Shamos [282] describe several of the interval trees

that appear in the literature, citing work by H. Edelsbrunner (1980) and E. M.

McCreight (1981). The book details an interval tree that, given a static database

of n intervals, allows us to enumerate all k intervals that overlap a given query

interval in O.k C lg n/ time.

IV Advanced Design and Analysis Techniques

Introduction

This part covers three important techniques used in designing and analyzing efﬁ-

cient algorithms: dynamic programming (Chapter 15), greedy algorithms (Chap-

ter 16), and amortized analysis (Chapter 17). Earlier parts have presented other

widely applicable techniques, such as divide-and-conquer, randomization, and how

to solve recurrences. The techniques in this part are somewhat more sophisticated,

but they help us to attack many computational problems. The themes introduced in

this part will recur later in this book.

Dynamic programming typically applies to optimization problems in which we

make a set of choices in order to arrive at an optimal solution. As we make

each choice, subproblems of the same form often arise. Dynamic programming

is effective when a given subproblem may arise from more than one partial set of

choices; the key technique is to store the solution to each such subproblem in case it

should reappear. Chapter 15 shows how this simple idea can sometimes transform

exponential-time algorithms into polynomial-time algorithms.

Like dynamic-programming algorithms, greedy algorithms typically apply to

optimization problems in which we make a set of choices in order to arrive at an

optimal solution. The idea of a greedy algorithm is to make each choice in a locally

optimal manner. A simple example is coin-changing: to minimize the number of

U.S. coins needed to make change for a given amount, we can repeatedly select

the largest-denomination coin that is not larger than the amount that remains. A

greedy approach provides an optimal solution for many such problems much more

quickly than would a dynamic-programming approach. We cannot always easily

tell whether a greedy approach will be effective, however. Chapter 16 introduces

358 Part IV Advanced Design and Analysis Techniques

matroid theory, which provides a mathematical basis that can help us to show that

a greedy algorithm yields an optimal solution.

We use amortized analysis to analyze certain algorithms that perform a sequence

of similar operations. Instead of bounding the cost of the sequence of operations

by bounding the actual cost of each operation separately, an amortized analysis

provides a bound on the actual cost of the entire sequence. One advantage of this

approach is that although some operations might be expensive, many others might

be cheap. In other words, many of the operations might run in well under the worst-

case time. Amortized analysis is not just an analysis tool, however; it is also a way

of thinking about the design of algorithms, since the design of an algorithm and the

analysis of its running time are often closely intertwined. Chapter 17 introduces

three ways to perform an amortized analysis of an algorithm.

15 Dynamic Programming

Dynamic programming, like the divide-and-conquer method, solves problems by

combining the solutions to subproblems. (“Programming” in this context refers

to a tabular method, not to writing computer code.) As we saw in Chapters 2

and 4, divide-and-conquer algorithms partition the problem into disjoint subprob-

lems, solve the subproblems recursively, and then combine their solutions to solve

the original problem. In contrast, dynamic programming applies when the subprob-

lems overlap—that is, when subproblems share subsubproblems. In this context,

a divide-and-conquer algorithm does more work than necessary, repeatedly solv-

ing the common subsubproblems. A dynamic-programming algorithm solves each

subsubproblem just once and then saves its answer in a table, thereby avoiding the

work of recomputing the answer every time it solves each subsubproblem.

We typically apply dynamic programming to optimization problems. Such prob-

lems can have many possible solutions. Each solution has a value, and we wish to

ﬁnd a solution with the optimal (minimum or maximum) value. We call such a

solution an optimal solution to the problem, as opposed to the optimal solution,

since there may be several solutions that achieve the optimal value.

When developing a dynamic-programming algorithm, we follow a sequence of

four steps:

1. Characterize the structure of an optimal solution.

2. Recursively deﬁne the value of an optimal solution.

3. Compute the value of an optimal solution, typically in a bottom-up fashion.

4. Construct an optimal solution from computed information.

Steps 1–3 form the basis of a dynamic-programming solution to a problem. If we

need only the value of an optimal solution, and not the solution itself, then we

can omit step 4. When we do perform step 4, we sometimes maintain additional

information during step 3 so that we can easily construct an optimal solution.

The sections that follow use the dynamic-programming method to solve some

optimization problems. Section 15.1 examines the problem of cutting a rod into

360 Chapter 15 Dynamic Programming

rods of smaller length in way that maximizes their total value. Section 15.2 asks

how we can multiply a chain of matrices while performing the fewest total scalar

multiplications. Given these examples of dynamic programming, Section 15.3 dis-

cusses two key characteristics that a problem must have for dynamic programming

to be a viable solution technique. Section 15.4 then shows how to ﬁnd the longest

common subsequence of two sequences via dynamic programming. Finally, Sec-

tion 15.5 uses dynamic programming to construct binary search trees that are opti-

mal, given a known distribution of keys to be looked up.

15.1 Rod cutting

Our ﬁrst example uses dynamic programming to solve a simple problem in decid-

ing where to cut steel rods. Serling Enterprises buys long steel rods and cuts them

into shorter rods, which it then sells. Each cut is free. The management of Serling

Enterprises wants to know the best way to cut up the rods.

We assume that we know, for i D 1; 2; : : :, the price p i in dollars that Serling

Enterprises charges for a rod of length i inches. Rod lengths are always an integral

number of inches. Figure 15.1 gives a sample price table.

The rod-cutting problem is the following. Given a rod of length n inches and a

table of prices p i for i D 1; 2; : : : ; n, determine the maximum revenue r n obtain-

able by cutting up the rod and selling the pieces. Note that if the price p n for a rod

of length n is large enough, an optimal solution may require no cutting at all.

Consider the case when n D 4. Figure 15.2 shows all the ways to cut up a rod

of 4 inches in length, including the way with no cuts at all. We see that cutting a

4-inch rod into two 2-inch pieces produces revenue p 2 C p 2 D 5 C 5 D 10, which

is optimal.

We can cut up a rod of length n in 2 n1

different ways, since we have an in-

dependent option of cutting, or not cutting, at distance i inches from the left end,

length i 1 2 3 4 5 6 7 8 9 10

price pi 1 5 8 9 10 17 17 20 24 30

Figure 15.1 A sample price table for rods. Each rod of length i inches earns the company pi

dollars of revenue.

15.1 Rod cutting 361

9

(a)

1

(b)

8

(c) (d)

(e) (f) (g)

1

(h)

1 1 1

5 5 1 8

5 1 1 5 1 1 5 1 1

Figure 15.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the

value of that piece, according to the sample price chart of Figure 15.1. The optimal strategy is

part (c)—cutting the rod into two pieces of length 2—which has total value 10.

for i D 1; 2; : : : ; n 1. 1 We denote a decomposition into pieces using ordinary

additive notation, so that 7 D 2 C 2 C 3 indicates that a rod of length 7 is cut into

three pieces—two of length 2 and one of length 3. If an optimal solution cuts the

rod into k pieces, for some 1 k n, then an optimal decomposition

n D i 1 C i 2 C C i k

of the rod into pieces of lengths i 1 , i 2 , . . . , i k provides maximum corresponding

revenue

r n D p i 1 C p i 2 C C p i k :

For our sample problem, we can determine the optimal revenue ﬁgures r i , for

i D 1; 2; : : : ; 10, by inspection, with the corresponding optimal decompositions

1

If we required the pieces to be cut in order of nondecreasing size, there would be fewer ways

to consider. For n D 4, we would consider only 5 such ways: parts (a), (b), (c), (e), and (h)

in Figure 15.2. The number of ways is called the partition function; it is approximately equal to

e

p

2n=3

=4n

p

3. This quantity is less than 2

n1

, but still much greater than any polynomial in n.

We shall not pursue this line of inquiry further, however.

362 Chapter 15 Dynamic Programming

r 1 D 1 from solution 1 D 1 (no cuts) ;

r 2 D 5 from solution 2 D 2 (no cuts) ;

r 3 D 8 from solution 3 D 3 (no cuts) ;

r 4 D 10 from solution 4 D 2 C 2 ;

r 5 D 13 from solution 5 D 2 C 3 ;

r 6 D 17 from solution 6 D 6 (no cuts) ;

r 7 D 18 from solution 7 D 1 C 6 or 7 D 2 C 2 C 3 ;

r 8 D 22 from solution 8 D 2 C 6 ;

r 9 D 25 from solution 9 D 3 C 6 ;

r 10 D 30 from solution 10 D 10 (no cuts) :

More generally, we can frame the values r n for n 1 in terms of optimal rev-

enues from shorter rods:

r n D max .p n ; r 1 C r n1 ; r 2 C r n2 ; : : : ; r n1 C r 1 / : (15.1)

The ﬁrst argument, p n , corresponds to making no cuts at all and selling the rod of

length n as is. The other n 1 arguments to max correspond to the maximum rev-

enue obtained by making an initial cut of the rod into two pieces of size i and n i,

for each i D 1; 2; : : : ; n 1, and then optimally cutting up those pieces further,

obtaining revenues r i and r ni from those two pieces. Since we don’t know ahead

of time which value of i optimizes revenue, we have to consider all possible values

for i and pick the one that maximizes revenue. We also have the option of picking

no i at all if we can obtain more revenue by selling the rod uncut.

Note that to solve the original problem of size n, we solve smaller problems of

the same type, but of smaller sizes. Once we make the ﬁrst cut, we may consider

the two pieces as independent instances of the rod-cutting problem. The overall

optimal solution incorporates optimal solutions to the two related subproblems,

maximizing revenue from each of those two pieces. We say that the rod-cutting

problem exhibits optimal substructure: optimal solutions to a problem incorporate

optimal solutions to related subproblems, which we may solve independently.

In a related, but slightly simpler, way to arrange a recursive structure for the rod-

cutting problem, we view a decomposition as consisting of a ﬁrst piece of length i

cut off the left-hand end, and then a right-hand remainder of length n i. Only

the remainder, and not the ﬁrst piece, may be further divided. We may view every

decomposition of a length-n rod in this way: as a ﬁrst piece followed by some

decomposition of the remainder. When doing so, we can couch the solution with

no cuts at all as saying that the ﬁrst piece has size i D n and revenue p n and that

the remainder has size 0 with corresponding revenue r 0 D 0. We thus obtain the

following simpler version of equation (15.1):

r n D max

1in

.p i C r ni / : (15.2)

15.1 Rod cutting 363

In this formulation, an optimal solution embodies the solution to only one related

subproblem—the remainder—rather than two.

Recursive top-down implementation

The following procedure implements the computation implicit in equation (15.2)

in a straightforward, top-down, recursive manner.

CUT-ROD.p; n/

1 if n == 0

2 return 0

3 q D 1

4 for i D 1 to n

5 q D max.q; pŒi C CUT-ROD.p; n i//

6 return q

Procedure CUT-ROD takes as input an array pŒ1 : : n of prices and an integer n,

and it returns the maximum revenue possible for a rod of length n. If n D 0, no

revenue is possible, and so CUT-ROD returns 0 in line 2. Line 3 initializes the

maximum revenue q to 1, so that the for loop in lines 4–5 correctly computes

q D max 1in .p i C CUT-ROD.p; n i//; line 6 then returns this value. A simple

induction on n proves that this answer is equal to the desired answer r n , using

equation (15.2).

If you were to code up CUT-ROD in your favorite programming language and run

it on your computer, you would ﬁnd that once the input size becomes moderately

large, your program would take a long time to run. For n D 40, you would ﬁnd that

your program takes at least several minutes, and most likely more than an hour. In

fact, you would ﬁnd that each time you increase n by 1, your program’s running

time would approximately double.

Why is CUT-ROD so inefﬁcient? The problem is that CUT-ROD calls itself

recursively over and over again with the same parameter values; it solves the

same subproblems repeatedly. Figure 15.3 illustrates what happens for n D 4:

CUT-ROD.p; n/ calls CUT-ROD.p; n i/ for i D 1; 2; : : : ; n. Equivalently,

CUT-ROD.p; n/ calls CUT-ROD.p; j / for each j D 0; 1; : : : ; n 1. When this

process unfolds recursively, the amount of work done, as a function of n, grows

explosively.

To analyze the running time of CUT-ROD, let T .n/ denote the total number of

calls made to CUT-ROD when called with its second parameter equal to n. This

expression equals the number of nodes in a subtree whose root is labeled n in the

recursion tree. The count includes the initial call at its root. Thus, T .0/ D 1 and

364 Chapter 15 Dynamic Programming

3

1 0

0

0

0 1

2 0

0

1

2

0

1 0

4

Figure 15.3 The recursion tree showing recursive calls resulting from a call CUT-ROD.p; n/ for

n D 4. Each node label gives the size n of the corresponding subproblem, so that an edge from

a parent with label s to a child with label t corresponds to cutting off an initial piece of size s t

and leaving a remaining subproblem of size t. A path from the root to a leaf corresponds to one of

the 2

n1

ways of cutting up a rod of length n. In general, this recursion tree has 2

n

nodes and 2

n1

leaves.

T .n/ D 1 C

n1 X

j D0

T .j / : (15.3)

The initial 1 is for the call at the root, and the term T .j / counts the number of calls

(including recursive calls) due to the call CUT-ROD.p; n i/, where j D n i.

As Exercise 15.1-1 asks you to show,

T .n/ D 2

n

; (15.4)

and so the running time of CUT-ROD is exponential in n.

In retrospect, this exponential running time is not so surprising. CUT-ROD ex-

plicitly considers all the 2 n1

possible ways of cutting up a rod of length n. The

tree of recursive calls has 2 n1

leaves, one for each possible way of cutting up the

rod. The labels on the simple path from the root to a leaf give the sizes of each

remaining right-hand piece before making each cut. That is, the labels give the

corresponding cut points, measured from the right-hand end of the rod.

Using dynamic programming for optimal rod cutting

We now show how to convert CUT-ROD into an efﬁcient algorithm, using dynamic

programming.

The dynamic-programming method works as follows. Having observed that a

naive recursive solution is inefﬁcient because it solves the same subproblems re-

peatedly, we arrange for each subproblem to be solved only once, saving its solu-

tion. If we need to refer to this subproblem’s solution again later, we can just look it

15.1 Rod cutting 365

up, rather than recompute it. Dynamic programming thus uses additional memory

to save computation time; it serves an example of a time-memory trade-off. The

savings may be dramatic: an exponential-time solution may be transformed into a

polynomial-time solution. A dynamic-programming approach runs in polynomial

time when the number of distinct subproblems involved is polynomial in the input

size and we can solve each such subproblem in polynomial time.

There are usually two equivalent ways to implement a dynamic-programming

approach. We shall illustrate both of them with our rod-cutting example.

The ﬁrst approach is top-down with memoization. 2 In this approach, we write

the procedure recursively in a natural manner, but modiﬁed to save the result of

each subproblem (usually in an array or hash table). The procedure now ﬁrst checks

to see whether it has previously solved this subproblem. If so, it returns the saved

value, saving further computation at this level; if not, the procedure computes the

value in the usual manner. We say that the recursive procedure has been memoized;

it “remembers” what results it has computed previously.

The second approach is the bottom-up method. This approach typically depends

on some natural notion of the “size” of a subproblem, such that solving any par-

ticular subproblem depends only on solving “smaller” subproblems. We sort the

subproblems by size and solve them in size order, smallest ﬁrst. When solving a

particular subproblem, we have already solved all of the smaller subproblems its

solution depends upon, and we have saved their solutions. We solve each sub-

problem only once, and when we ﬁrst see it, we have already solved all of its

prerequisite subproblems.

These two approaches yield algorithms with the same asymptotic running time,

except in unusual circumstances where the top-down approach does not actually

recurse to examine all possible subproblems. The bottom-up approach often has

much better constant factors, since it has less overhead for procedure calls.

Here is the the pseudocode for the top-down CUT-ROD procedure, with memo-

ization added:

MEMOIZED-CUT-ROD.p; n/

1 let rŒ0 : : n be a new array

2 for i D 0 to n

3 rŒi D 1

4 return MEMOIZED-CUT-ROD-AUX.p; n; r/

2

This is not a misspelling. The word really is memoization, not memorization. Memoization comes

from memo, since the technique consists of recording a value so that we can look it up later.

366 Chapter 15 Dynamic Programming

MEMOIZED-CUT-ROD-AUX.p; n; r/

1 if rŒn 0

2 return rŒn

3 if n == 0

4 q D 0

5 else q D 1

6 for i D 1 to n

7 q D max.q; pŒi C MEMOIZED-CUT-ROD-AUX.p; n i; r//

8 rŒn D q

9 return q

Here, the main procedure MEMOIZED-CUT-ROD initializes a new auxiliary ar-

ray rŒ0 : : n with the value 1, a convenient choice with which to denote “un-

known.” (Known revenue values are always nonnegative.) It then calls its helper

routine, MEMOIZED-CUT-ROD-AUX.

The procedure MEMOIZED-CUT-ROD-AUX is just the memoized version of our

previous procedure, CUT-ROD. It ﬁrst checks in line 1 to see whether the desired

value is already known and, if it is, then line 2 returns it. Otherwise, lines 3–7

compute the desired value q in the usual manner, line 8 saves it in rŒn, and line 9

returns it.

The bottom-up version is even simpler:

BOTTOM-UP-CUT-ROD.p; n/

1 let rŒ0 : : n be a new array

2 rŒ0 D 0

3 for j D 1 to n

4 q D 1

5 for i D 1 to j

6 q D max.q; pŒi C rŒj i/

7 rŒj D q

8 return rŒn

For the bottom-up dynamic-programming approach, BOTTOM-UP-CUT-ROD

uses the natural ordering of the subproblems: a problem of size i is “smaller”

than a subproblem of size j if i < j . Thus, the procedure solves subproblems of

sizes j D 0; 1; : : : ; n, in that order.

Line 1 of procedure BOTTOM-UP-CUT-ROD creates a new array rŒ0 : : n in

which to save the results of the subproblems, and line 2 initializes rŒ0 to 0, since

a rod of length 0 earns no revenue. Lines 3–6 solve each subproblem of size j , for

j D 1; 2; : : : ; n, in order of increasing size. The approach used to solve a problem

of a particular size j is the same as that used by CUT-ROD, except that line 6 now

15.1 Rod cutting 367

3

0

1

2

4

Figure 15.4 The subproblem graph for the rod-cutting problem with n D 4. The vertex labels

give the sizes of the corresponding subproblems. A directed edge .x; y/ indicates that we need a

solution to subproblem y when solving subproblem x. This graph is a reduced version of the tree of

Figure 15.3, in which all nodes with the same label are collapsed into a single vertex and all edges

go from parent to child.

directly references array entry rŒj i instead of making a recursive call to solve

the subproblem of size j i. Line 7 saves in rŒj the solution to the subproblem

of size j . Finally, line 8 returns rŒn, which equals the optimal value r n .

The bottom-up and top-down versions have the same asymptotic running time.

The running time of procedure BOTTOM-UP-CUT-ROD is ‚.n 2 /, due to its

doubly-nested loop structure. The number of iterations of its inner for loop, in

lines 5–6, forms an arithmetic series. The running time of its top-down counterpart,

MEMOIZED-CUT-ROD, is also ‚.n 2 /, although this running time may be a little

harder to see. Because a recursive call to solve a previously solved subproblem

returns immediately, MEMOIZED-CUT-ROD solves each subproblem just once. It

solves subproblems for sizes 0; 1; : : : ; n. To solve a subproblem of size n, the for

loop of lines 6–7 iterates n times. Thus, the total number of iterations of this for

loop, over all recursive calls of MEMOIZED-CUT-ROD, forms an arithmetic series,

giving a total of ‚.n 2 / iterations, just like the inner for loop of BOTTOM-UP-

CUT-ROD. (We actually are using a form of aggregate analysis here. We shall see

aggregate analysis in detail in Section 17.1.)

Subproblem graphs

When we think about a dynamic-programming problem, we should understand the

set of subproblems involved and how subproblems depend on one another.

The subproblem graph for the problem embodies exactly this information. Fig-

ure 15.4 shows the subproblem graph for the rod-cutting problem with n D 4. It

is a directed graph, containing one vertex for each distinct subproblem. The sub-

368 Chapter 15 Dynamic Programming

problem graph has a directed edge from the vertex for subproblem x to the vertex

for subproblem y if determining an optimal solution for subproblem x involves

directly considering an optimal solution for subproblem y. For example, the sub-

problem graph contains an edge from x to y if a top-down recursive procedure for

solving x directly calls itself to solve y. We can think of the subproblem graph

as a “reduced” or “collapsed” version of the recursion tree for the top-down recur-

sive method, in which we coalesce all nodes for the same subproblem into a single

vertex and direct all edges from parent to child.

The bottom-up method for dynamic programming considers the vertices of the

subproblem graph in such an order that we solve the subproblems y adjacent to

a given subproblem x before we solve subproblem x. (Recall from Section B.4

that the adjacency relation is not necessarily symmetric.) Using the terminology

from Chapter 22, in a bottom-up dynamic-programming algorithm, we consider the

vertices of the subproblem graph in an order that is a “reverse topological sort,” or

a “topological sort of the transpose” (see Section 22.4) of the subproblem graph. In

other words, no subproblem is considered until all of the subproblems it depends

upon have been solved. Similarly, using notions from the same chapter, we can

view the top-down method (with memoization) for dynamic programming as a

“depth-ﬁrst search” of the subproblem graph (see Section 22.3).

The size of the subproblem graph G D .V; E/ can help us determine the running

time of the dynamic programming algorithm. Since we solve each subproblem just

once, the running time is the sum of the times needed to solve each subproblem.

Typically, the time to compute the solution to a subproblem is proportional to the

degree (number of outgoing edges) of the corresponding vertex in the subproblem

graph, and the number of subproblems is equal to the number of vertices in the sub-

problem graph. In this common case, the running time of dynamic programming

is linear in the number of vertices and edges.

Reconstructing a solution

Our dynamic-programming solutions to the rod-cutting problem return the value of

an optimal solution, but they do not return an actual solution: a list of piece sizes.

We can extend the dynamic-programming approach to record not only the optimal

value computed for each subproblem, but also a choice that led to the optimal

value. With this information, we can readily print an optimal solution.

Here is an extended version of BOTTOM-UP-CUT-ROD that computes, for each

rod size j , not only the maximum revenue r j , but also s j , the optimal size of the

ﬁrst piece to cut off:

15.1 Rod cutting 369

EXTENDED-BOTTOM-UP-CUT-ROD.p; n/

1 let rŒ0 : : n and sŒ0 : : n be new arrays

2 rŒ0 D 0

3 for j D 1 to n

4 q D 1

5 for i D 1 to j

6 if q < pŒi C rŒj i

7 q D pŒi C rŒj i

8 sŒj D i

9 rŒj D q

10 return r and s

This procedure is similar to BOTTOM-UP-CUT-ROD, except that it creates the ar-

ray s in line 1, and it updates sŒj in line 8 to hold the optimal size i of the ﬁrst

piece to cut off when solving a subproblem of size j .

The following procedure takes a price table p and a rod size n, and it calls

EXTENDED-BOTTOM-UP-CUT-ROD to compute the array sŒ1 : : n of optimal

ﬁrst-piece sizes and then prints out the complete list of piece sizes in an optimal

decomposition of a rod of length n:

PRINT-CUT-ROD-SOLUTION.p; n/

1 .r; s/ D EXTENDED-BOTTOM-UP-CUT-ROD.p; n/

2 while n > 0

3 print sŒn

4 n D n sŒn

In our rod-cutting example, the call EXTENDED-BOTTOM-UP-CUT-ROD.p; 10/

would return the following arrays:

i 0 1 2 3 4 5 6 7 8 9 10

rŒi 0 1 5 8 10 13 17 18 22 25 30

sŒi 0 1 2 3 2 2 6 1 2 3 10

A call to PRINT-CUT-ROD-SOLUTION.p; 10/ would print just 10, but a call with

n D 7 would print the cuts 1 and 6, corresponding to the ﬁrst optimal decomposi-

tion for r 7 given earlier.

Exercises

15.1-1

Show that equation (15.4) follows from equation (15.3) and the initial condition

T .0/ D 1.

370 Chapter 15 Dynamic Programming

15.1-2

Show, by means of a counterexample, that the following “greedy” strategy does

not always determine an optimal way to cut rods. Deﬁne the density of a rod of

length i to be p i =i, that is, its value per inch. The greedy strategy for a rod of

length n cuts off a ﬁrst piece of length i, where 1 i n, having maximum

density. It then continues by applying the greedy strategy to the remaining piece of

length n i.

15.1-3

Consider a modiﬁcation of the rod-cutting problem in which, in addition to a

price p i for each rod, each cut incurs a ﬁxed cost of c. The revenue associated with

a solution is now the sum of the prices of the pieces minus the costs of making the

cuts. Give a dynamic-programming algorithm to solve this modiﬁed problem.

15.1-4

Modify MEMOIZED-CUT-ROD to return not only the value but the actual solution,

too.

15.1-5

The Fibonacci numbers are deﬁned by recurrence (3.22). Give an O.n/-time

dynamic-programming algorithm to compute the nth Fibonacci number. Draw the

subproblem graph. How many vertices and edges are in the graph?

15.2 Matrix-chain multiplication

Our next example of dynamic programming is an algorithm that solves the problem

of matrix-chain multiplication. We are given a sequence (chain) hA 1 ; A 2 ; : : : ; A n i

of n matrices to be multiplied, and we wish to compute the product

A 1 A 2 A n : (15.5)

We can evaluate the expression (15.5) using the standard algorithm for multiply-

ing pairs of matrices as a subroutine once we have parenthesized it to resolve all

ambiguities in how the matrices are multiplied together. Matrix multiplication is

associative, and so all parenthesizations yield the same product. A product of ma-

trices is fully parenthesized if it is either a single matrix or the product of two fully

parenthesized matrix products, surrounded by parentheses. For example, if the

chain of matrices is hA 1 ; A 2 ; A 3 ; A 4 i, then we can fully parenthesize the product

A 1 A 2 A 3 A 4 in ﬁve distinct ways:

15.2 Matrix-chain multiplication 371

.A 1 .A 2 .A 3 A 4 /// ;

.A 1 ..A 2 A 3 /A 4 // ;

..A 1 A 2 /.A 3 A 4 // ;

..A 1 .A 2 A 3 //A 4 / ;

...A 1 A 2 /A 3 /A 4 / :

How we parenthesize a chain of matrices can have a dramatic impact on the cost

of evaluating the product. Consider ﬁrst the cost of multiplying two matrices. The

standard algorithm is given by the following pseudocode, which generalizes the

SQUARE-MATRIX-MULTIPLY procedure from Section 4.2. The attributes rows

and columns are the numbers of rows and columns in a matrix.

MATRIX-MULTIPLY.A; B/

1 if A:columns ¤ B:rows

2 error “incompatible dimensions”

3 else let C be a new A:rows B:columns matrix

4 for i D 1 to A:rows

5 for j D 1 to B:columns

6 c ij D 0

7 for k D 1 to A:columns

8 c ij D c ij C a ik b kj

9 return C

We can multiply two matrices A and B only if they are compatible: the number of

columns of A must equal the number of rows of B. If A is a p q matrix and B is

a q r matrix, the resulting matrix C is a p r matrix. The time to compute C is

dominated by the number of scalar multiplications in line 8, which is pqr. In what

follows, we shall express costs in terms of the number of scalar multiplications.

To illustrate the different costs incurred by different parenthesizations of a matrix

product, consider the problem of a chain hA 1 ; A 2 ; A 3 i of three matrices. Suppose

that the dimensions of the matrices are 10 100, 100 5, and 5 50, respec-

tively. If we multiply according to the parenthesization ..A 1 A 2 /A 3 /, we perform

10 100 5 D 5000 scalar multiplications to compute the 10 5 matrix prod-

uct A 1 A 2 , plus another 10 5 50 D 2500 scalar multiplications to multiply this

matrix by A 3 , for a total of 7500 scalar multiplications. If instead we multiply

according to the parenthesization .A 1 .A 2 A 3 //, we perform 100 5 50 D 25,000

scalar multiplications to compute the 100 50 matrix product A 2 A 3 , plus another

10 100 50 D 50,000 scalar multiplications to multiply A 1 by this matrix, for a

total of 75,000 scalar multiplications. Thus, computing the product according to

the ﬁrst parenthesization is 10 times faster.

We state the matrix-chain multiplication problem as follows: given a chain

hA 1 ;A 2 ;: : : ;A n i of n matrices, where for i D 1; 2; : : : ; n, matrix A i has dimension

372 Chapter 15 Dynamic Programming

p i1 p i , fully parenthesize the product A 1 A 2 A n in a way that minimizes the

number of scalar multiplications.

Note that in the matrix-chain multiplication problem, we are not actually multi-

plying matrices. Our goal is only to determine an order for multiplying matrices

that has the lowest cost. Typically, the time invested in determining this optimal

order is more than paid for by the time saved later on when actually performing the

matrix multiplications (such as performing only 7500 scalar multiplications instead

of 75,000).

Counting the number of parenthesizations

Before solving the matrix-chain multiplication problem by dynamic programming,

let us convince ourselves that exhaustively checking all possible parenthesizations

does not yield an efﬁcient algorithm. Denote the number of alternative parenthe-

sizations of a sequence of n matrices by P.n/. When n D 1, we have just one

matrix and therefore only one way to fully parenthesize the matrix product. When

n 2, a fully parenthesized matrix product is the product of two fully parenthe-

sized matrix subproducts, and the split between the two subproducts may occur

between the kth and .k C 1/st matrices for any k D 1; 2; : : : ; n 1. Thus, we

obtain the recurrence

P.n/ D

1 if n D 1 ;

n1 X

kD1

P.k/P.n k/ if n 2 :

(15.6)

Problem 12-4 asked you to show that the solution to a similar recurrence is the

sequence of Catalan numbers, which grows as .4 n =n 3=2 /. A simpler exercise

(see Exercise 15.2-3) is to show that the solution to the recurrence (15.6) is .2 n /.

The number of solutions is thus exponential in n, and the brute-force method of

exhaustive search makes for a poor strategy when determining how to optimally

parenthesize a matrix chain.

Applying dynamic programming

We shall use the dynamic-programming method to determine how to optimally

parenthesize a matrix chain. In so doing, we shall follow the four-step sequence

that we stated at the beginning of this chapter:

1. Characterize the structure of an optimal solution.

2. Recursively deﬁne the value of an optimal solution.

3. Compute the value of an optimal solution.

15.2 Matrix-chain multiplication 373

4. Construct an optimal solution from computed information.

We shall go through these steps in order, demonstrating clearly how we apply each

step to the problem.

Step 1: The structure of an optimal parenthesization

For our ﬁrst step in the dynamic-programming paradigm, we ﬁnd the optimal sub-

structure and then use it to construct an optimal solution to the problem from opti-

mal solutions to subproblems. In the matrix-chain multiplication problem, we can

perform this step as follows. For convenience, let us adopt the notation A i::j , where

i j , for the matrix that results from evaluating the product A i A iC1 A j . Ob-

serve that if the problem is nontrivial, i.e., i < j , then to parenthesize the product

A i A iC1 A j , we must split the product between A k and A kC1 for some integer k

in the range i k < j . That is, for some value of k, we ﬁrst compute the matrices

A i::k and A kC1::j and then multiply them together to produce the ﬁnal product A i::j .

The cost of parenthesizing this way is the cost of computing the matrix A i::k , plus

the cost of computing A kC1::j , plus the cost of multiplying them together.

The optimal substructure of this problem is as follows. Suppose that to op-

timally parenthesize A i A iC1 A j , we split the product between A k and A kC1 .

Then the way we parenthesize the “preﬁx” subchain A i A iC1 A k within this

optimal parenthesization of A i A iC1 A j must be an optimal parenthesization of

A i A iC1 A k . Why? If there were a less costly way to parenthesize A i A iC1 A k ,

then we could substitute that parenthesization in the optimal parenthesization

of A i A iC1 A j to produce another way to parenthesize A i A iC1 A j whose cost

was lower than the optimum: a contradiction. A similar observation holds for how

we parenthesize the subchain A kC1 A kC2 A j in the optimal parenthesization of

A i A iC1 A j : it must be an optimal parenthesization of A kC1 A kC2 A j .

Now we use our optimal substructure to show that we can construct an optimal

solution to the problem from optimal solutions to subproblems. We have seen that

any solution to a nontrivial instance of the matrix-chain multiplication problem

requires us to split the product, and that any optimal solution contains within it op-

timal solutions to subproblem instances. Thus, we can build an optimal solution to

an instance of the matrix-chain multiplication problem by splitting the problem into

two subproblems (optimally parenthesizing A i A iC1 A k and A kC1 A kC2 A j ),

ﬁnding optimal solutions to subproblem instances, and then combining these op-

timal subproblem solutions. We must ensure that when we search for the correct

place to split the product, we have considered all possible places, so that we are

sure of having examined the optimal one.

374 Chapter 15 Dynamic Programming

Step 2: A recursive solution

Next, we deﬁne the cost of an optimal solution recursively in terms of the optimal

solutions to subproblems. For the matrix-chain multiplication problem, we pick as

our subproblems the problems of determining the minimum cost of parenthesizing

A i A iC1 A j for 1 i j n. Let mŒi; j be the minimum number of scalar

multiplications needed to compute the matrix A i::j ; for the full problem, the lowest-

cost way to compute A 1::n would thus be mŒ1; n.

We can deﬁne mŒi; j recursively as follows. If i D j , the problem is trivial;

the chain consists of just one matrix A i::i D A i , so that no scalar multiplications

are necessary to compute the product. Thus, mŒi; i D 0 for i D 1; 2; : : : ; n. To

compute mŒi; j when i < j , we take advantage of the structure of an optimal

solution from step 1. Let us assume that to optimally parenthesize, we split the

product A i A iC1 A j between A k and A kC1 , where i k < j . Then, mŒi; j

equals the minimum cost for computing the subproducts A i::k and A kC1::j , plus the

cost of multiplying these two matrices together. Recalling that each matrix A i is

p i1 p i , we see that computing the matrix product A i::k A kC1::j takes p i1 p k p j

scalar multiplications. Thus, we obtain

mŒi; j D mŒi; k C mŒk C 1; j C p i1 p k p j :

This recursive equation assumes that we know the value of k, which we do not.

There are only j i possible values for k, however, namely k D i; i C1; : : : ; j 1.

Since the optimal parenthesization must use one of these values for k, we need only

check them all to ﬁnd the best. Thus, our recursive deﬁnition for the minimum cost

of parenthesizing the product A i A iC1 A j becomes

mŒi; j D

(

0 if i D j ;

min

ik<j

fmŒi; k C mŒk C 1; j C p i1 p k p j g if i < j :

(15.7)

The mŒi; j values give the costs of optimal solutions to subproblems, but they

do not provide all the information we need to construct an optimal solution. To

help us do so, we deﬁne sŒi; j to be a value of k at which we split the product

A i A iC1 A j in an optimal parenthesization. That is, sŒi; j equals a value k such

that mŒi; j D mŒi; k C mŒk C 1; j C p i1 p k p j .

Step 3: Computing the optimal costs

At this point, we could easily write a recursive algorithm based on recurrence (15.7)

to compute the minimum cost mŒ1; n for multiplying A 1 A 2 A n . As we saw for

the rod-cutting problem, and as we shall see in Section 15.3, this recursive algo-

rithm takes exponential time, which is no better than the brute-force method of

checking each way of parenthesizing the product.

15.2 Matrix-chain multiplication 375

Observe that we have relatively few distinct subproblems: one subproblem for

each choice of i and j satisfying 1 i j n, or

n

2

C n D ‚.n 2 / in all.

A recursive algorithm may encounter each subproblem many times in different

branches of its recursion tree. This property of overlapping subproblems is the

second hallmark of when dynamic programming applies (the ﬁrst hallmark being

optimal substructure).

Instead of computing the solution to recurrence (15.7) recursively, we compute

the optimal cost by using a tabular, bottom-up approach. (We present the corre-

sponding top-down approach using memoization in Section 15.3.)

We shall implement the tabular, bottom-up method in the procedure MATRIX-

CHAIN-ORDER, which appears below. This procedure assumes that matrix A i

has dimensions p i1 p i for i D 1; 2; : : : ; n. Its input is a sequence p D

hp 0 ; p 1 ; : : : ; p n i, where p:length D n C 1. The procedure uses an auxiliary

table mŒ1 : : n; 1 : : n for storing the mŒi; j costs and another auxiliary table

sŒ1 : : n 1; 2 : : n that records which index of k achieved the optimal cost in com-

puting mŒi; j . We shall use the table s to construct an optimal solution.

In order to implement the bottom-up approach, we must determine which entries

of the table we refer to when computing mŒi; j . Equation (15.7) shows that the

cost mŒi; j of computing a matrix-chain product of j i C1 matrices depends only

on the costs of computing matrix-chain products of fewer than j i C 1 matrices.

That is, for k D i; i C 1; : : : ; j 1, the matrix A i::k is a product of k i C 1 <

j i C 1 matrices and the matrix A kC1::j is a product of j k < j i C 1

matrices. Thus, the algorithm should ﬁll in the table m in a manner that corresponds

to solving the parenthesization problem on matrix chains of increasing length. For

the subproblem of optimally parenthesizing the chain A i A iC1 A j , we consider

the subproblem size to be the length j i C 1 of the chain.

MATRIX-CHAIN-ORDER.p/

1 n D p:length 1

2 let mŒ1 : : n; 1 : : n and sŒ1 : : n 1; 2 : : n be new tables

3 for i D 1 to n

4 mŒi; i D 0

5 for l D 2 to n // l is the chain length

6 for i D 1 to n l C 1

7 j D i C l 1

8 mŒi; j D 1

9 for k D i to j 1

10 q D mŒi; k C mŒk C 1; j C p i1 p k p j

11 if q < mŒi; j

12 mŒi; j D q

13 sŒi; j D k

14 return m and s

376 Chapter 15 Dynamic Programming

A

6

A

5

A

4

A

3

A

2

A

1

0 0 0 0 0 0

15,750 2,625 750 1,000 5,000

7,875 4,375 2,500 3,500

9,375 7,125 5,375

11,875 10,500

15,125

1

2

3

4

5

6 1

2

3

4

5

6

j i

m

1 2 3 4 5

1 3 3 5

3 3 3

3 3

3

2

3

4

5

6 1

2

3

4

5

j i

s

Figure 15.5 The m and s tables computed by MATRIX-CHAIN-ORDER for n D 6 and the follow-

ing matrix dimensions:

matrix A1 A2 A3 A4 A5 A6

dimension 30 35 35 15 15 5 5 10 10 20 20 25

The tables are rotated so that the main diagonal runs horizontally. The m table uses only the main

diagonal and upper triangle, and the s table uses only the upper triangle. The minimum number of

scalar multiplications to multiply the 6 matrices is mŒ1; 6 D 15,125. Of the darker entries, the pairs

that have the same shading are taken together in line 10 when computing

mŒ2; 5 D min

8

ˆ<

ˆ:

mŒ2; 2 C mŒ3; 5 C p1p2p5 D 0 C 2500 C 35 15 20 D 13,000 ;

mŒ2; 3 C mŒ4; 5 C p1p3p5 D 2625 C 1000 C 35 5 20 D 7125 ;

mŒ2; 4 C mŒ5; 5 C p1p4p5 D 4375 C 0 C 35 10 20 D 11,375

D 7125 :

The algorithm ﬁrst computes mŒi; i D 0 for i D 1; 2; : : : ; n (the minimum

costs for chains of length 1) in lines 3–4. It then uses recurrence (15.7) to compute

mŒi; i C 1 for i D 1; 2; : : : ; n 1 (the minimum costs for chains of length l D 2)

during the ﬁrst execution of the for loop in lines 5–13. The second time through the

loop, it computes mŒi; iC2 for i D 1; 2; : : : ; n2 (the minimum costs for chains of

length l D 3), and so forth. At each step, the mŒi; j cost computed in lines 10–13

depends only on table entries mŒi; k and mŒk C 1; j already computed.

Figure 15.5 illustrates this procedure on a chain of n D 6 matrices. Since

we have deﬁned mŒi; j only for i j , only the portion of the table m strictly

above the main diagonal is used. The ﬁgure shows the table rotated to make the

main diagonal run horizontally. The matrix chain is listed along the bottom. Us-

ing this layout, we can ﬁnd the minimum cost mŒi; j for multiplying a subchain

A i A iC1 A j of matrices at the intersection of lines running northeast from A i and

15.2 Matrix-chain multiplication 377

northwest from A j . Each horizontal row in the table contains the entries for matrix

chains of the same length. MATRIX-CHAIN-ORDER computes the rows from bot-

tom to top and from left to right within each row. It computes each entry mŒi; j

using the products p i1 p k p j for k D i; i C 1; : : : ; j 1 and all entries southwest

and southeast from mŒi; j .

A simple inspection of the nested loop structure of MATRIX-CHAIN-ORDER

yields a running time of O.n 3 / for the algorithm. The loops are nested three deep,

and each loop index (l, i, and k) takes on at most n1 values. Exercise 15.2-5 asks

you to show that the running time of this algorithm is in fact also .n 3 /. The al-

gorithm requires ‚.n 2 / space to store the m and s tables. Thus, MATRIX-CHAIN-

ORDER is much more efﬁcient than the exponential-time method of enumerating

all possible parenthesizations and checking each one.

Step 4: Constructing an optimal solution

Although MATRIX-CHAIN-ORDER determines the optimal number of scalar mul-

tiplications needed to compute a matrix-chain product, it does not directly show

how to multiply the matrices. The table sŒ1 : : n 1; 2 : : n gives us the informa-

tion we need to do so. Each entry sŒi; j records a value of k such that an op-

timal parenthesization of A i A iC1 A j splits the product between A k and A kC1 .

Thus, we know that the ﬁnal matrix multiplication in computing A 1::n optimally

is A 1::sŒ1;n A sŒ1;nC1::n . We can determine the earlier matrix multiplications recur-

sively, since sŒ1; sŒ1; n determines the last matrix multiplication when computing

A 1::sŒ1;n and sŒsŒ1; n C 1; n determines the last matrix multiplication when com-

puting A sŒ1;nC1::n . The following recursive procedure prints an optimal parenthe-

sization of hA i ; A iC1 ; : : : ; A j i, given the s table computed by MATRIX-CHAIN-

ORDER and the indices i and j . The initial call PRINT-OPTIMAL-PARENS.s; 1; n/

prints an optimal parenthesization of hA 1 ; A 2 ; : : : ; A n i.

PRINT-OPTIMAL-PARENS.s; i; j /

1 if i == j

2 print “A” i

3 else print “(”

4 PRINT-OPTIMAL-PARENS.s; i; sŒi; j /

5 PRINT-OPTIMAL-PARENS.s; sŒi; j C 1; j /

6 print “)”

In the example of Figure 15.5, the call PRINT-OPTIMAL-PARENS.s; 1; 6/ prints

the parenthesization ..A 1 .A 2 A 3 //..A 4 A 5 /A 6 //.

378 Chapter 15 Dynamic Programming

Exercises

15.2-1

Find an optimal parenthesization of a matrix-chain product whose sequence of

dimensions is h5; 10; 3; 12; 5; 50; 6i.

15.2-2

Give a recursive algorithm MATRIX-CHAIN-MULTIPLY.A; s; i; j / that actually

performs the optimal matrix-chain multiplication, given the sequence of matrices

hA 1 ; A 2 ; : : : ; A n i, the s table computed by MATRIX-CHAIN-ORDER, and the in-

dices i and j . (The initial call would be MATRIX-CHAIN-MULTIPLY.A; s; 1; n/.)

15.2-3

Use the substitution method to show that the solution to the recurrence (15.6)

is .2 n /.

15.2-4

Describe the subproblem graph for matrix-chain multiplication with an input chain

of length n. How many vertices does it have? How many edges does it have, and

which edges are they?

15.2-5

Let R.i; j / be the number of times that table entry mŒi; j is referenced while

computing other table entries in a call of MATRIX-CHAIN-ORDER. Show that the

total number of references for the entire table is

n X

iD1

n X

j Di

R.i; j / D

n 3 n

3

:

(Hint: You may ﬁnd equation (A.3) useful.)

15.2-6

Show that a full parenthesization of an n-element expression has exactly n1 pairs

of parentheses.

15.3 Elements of dynamic programming

Although we have just worked through two examples of the dynamic-programming

method, you might still be wondering just when the method applies. From an en-

gineering perspective, when should we look for a dynamic-programming solution

to a problem? In this section, we examine the two key ingredients that an opti-

15.3 Elements of dynamic programming 379

mization problem must have in order for dynamic programming to apply: optimal

substructure and overlapping subproblems. We also revisit and discuss more fully

how memoization might help us take advantage of the overlapping-subproblems

property in a top-down recursive approach.

Optimal substructure

The ﬁrst step in solving an optimization problem by dynamic programming is to

characterize the structure of an optimal solution. Recall that a problem exhibits

optimal substructure if an optimal solution to the problem contains within it opti-

mal solutions to subproblems. Whenever a problem exhibits optimal substructure,

we have a good clue that dynamic programming might apply. (As Chapter 16 dis-

cusses, it also might mean that a greedy strategy applies, however.) In dynamic

programming, we build an optimal solution to the problem from optimal solutions

to subproblems. Consequently, we must take care to ensure that the range of sub-

problems we consider includes those used in an optimal solution.

We discovered optimal substructure in both of the problems we have examined

in this chapter so far. In Section 15.1, we observed that the optimal way of cut-

ting up a rod of length n (if we make any cuts at all) involves optimally cutting

up the two pieces resulting from the ﬁrst cut. In Section 15.2, we observed that

an optimal parenthesization of A i A iC1 A j that splits the product between A k

and A kC1 contains within it optimal solutions to the problems of parenthesizing

A i A iC1 A k and A kC1 A kC2 A j .

You will ﬁnd yourself following a common pattern in discovering optimal sub-

structure:

1. You show that a solution to the problem consists of making a choice, such as

choosing an initial cut in a rod or choosing an index at which to split the matrix

chain. Making this choice leaves one or more subproblems to be solved.

2. You suppose that for a given problem, you are given the choice that leads to an

optimal solution. You do not concern yourself yet with how to determine this

choice. You just assume that it has been given to you.

3. Given this choice, you determine which subproblems ensue and how to best

characterize the resulting space of subproblems.

4. You show that the solutions to the subproblems used within an optimal solution

to the problem must themselves be optimal by using a “cut-and-paste” tech-

nique. You do so by supposing that each of the subproblem solutions is not

optimal and then deriving a contradiction. In particular, by “cutting out” the

nonoptimal solution to each subproblem and “pasting in” the optimal one, you

show that you can get a better solution to the original problem, thus contradict-

ing your supposition that you already had an optimal solution. If an optimal

380 Chapter 15 Dynamic Programming

solution gives rise to more than one subproblem, they are typically so similar

that you can modify the cut-and-paste argument for one to apply to the others

with little effort.

To characterize the space of subproblems, a good rule of thumb says to try to

keep the space as simple as possible and then expand it as necessary. For example,

the space of subproblems that we considered for the rod-cutting problem contained

the problems of optimally cutting up a rod of length i for each size i. This sub-

problem space worked well, and we had no need to try a more general space of

subproblems.

Conversely, suppose that we had tried to constrain our subproblem space for

matrix-chain multiplication to matrix products of the form A 1 A 2 A j . As before,

an optimal parenthesization must split this product between A k and A kC1 for some

1 k < j . Unless we could guarantee that k always equals j 1, we would ﬁnd

that we had subproblems of the form A 1 A 2 A k and A kC1 A kC2 A j , and that

the latter subproblem is not of the form A 1 A 2 A j . For this problem, we needed

to allow our subproblems to vary at “both ends,” that is, to allow both i and j to

vary in the subproblem A i A iC1 A j .

Optimal substructure varies across problem domains in two ways:

1. how many subproblems an optimal solution to the original problem uses, and

2. how many choices we have in determining which subproblem(s) to use in an

optimal solution.

In the rod-cutting problem, an optimal solution for cutting up a rod of size n

uses just one subproblem (of size n i), but we must consider n choices for i

in order to determine which one yields an optimal solution. Matrix-chain mul-

tiplication for the subchain A i A iC1 A j serves as an example with two sub-

problems and j i choices. For a given matrix A k at which we split the prod-

uct, we have two subproblems—parenthesizing A i A iC1 A k and parenthesizing

A kC1 A kC2 A j —and we must solve both of them optimally. Once we determine

the optimal solutions to subproblems, we choose from among j i candidates for

the index k.

Informally, the running time of a dynamic-programming algorithm depends on

the product of two factors: the number of subproblems overall and how many

choices we look at for each subproblem. In rod cutting, we had ‚.n/ subproblems

overall, and at most n choices to examine for each, yielding an O.n 2 / running time.

Matrix-chain multiplication had ‚.n 2 / subproblems overall, and in each we had at

most n 1 choices, giving an O.n 3 / running time (actually, a ‚.n 3 / running time,

by Exercise 15.2-5).

Usually, the subproblem graph gives an alternative way to perform the same

analysis. Each vertex corresponds to a subproblem, and the choices for a sub-

15.3 Elements of dynamic programming 381

problem are the edges incident to that subproblem. Recall that in rod cutting,

the subproblem graph had n vertices and at most n edges per vertex, yielding an

O.n 2 / running time. For matrix-chain multiplication, if we were to draw the sub-

problem graph, it would have ‚.n 2 / vertices and each vertex would have degree at

most n 1, giving a total of O.n 3 / vertices and edges.

Dynamic programming often uses optimal substructure in a bottom-up fashion.

That is, we ﬁrst ﬁnd optimal solutions to subproblems and, having solved the sub-

problems, we ﬁnd an optimal solution to the problem. Finding an optimal solu-

tion to the problem entails making a choice among subproblems as to which we

will use in solving the problem. The cost of the problem solution is usually the

subproblem costs plus a cost that is directly attributable to the choice itself. In

rod cutting, for example, ﬁrst we solved the subproblems of determining optimal

ways to cut up rods of length i for i D 0; 1; : : : ; n 1, and then we determined

which such subproblem yielded an optimal solution for a rod of length n, using

equation (15.2). The cost attributable to the choice itself is the term p i in equa-

tion (15.2). In matrix-chain multiplication, we determined optimal parenthesiza-

tions of subchains of A i A iC1 A j , and then we chose the matrix A k at which to

split the product. The cost attributable to the choice itself is the term p i1 p k p j .

In Chapter 16, we shall examine “greedy algorithms,” which have many similar-

ities to dynamic programming. In particular, problems to which greedy algorithms

apply have optimal substructure. One major difference between greedy algorithms

and dynamic programming is that instead of ﬁrst ﬁnding optimal solutions to sub-

problems and then making an informed choice, greedy algorithms ﬁrst make a

“greedy” choice—the choice that looks best at the time—and then solve a resulting

subproblem, without bothering to solve all possible related smaller subproblems.

Surprisingly, in some cases this strategy works!

Subtleties

You should be careful not to assume that optimal substructure applies when it does

not. Consider the following two problems in which we are given a directed graph

G D .V; E/ and vertices u; 2 V .

Unweighted shortest path: 3 Find a path from u to consisting of the fewest

edges. Such a path must be simple, since removing a cycle from a path pro-

duces a path with fewer edges.

3

We use the term “unweighted” to distinguish this problem from that of ﬁnding shortest paths with

weighted edges, which we shall see in Chapters 24 and 25. We can use the breadth-ﬁrst search

technique of Chapter 22 to solve the unweighted problem.

382 Chapter 15 Dynamic Programming

q r

s t

Figure 15.6 A directed graph showing that the problem of ﬁnding a longest simple path in an

unweighted directed graph does not have optimal substructure. The path q ! r ! t is a longest

simple path from q to t, but the subpath q ! r is not a longest simple path from q to r, nor is the

subpath r ! t a longest simple path from r to t.

Unweighted longest simple path: Find a simple path from u to consisting of

the most edges. We need to include the requirement of simplicity because other-

wise we can traverse a cycle as many times as we like to create paths with an

arbitrarily large number of edges.

The unweighted shortest-path problem exhibits optimal substructure, as follows.

Suppose that u ¤ , so that the problem is nontrivial. Then, any path p from u

to must contain an intermediate vertex, say w. (Note that w may be u or .)

Thus, we can decompose the path u

p into subpaths u

p 1 w

p 2 . Clearly, the

number of edges in p equals the number of edges in p 1 plus the number of edges

in p 2 . We claim that if p is an optimal (i.e., shortest) path from u to , then p 1

must be a shortest path from u to w. Why? We use a “cut-and-paste” argument:

if there were another path, say p 0

1

, from u to w with fewer edges than p 1 , then we

could cut out p 1 and paste in p 0

1

to produce a path u

p

0

1 w

p 2 with fewer edges

than p, thus contradicting p’s optimality. Symmetrically, p 2 must be a shortest

path from w to . Thus, we can ﬁnd a shortest path from u to by considering

all intermediate vertices w, ﬁnding a shortest path from u to w and a shortest path

from w to , and choosing an intermediate vertex w that yields the overall shortest

path. In Section 25.2, we use a variant of this observation of optimal substructure

to ﬁnd a shortest path between every pair of vertices on a weighted, directed graph.

You might be tempted to assume that the problem of ﬁnding an unweighted

longest simple path exhibits optimal substructure as well. After all, if we decom-

pose a longest simple path u

p into subpaths u

p 1 w

p 2 , then mustn’t p 1

be a longest simple path from u to w, and mustn’t p 2 be a longest simple path

from w to ? The answer is no! Figure 15.6 supplies an example. Consider the

path q ! r ! t, which is a longest simple path from q to t. Is q ! r a longest

simple path from q to r? No, for the path q ! s ! t ! r is a simple path

that is longer. Is r ! t a longest simple path from r to t? No again, for the path

r ! q ! s ! t is a simple path that is longer.

15.3 Elements of dynamic programming 383

This example shows that for longest simple paths, not only does the problem

lack optimal substructure, but we cannot necessarily assemble a “legal” solution

to the problem from solutions to subproblems. If we combine the longest simple

paths q ! s ! t ! r and r ! q ! s ! t, we get the path q ! s ! t ! r !

q ! s ! t, which is not simple. Indeed, the problem of ﬁnding an unweighted

longest simple path does not appear to have any sort of optimal substructure. No

efﬁcient dynamic-programming algorithm for this problem has ever been found. In

fact, this problem is NP-complete, which—as we shall see in Chapter 34—means

that we are unlikely to ﬁnd a way to solve it in polynomial time.

Why is the substructure of a longest simple path so different from that of a short-

est path? Although a solution to a problem for both longest and shortest paths uses

two subproblems, the subproblems in ﬁnding the longest simple path are not inde-

pendent, whereas for shortest paths they are. What do we mean by subproblems

being independent? We mean that the solution to one subproblem does not affect

the solution to another subproblem of the same problem. For the example of Fig-

ure 15.6, we have the problem of ﬁnding a longest simple path from q to t with two

subproblems: ﬁnding longest simple paths from q to r and from r to t. For the ﬁrst

of these subproblems, we choose the path q ! s ! t ! r, and so we have also

used the vertices s and t. We can no longer use these vertices in the second sub-

problem, since the combination of the two solutions to subproblems would yield a

path that is not simple. If we cannot use vertex t in the second problem, then we

cannot solve it at all, since t is required to be on the path that we ﬁnd, and it is

not the vertex at which we are “splicing” together the subproblem solutions (that

vertex being r). Because we use vertices s and t in one subproblem solution, we

cannot use them in the other subproblem solution. We must use at least one of them

to solve the other subproblem, however, and we must use both of them to solve it

optimally. Thus, we say that these subproblems are not independent. Looked at

another way, using resources in solving one subproblem (those resources being

vertices) renders them unavailable for the other subproblem.

Why, then, are the subproblems independent for ﬁnding a shortest path? The

answer is that by nature, the subproblems do not share resources. We claim that

if a vertex w is on a shortest path p from u to , then we can splice together any

shortest path u

p 1 w and any shortest path w

p 2 to produce a shortest path from u

to . We are assured that, other than w, no vertex can appear in both paths p 1

and p 2 . Why? Suppose that some vertex x ¤ w appears in both p 1 and p 2 , so that

we can decompose p 1 as u

p ux x w and p 2 as w x

p x . By the optimal

substructure of this problem, path p has as many edges as p 1 and p 2 together; let’s

say that p has e edges. Now let us construct a path p 0 D u

p ux x

p x from u to .

Because we have excised the paths from x to w and from w to x, each of which

contains at least one edge, path p 0

contains at most e 2 edges, which contradicts

384 Chapter 15 Dynamic Programming

the assumption that p is a shortest path. Thus, we are assured that the subproblems

for the shortest-path problem are independent.

Both problems examined in Sections 15.1 and 15.2 have independent subprob-

lems. In matrix-chain multiplication, the subproblems are multiplying subchains

A i A iC1 A k and A kC1 A kC2 A j . These subchains are disjoint, so that no ma-

trix could possibly be included in both of them. In rod cutting, to determine the

best way to cut up a rod of length n, we look at the best ways of cutting up rods

of length i for i D 0; 1; : : : ; n 1. Because an optimal solution to the length-n

problem includes just one of these subproblem solutions (after we have cut off the

ﬁrst piece), independence of subproblems is not an issue.

Overlapping subproblems

The second ingredient that an optimization problem must have for dynamic pro-

gramming to apply is that the space of subproblems must be “small” in the sense

that a recursive algorithm for the problem solves the same subproblems over and

over, rather than always generating new subproblems. Typically, the total number

of distinct subproblems is a polynomial in the input size. When a recursive algo-

rithm revisits the same problem repeatedly, we say that the optimization problem

has overlapping subproblems. 4 In contrast, a problem for which a divide-and-

conquer approach is suitable usually generates brand-new problems at each step

of the recursion. Dynamic-programming algorithms typically take advantage of

overlapping subproblems by solving each subproblem once and then storing the

solution in a table where it can be looked up when needed, using constant time per

lookup.

In Section 15.1, we brieﬂy examined how a recursive solution to rod cut-

ting makes exponentially many calls to ﬁnd solutions of smaller subproblems.

Our dynamic-programming solution takes an exponential-time recursive algorithm

down to quadratic time.

To illustrate the overlapping-subproblems property in greater detail, let us re-

examine the matrix-chain multiplication problem. Referring back to Figure 15.5,

observe that MATRIX-CHAIN-ORDER repeatedly looks up the solution to subprob-

lems in lower rows when solving subproblems in higher rows. For example, it

references entry mŒ3; 4 four times: during the computations of mŒ2; 4, mŒ1; 4,

4

It may seem strange that dynamic programming relies on subproblems being both independent

and overlapping. Although these requirements may sound contradictory, they describe two different

notions, rather than two points on the same axis. Two subproblems of the same problem are inde-

pendent if they do not share resources. Two subproblems are overlapping if they are really the same

subproblem that occurs as a subproblem of different problems.

15.3 Elements of dynamic programming 385

1..4

1..1 2..4 1..2 3..4 1..3 4..4

2..2 3..4 2..3 4..4 1..1 2..2 3..3 4..4 1..1 2..3 1..2 3..3

3..3 4..4 2..2 3..3 2..2 3..3 1..1 2..2

Figure 15.7 The recursion tree for the computation of RECURSIVE-MATRIX-CHAIN.p; 1; 4/.

Each node contains the parameters i and j . The computations performed in a shaded subtree are

replaced by a single table lookup in MEMOIZED-MATRIX-CHAIN.

mŒ3; 5, and mŒ3; 6. If we were to recompute mŒ3; 4 each time, rather than just

looking it up, the running time would increase dramatically. To see how, consider

the following (inefﬁcient) recursive procedure that determines mŒi; j , the mini-

mum number of scalar multiplications needed to compute the matrix-chain product

A i::j D A i A iC1 A j . The procedure is based directly on the recurrence (15.7).

RECURSIVE-MATRIX-CHAIN.p; i; j /

1 if i == j

2 return 0

3 mŒi; j D 1

4 for k D i to j 1

5 q D RECURSIVE-MATRIX-CHAIN.p; i; k/

C RECURSIVE-MATRIX-CHAIN.p; k C 1; j /

C p i1 p k p j

6 if q < mŒi; j

7 mŒi; j D q

8 return mŒi; j

Figure 15.7 shows the recursion tree produced by the call RECURSIVE-MATRIX-

CHAIN.p; 1; 4/. Each node is labeled by the values of the parameters i and j .

Observe that some pairs of values occur many times.

In fact, we can show that the time to compute mŒ1; n by this recursive proce-

dure is at least exponential in n. Let T .n/ denote the time taken by RECURSIVE-

MATRIX-CHAIN to compute an optimal parenthesization of a chain of n matrices.

Because the execution of lines 1–2 and of lines 6–7 each take at least unit time, as

386 Chapter 15 Dynamic Programming

does the multiplication in line 5, inspection of the procedure yields the recurrence

T .1/ 1 ;

T .n/ 1 C

n1 X

kD1

.T .k/ C T .n k/ C 1/ for n > 1 :

Noting that for i D 1; 2; : : : ; n 1, each term T .i/ appears once as T .k/ and once

as T .n k/, and collecting the n 1 1s in the summation together with the 1 out

front, we can rewrite the recurrence as

T .n/ 2

n1 X

iD1

T .i/ C n : (15.8)

We shall prove that T .n/ D .2 n / using the substitution method. Speciﬁ-

cally, we shall show that T .n/ 2 n1

for all n 1. The basis is easy, since

T .1/ 1 D 2 0

. Inductively, for n 2 we have

T .n/ 2

n1 X

iD1

2

i1

C n

D 2

n2 X

iD0

2

i

C n

D 2.2

n1

1/ C n (by equation (A.5))

D 2

n

2 C n

2

n1

;

which completes the proof. Thus, the total amount of work performed by the call

RECURSIVE-MATRIX-CHAIN.p; 1; n/ is at least exponential in n.

Compare this top-down, recursive algorithm (without memoization) with the

bottom-up dynamic-programming algorithm. The latter is more efﬁcient because

it takes advantage of the overlapping-subproblems property. Matrix-chain mul-

tiplication has only ‚.n 2 / distinct subproblems, and the dynamic-programming

algorithm solves each exactly once. The recursive algorithm, on the other hand,

must again solve each subproblem every time it reappears in the recursion tree.

Whenever a recursion tree for the natural recursive solution to a problem contains

the same subproblem repeatedly, and the total number of distinct subproblems is

small, dynamic programming can improve efﬁciency, sometimes dramatically.

15.3 Elements of dynamic programming 387

Reconstructing an optimal solution

As a practical matter, we often store which choice we made in each subproblem in

a table so that we do not have to reconstruct this information from the costs that we

stored.

For matrix-chain multiplication, the table sŒi; j saves us a signiﬁcant amount of

work when reconstructing an optimal solution. Suppose that we did not maintain

the sŒi; j table, having ﬁlled in only the table mŒi; j containing optimal subprob-

lem costs. We choose from among j i possibilities when we determine which

subproblems to use in an optimal solution to parenthesizing A i A iC1 A j , and

j i is not a constant. Therefore, it would take ‚.j i/ D !.1/ time to recon-

struct which subproblems we chose for a solution to a given problem. By storing

in sŒi; j the index of the matrix at which we split the product A i A iC1 A j , we

can reconstruct each choice in O.1/ time.

Memoization

As we saw for the rod-cutting problem, there is an alternative approach to dy-

namic programming that often offers the efﬁciency of the bottom-up dynamic-

programming approach while maintaining a top-down strategy. The idea is to

memoize the natural, but inefﬁcient, recursive algorithm. As in the bottom-up ap-

proach, we maintain a table with subproblem solutions, but the control structure

for ﬁlling in the table is more like the recursive algorithm.

A memoized recursive algorithm maintains an entry in a table for the solution to

each subproblem. Each table entry initially contains a special value to indicate that

the entry has yet to be ﬁlled in. When the subproblem is ﬁrst encountered as the

recursive algorithm unfolds, its solution is computed and then stored in the table.

Each subsequent time that we encounter this subproblem, we simply look up the

value stored in the table and return it. 5

Here is a memoized version of RECURSIVE-MATRIX-CHAIN. Note where it

resembles the memoized top-down method for the rod-cutting problem.

5

This approach presupposes that we know the set of all possible subproblem parameters and that we

have established the relationship between table positions and subproblems. Another, more general,

approach is to memoize by using hashing with the subproblem parameters as keys.

388 Chapter 15 Dynamic Programming

MEMOIZED-MATRIX-CHAIN.p/

1 n D p:length 1

2 let mŒ1 : : n; 1 : : n be a new table

3 for i D 1 to n

4 for j D i to n

5 mŒi; j D 1

6 return LOOKUP-CHAIN.m; p; 1; n/

LOOKUP-CHAIN.m; p; i; j /

1 if mŒi; j < 1

2 return mŒi; j

3 if i == j

4 mŒi; j D 0

5 else for k D i to j 1

6 q D LOOKUP-CHAIN.m; p; i; k/

C LOOKUP-CHAIN.m; p; k C 1; j / C p i1 p k p j

7 if q < mŒi; j

8 mŒi; j D q

9 return mŒi; j

The MEMOIZED-MATRIX-CHAIN procedure, like MATRIX-CHAIN-ORDER,

maintains a table mŒ1 : : n; 1 : : n of computed values of mŒi; j , the minimum num-

ber of scalar multiplications needed to compute the matrix A i::j . Each table entry

initially contains the value 1 to indicate that the entry has yet to be ﬁlled in. Upon

calling LOOKUP-CHAIN.m; p; i; j /, if line 1 ﬁnds that mŒi; j < 1, then the pro-

cedure simply returns the previously computed cost mŒi; j in line 2. Otherwise,

the cost is computed as in RECURSIVE-MATRIX-CHAIN, stored in mŒi; j , and

returned. Thus, LOOKUP-CHAIN.m; p; i; j / always returns the value of mŒi; j ,

but it computes it only upon the ﬁrst call of LOOKUP-CHAIN with these speciﬁc

values of i and j .

Figure 15.7 illustrates how MEMOIZED-MATRIX-CHAIN saves time compared

with RECURSIVE-MATRIX-CHAIN. Shaded subtrees represent values that it looks

up rather than recomputes.

Like the bottom-up dynamic-programming algorithm MATRIX-CHAIN-ORDER,

the procedure MEMOIZED-MATRIX-CHAIN runs in O.n 3 / time. Line 5 of

MEMOIZED-MATRIX-CHAIN executes ‚.n 2 / times. We can categorize the calls

of LOOKUP-CHAIN into two types:

1. calls in which mŒi; j D 1, so that lines 3–9 execute, and

2. calls in which mŒi; j < 1, so that LOOKUP-CHAIN simply returns in line 2.

15.3 Elements of dynamic programming 389

There are ‚.n 2 / calls of the ﬁrst type, one per table entry. All calls of the sec-

ond type are made as recursive calls by calls of the ﬁrst type. Whenever a given

call of LOOKUP-CHAIN makes recursive calls, it makes O.n/ of them. There-

fore, there are O.n 3 / calls of the second type in all. Each call of the second type

takes O.1/ time, and each call of the ﬁrst type takes O.n/ time plus the time spent

in its recursive calls. The total time, therefore, is O.n 3 /. Memoization thus turns

an .2 n /-time algorithm into an O.n 3 /-time algorithm.

In summary, we can solve the matrix-chain multiplication problem by either a

top-down, memoized dynamic-programming algorithm or a bottom-up dynamic-

programming algorithm in O.n 3 / time. Both methods take advantage of the

overlapping-subproblems property. There are only ‚.n 2 / distinct subproblems in

total, and either of these methods computes the solution to each subproblem only

once. Without memoization, the natural recursive algorithm runs in exponential

time, since solved subproblems are repeatedly solved.

In general practice, if all subproblems must be solved at least once, a bottom-up

dynamic-programming algorithm usually outperforms the corresponding top-down

memoized algorithm by a constant factor, because the bottom-up algorithm has no

overhead for recursion and less overhead for maintaining the table. Moreover, for

some problems we can exploit the regular pattern of table accesses in the dynamic-

programming algorithm to reduce time or space requirements even further. Alter-

natively, if some subproblems in the subproblem space need not be solved at all,

the memoized solution has the advantage of solving only those subproblems that

are deﬁnitely required.

Exercises

15.3-1

Which is a more efﬁcient way to determine the optimal number of multiplications

in a matrix-chain multiplication problem: enumerating all the ways of parenthesiz-

ing the product and computing the number of multiplications for each, or running

RECURSIVE-MATRIX-CHAIN? Justify your answer.

15.3-2

Draw the recursion tree for the MERGE-SORT procedure from Section 2.3.1 on an

array of 16 elements. Explain why memoization fails to speed up a good divide-

and-conquer algorithm such as MERGE-SORT.

15.3-3

Consider a variant of the matrix-chain multiplication problem in which the goal is

to parenthesize the sequence of matrices so as to maximize, rather than minimize,

390 Chapter 15 Dynamic Programming

the number of scalar multiplications. Does this problem exhibit optimal substruc-

ture?

15.3-4

As stated, in dynamic programming we ﬁrst solve the subproblems and then choose

which of them to use in an optimal solution to the problem. Professor Capulet

claims that we do not always need to solve all the subproblems in order to ﬁnd an

optimal solution. She suggests that we can ﬁnd an optimal solution to the matrix-

chain multiplication problem by always choosing the matrix A k at which to split

the subproduct A i A iC1 A j (by selecting k to minimize the quantity p i1 p k p j )

before solving the subproblems. Find an instance of the matrix-chain multiplica-

tion problem for which this greedy approach yields a suboptimal solution.

15.3-5

Suppose that in the rod-cutting problem of Section 15.1, we also had limit l i on the

number of pieces of length i that we are allowed to produce, for i D 1; 2; : : : ; n.

Show that the optimal-substructure property described in Section 15.1 no longer

holds.

15.3-6

Imagine that you wish to exchange one currency for another. You realize that

instead of directly exchanging one currency for another, you might be better off

making a series of trades through other currencies, winding up with the currency

you want. Suppose that you can trade n different currencies, numbered 1; 2; : : : ; n,

where you start with currency 1 and wish to wind up with currency n. You are

given, for each pair of currencies i and j , an exchange rate r ij , meaning that if

you start with d units of currency i, you can trade for dr ij units of currency j .

A sequence of trades may entail a commission, which depends on the number of

trades you make. Let c k be the commission that you are charged when you make k

trades. Show that, if c k D 0 for all k D 1; 2; : : : ; n, then the problem of ﬁnding the

best sequence of exchanges from currency 1 to currency n exhibits optimal sub-

structure. Then show that if commissions c k are arbitrary values, then the problem

of ﬁnding the best sequence of exchanges from currency 1 to currency n does not

necessarily exhibit optimal substructure.

15.4 Longest common subsequence

Biological applications often need to compare the DNA of two (or more) dif-

ferent organisms. A strand of DNA consists of a string of molecules called

15.4 Longest common subsequence 391

bases, where the possible bases are adenine, guanine, cytosine, and thymine.

Representing each of these bases by its initial letter, we can express a strand

of DNA as a string over the ﬁnite set fA; C; G; Tg. (See Appendix C for

the deﬁnition of a string.) For example, the DNA of one organism may be

S 1 D ACCGGTCGAGTGCGCGGAAGCCGGCCGAA, and the DNA of another organ-

ism may be S 2 D GTCGTTCGGAATGCCGTTGCTCTGTAAA. One reason to com-

pare two strands of DNA is to determine how “similar” the two strands are, as some

measure of how closely related the two organisms are. We can, and do, deﬁne sim-

ilarity in many different ways. For example, we can say that two DNA strands are

similar if one is a substring of the other. (Chapter 32 explores algorithms to solve

this problem.) In our example, neither S 1 nor S 2 is a substring of the other. Alter-

natively, we could say that two strands are similar if the number of changes needed

to turn one into the other is small. (Problem 15-5 looks at this notion.) Yet another

way to measure the similarity of strands S 1 and S 2 is by ﬁnding a third strand S 3

in which the bases in S 3 appear in each of S 1 and S 2 ; these bases must appear

in the same order, but not necessarily consecutively. The longer the strand S 3 we

can ﬁnd, the more similar S 1 and S 2 are. In our example, the longest strand S 3 is

GTCGTCGGAAGCCGGCCGAA.

We formalize this last notion of similarity as the longest-common-subsequence

problem. A subsequence of a given sequence is just the given sequence with zero or

more elements left out. Formally, given a sequence X D hx 1 ; x 2 ; : : : ; x m i, another

sequence Z D h´ 1 ; ´ 2 ; : : : ; ´ k i is a subsequence of X if there exists a strictly

increasing sequence hi 1 ;i 2 ;: : : ;i k i of indices of X such that for all j D 1; 2; : : : ; k,

we have x i j D ´ j . For example, Z D hB; C; D; Bi is a subsequence of X D

hA; B; C; B; D; A; Bi with corresponding index sequence h2; 3; 5; 7i.

Given two sequences X and Y , we say that a sequence Z is a common sub-

sequence of X and Y if Z is a subsequence of both X and Y . For example, if

X D hA;B;C;B;D;A;Bi and Y D hB;D;C;A;B;Ai, the sequence hB;C;Ai is

a common subsequence of both X and Y . The sequence hB; C; Ai is not a longest

common subsequence (LCS) of X and Y , however, since it has length 3 and the

sequence hB; C; B; Ai, which is also common to both X and Y , has length 4. The

sequence hB; C; B; Ai is an LCS of X and Y , as is the sequence hB; D; A; Bi,

since X and Y have no common subsequence of length 5 or greater.

In the longest-common-subsequence problem, we are given two sequences

X D hx 1 ; x 2 ; : : : ; x m i and Y D hy 1 ; y 2 ; : : : ; y n i and wish to ﬁnd a maximum-

length common subsequence of X and Y . This section shows how to efﬁciently

solve the LCS problem using dynamic programming.

392 Chapter 15 Dynamic Programming

Step 1: Characterizing a longest common subsequence

In a brute-force approach to solving the LCS problem, we would enumerate all

subsequences of X and check each subsequence to see whether it is also a subse-

quence of Y , keeping track of the longest subsequence we ﬁnd. Each subsequence

of X corresponds to a subset of the indices f1; 2; : : : ; mg of X. Because X has 2 m

subsequences, this approach requires exponential time, making it impractical for

long sequences.

The LCS problem has an optimal-substructure property, however, as the follow-

ing theorem shows. As we shall see, the natural classes of subproblems corre-

spond to pairs of “preﬁxes” of the two input sequences. To be precise, given a

sequence X D hx 1 ;x 2 ;: : : ;x m i, we deﬁne the ith preﬁx of X, for i D 0; 1; : : : ; m,

as X i D hx 1 ; x 2 ; : : : ; x i i. For example, if X D hA; B; C; B; D; A; Bi, then

X 4 D hA; B; C; Bi and X 0 is the empty sequence.

Theorem 15.1 (Optimal substructure of an LCS)

Let X D hx 1 ; x 2 ; : : : ; x m i and Y D hy 1 ; y 2 ; : : : ; y n i be sequences, and let Z D

h´ 1 ; ´ 2 ; : : : ; ´ k i be any LCS of X and Y .

1. If x m D y n , then ´ k D x m D y n and Z k1 is an LCS of X m1 and Y n1 .

2. If x m ¤ y n , then ´ k ¤ x m implies that Z is an LCS of X m1 and Y .

3. If x m ¤ y n , then ´ k ¤ y n implies that Z is an LCS of X and Y n1 .

Proof (1) If ´ k ¤ x m , then we could append x m D y n to Z to obtain a common

subsequence of X and Y of length k C 1, contradicting the supposition that Z is

a longest common subsequence of X and Y . Thus, we must have ´ k D x m D y n .

Now, the preﬁx Z k1 is a length-.k 1/ common subsequence of X m1 and Y n1 .

We wish to show that it is an LCS. Suppose for the purpose of contradiction

that there exists a common subsequence W of X m1 and Y n1 with length greater

than k 1. Then, appending x m D y n to W produces a common subsequence of

X and Y whose length is greater than k, which is a contradiction.

(2) If ´ k ¤ x m , then Z is a common subsequence of X m1 and Y . If there were a

common subsequence W of X m1 and Y with length greater than k, then W would

also be a common subsequence of X m and Y , contradicting the assumption that Z

is an LCS of X and Y .

(3) The proof is symmetric to (2).

The way that Theorem 15.1 characterizes longest common subsequences tells

us that an LCS of two sequences contains within it an LCS of preﬁxes of the two

sequences. Thus, the LCS problem has an optimal-substructure property. A recur-

15.4 Longest common subsequence 393

sive solution also has the overlapping-subproblems property, as we shall see in a

moment.

Step 2: A recursive solution

Theorem 15.1 implies that we should examine either one or two subproblems when

ﬁnding an LCS of X D hx 1 ; x 2 ; : : : ; x m i and Y D hy 1 ; y 2 ; : : : ; y n i. If x m D y n ,

we must ﬁnd an LCS of X m1 and Y n1 . Appending x m D y n to this LCS yields

an LCS of X and Y . If x m ¤ y n , then we must solve two subproblems: ﬁnding an

LCS of X m1 and Y and ﬁnding an LCS of X and Y n1 . Whichever of these two

LCSs is longer is an LCS of X and Y . Because these cases exhaust all possibilities,

we know that one of the optimal subproblem solutions must appear within an LCS

of X and Y .

We can readily see the overlapping-subproblems property in the LCS problem.

To ﬁnd an LCS of X and Y , we may need to ﬁnd the LCSs of X and Y n1 and

of X m1 and Y . But each of these subproblems has the subsubproblem of ﬁnding

an LCS of X m1 and Y n1 . Many other subproblems share subsubproblems.

As in the matrix-chain multiplication problem, our recursive solution to the LCS

problem involves establishing a recurrence for the value of an optimal solution.

Let us deﬁne cŒi; j to be the length of an LCS of the sequences X i and Y j . If

either i D 0 or j D 0, one of the sequences has length 0, and so the LCS has

length 0. The optimal substructure of the LCS problem gives the recursive formula

cŒi; j D

0 if i D 0 or j D 0 ;

cŒi 1; j 1 C 1 if i; j > 0 and x i D y j ;

max.cŒi; j 1; cŒi 1; j / if i; j > 0 and x i ¤ y j :

(15.9)

Observe that in this recursive formulation, a condition in the problem restricts

which subproblems we may consider. When x i D y j , we can and should consider

the subproblem of ﬁnding an LCS of X i1 and Y j 1 . Otherwise, we instead con-

sider the two subproblems of ﬁnding an LCS of X i and Y j 1 and of X i1 and Y j . In

the previous dynamic-programming algorithms we have examined—for rod cutting

and matrix-chain multiplication—we ruled out no subproblems due to conditions

in the problem. Finding an LCS is not the only dynamic-programming algorithm

that rules out subproblems based on conditions in the problem. For example, the

edit-distance problem (see Problem 15-5) has this characteristic.

Step 3: Computing the length of an LCS

Based on equation (15.9), we could easily write an exponential-time recursive al-

gorithm to compute the length of an LCS of two sequences. Since the LCS problem

394 Chapter 15 Dynamic Programming

has only ‚.mn/ distinct subproblems, however, we can use dynamic programming

to compute the solutions bottom up.

Procedure LCS-LENGTH takes two sequences X D hx 1 ; x 2 ; : : : ; x m i and

Y D hy 1 ;y 2 ;: : : ;y n i as inputs. It stores the cŒi; j values in a table cŒ0 : : m; 0 : : n,

and it computes the entries in row-major order. (That is, the procedure ﬁlls in the

ﬁrst row of c from left to right, then the second row, and so on.) The procedure also

maintains the table bŒ1 : : m; 1 : : n to help us construct an optimal solution. Intu-

itively, bŒi; j points to the table entry corresponding to the optimal subproblem

solution chosen when computing cŒi; j . The procedure returns the b and c tables;

cŒm; n contains the length of an LCS of X and Y .

LCS-LENGTH.X; Y /

1 m D X:length

2 n D Y:length

3 let bŒ1 : : m; 1 : : n and cŒ0 : : m; 0 : : n be new tables

4 for i D 1 to m

5 cŒi; 0 D 0

6 for j D 0 to n

7 cŒ0; j D 0

8 for i D 1 to m

9 for j D 1 to n

10 if x i == y j

11 cŒi; j D cŒi 1; j 1 C 1

12 bŒi; j D “-”

13 elseif cŒi 1; j cŒi; j 1

14 cŒi; j D cŒi 1; j

15 bŒi; j D “"”

16 else cŒi; j D cŒi; j 1

17 bŒi; j D “ ”

18 return c and b

Figure 15.8 shows the tables produced by LCS-LENGTH on the sequences X D

hA; B; C; B; D; A; Bi and Y D hB; D; C; A; B; Ai. The running time of the

procedure is ‚.mn/, since each table entry takes ‚.1/ time to compute.

Step 4: Constructing an LCS

The b table returned by LCS-LENGTH enables us to quickly construct an LCS of

X D hx 1 ; x 2 ; : : : ; x m i and Y D hy 1 ; y 2 ; : : : ; y n i. We simply begin at bŒm; n and

trace through the table by following the arrows. Whenever we encounter a “-” in

entry bŒi; j , it implies that x i D y j is an element of the LCS that LCS-LENGTH

15.4 Longest common subsequence 395

0 0 0 0 0 0 0

0 0 0 0 1 1 1

0 1 1 1 2 2

0 1 1 2 2 2

0 1 1 2 2 3

0 1 2 2 2 3 3

0 1 2 2 3 3

0 1 2 2 3 4 4

1

2

3

4

B D C A B A

1 2 3 4 5 6 0

A

B

C

B

D

A

B

1

2

3

4

5

6

7

0

j

i

x

i

y

j

Figure 15.8 The c and b tables computed by LCS-LENGTH on the sequences X D hA; B; C; B;

D;A;Bi and Y D hB;D;C;A;B;Ai. The square in row i and column j contains the value of cŒi; j

and the appropriate arrow for the value of bŒi; j . The entry 4 in cŒ7; 6—the lower right-hand corner

of the table—is the length of an LCS hB; C; B; Ai of X and Y . For i; j > 0, entry cŒi; j depends

only on whether xi D yj and the values in entries cŒi 1; j , cŒi; j 1, and cŒi 1; j 1, which

are computed before cŒi; j . To reconstruct the elements of an LCS, follow the bŒi; j arrows from

the lower right-hand corner; the sequence is shaded. Each “-” on the shaded sequence corresponds

to an entry (highlighted) for which xi D yj is a member of an LCS.

found. With this method, we encounter the elements of this LCS in reverse order.

The following recursive procedure prints out an LCS of X and Y in the proper,

forward order. The initial call is PRINT-LCS.b; X; X:length; Y:length/.

PRINT-LCS.b; X; i; j /

1 if i == 0 or j == 0

2 return

3 if bŒi; j == “-”

4 PRINT-LCS.b; X; i 1; j 1/

5 print x i

6 elseif bŒi; j == “"”

7 PRINT-LCS.b; X; i 1; j /

8 else PRINT-LCS.b; X; i; j 1/

For the b table in Figure 15.8, this procedure prints BCBA. The procedure takes

time O.m C n/, since it decrements at least one of i and j in each recursive call.

396 Chapter 15 Dynamic Programming

Improving the code

Once you have developed an algorithm, you will often ﬁnd that you can improve

on the time or space it uses. Some changes can simplify the code and improve

constant factors but otherwise yield no asymptotic improvement in performance.

Others can yield substantial asymptotic savings in time and space.

In the LCS algorithm, for example, we can eliminate the b table altogether. Each

cŒi; j entry depends on only three other c table entries: cŒi 1; j 1, cŒi 1; j ,

and cŒi; j 1. Given the value of cŒi; j , we can determine in O.1/ time which of

these three values was used to compute cŒi; j , without inspecting table b. Thus, we

can reconstruct an LCS in O.mCn/ time using a procedure similar to PRINT-LCS.

(Exercise 15.4-2 asks you to give the pseudocode.) Although we save ‚.mn/ space

by this method, the auxiliary space requirement for computing an LCS does not

asymptotically decrease, since we need ‚.mn/ space for the c table anyway.

We can, however, reduce the asymptotic space requirements for LCS-LENGTH,

since it needs only two rows of table c at a time: the row being computed and the

previous row. (In fact, as Exercise 15.4-4 asks you to show, we can use only slightly

more than the space for one row of c to compute the length of an LCS.) This

improvement works if we need only the length of an LCS; if we need to reconstruct

the elements of an LCS, the smaller table does not keep enough information to

retrace our steps in O.m C n/ time.

Exercises

15.4-1

Determine an LCS of h1; 0; 0; 1; 0; 1; 0; 1i and h0; 1; 0; 1; 1; 0; 1; 1; 0i.

15.4-2

Give pseudocode to reconstruct an LCS from the completed c table and the original

sequences X D hx 1 ; x 2 ; : : : ; x m i and Y D hy 1 ; y 2 ; : : : ; y n i in O.m C n/ time,

without using the b table.

15.4-3

Give a memoized version of LCS-LENGTH that runs in O.mn/ time.

15.4-4

Show how to compute the length of an LCS using only 2min.m; n/ entries in the c

table plus O.1/ additional space. Then show how to do the same thing, but using

min.m; n/ entries plus O.1/ additional space.

15.5 Optimal binary search trees 397

15.4-5

Give an O.n 2 /-time algorithm to ﬁnd the longest monotonically increasing subse-

quence of a sequence of n numbers.

15.4-6 ?

Give an O.n lg n/-time algorithm to ﬁnd the longest monotonically increasing sub-

sequence of a sequence of n numbers. (Hint: Observe that the last element of a

candidate subsequence of length i is at least as large as the last element of a can-

didate subsequence of length i 1. Maintain candidate subsequences by linking

them through the input sequence.)

15.5 Optimal binary search trees

Suppose that we are designing a program to translate text from English to French.

For each occurrence of each English word in the text, we need to look up its French

equivalent. We could perform these lookup operations by building a binary search

tree with n English words as keys and their French equivalents as satellite data.

Because we will search the tree for each individual word in the text, we want the

total time spent searching to be as low as possible. We could ensure an O.lg n/

search time per occurrence by using a red-black tree or any other balanced binary

search tree. Words appear with different frequencies, however, and a frequently

used word such as the may appear far from the root while a rarely used word such

as machicolation appears near the root. Such an organization would slow down the

translation, since the number of nodes visited when searching for a key in a binary

search tree equals one plus the depth of the node containing the key. We want

words that occur frequently in the text to be placed nearer the root. 6 Moreover,

some words in the text might have no French translation, 7 and such words would

not appear in the binary search tree at all. How do we organize a binary search tree

so as to minimize the number of nodes visited in all searches, given that we know

how often each word occurs?

What we need is known as an optimal binary search tree. Formally, we are

given a sequence K D hk 1 ; k 2 ; : : : ; k n i of n distinct keys in sorted order (so that

k 1 < k 2 < < k n ), and we wish to build a binary search tree from these keys.

For each key k i , we have a probability p i that a search will be for k i . Some

searches may be for values not in K, and so we also have n C 1 “dummy keys”

6

If the subject of the text is castle architecture, we might want machicolation to appear near the root.

7

Yes, machicolation has a French counterpart: mˆachicoulis.

398 Chapter 15 Dynamic Programming

k

2

k

1

k

4

k

3

k

5

d

0

d

1

d

2

d

3

d

4

d

5

(a)

k

2

k

1

k

4

k

3

k

5

d

0

d

1

d

2

d

3

d

4

d

5

(b)

Figure 15.9 Two binary search trees for a set of n D 5 keys with the following probabilities:

i 0 1 2 3 4 5

pi 0.15 0.10 0.05 0.10 0.20

qi 0.05 0.10 0.05 0.05 0.05 0.10

(a) A binary search tree with expected search cost 2.80. (b) A binary search tree with expected search

cost 2.75. This tree is optimal.

d 0 ; d 1 ; d 2 ; : : : ; d n representing values not in K. In particular, d 0 represents all val-

ues less than k 1 , d n represents all values greater than k n , and for i D 1; 2; : : : ; n1,

the dummy key d i represents all values between k i and k iC1 . For each dummy

key d i , we have a probability q i that a search will correspond to d i . Figure 15.9

shows two binary search trees for a set of n D 5 keys. Each key k i is an internal

node, and each dummy key d i is a leaf. Every search is either successful (ﬁnding

some key k i ) or unsuccessful (ﬁnding some dummy key d i ), and so we have

n X

iD1

p i C

n X

iD0

q i D 1 : (15.10)

Because we have probabilities of searches for each key and each dummy key,

we can determine the expected cost of a search in a given binary search tree T . Let

us assume that the actual cost of a search equals the number of nodes examined,

i.e., the depth of the node found by the search in T , plus 1. Then the expected cost

of a search in T is

E Œsearch cost in T D

n X

iD1

.depth

T

.k i / C 1/ p i C

n X

iD0

.depth

T

.d i / C 1/ q i

D 1 C

n X

iD1

depth T .k i / p i C

n X

iD0

depth T .d i / q i ; (15.11)

15.5 Optimal binary search trees 399

where depth

T

denotes a node’s depth in the tree T . The last equality follows from

equation (15.10). In Figure 15.9(a), we can calculate the expected search cost node

by node:

node depth probability contribution

k1 1 0.15 0.30

k2 0 0.10 0.10

k3 2 0.05 0.15

k4 1 0.10 0.20

k5 2 0.20 0.60

d0 2 0.05 0.15

d1 2 0.10 0.30

d2 3 0.05 0.20

d3 3 0.05 0.20

d4 3 0.05 0.20

d5 3 0.10 0.40

Total 2.80

For a given set of probabilities, we wish to construct a binary search tree whose

expected search cost is smallest. We call such a tree an optimal binary search tree.

Figure 15.9(b) shows an optimal binary search tree for the probabilities given in

the ﬁgure caption; its expected cost is 2.75. This example shows that an optimal

binary search tree is not necessarily a tree whose overall height is smallest. Nor

can we necessarily construct an optimal binary search tree by always putting the

key with the greatest probability at the root. Here, key k 5 has the greatest search

probability of any key, yet the root of the optimal binary search tree shown is k 2 .

(The lowest expected cost of any binary search tree with k 5 at the root is 2.85.)

As with matrix-chain multiplication, exhaustive checking of all possibilities fails

to yield an efﬁcient algorithm. We can label the nodes of any n-node binary tree

with the keys k 1 ; k 2 ; : : : ; k n to construct a binary search tree, and then add in the

dummy keys as leaves. In Problem 12-4, we saw that the number of binary trees

with n nodes is .4 n =n 3=2 /, and so we would have to examine an exponential

number of binary search trees in an exhaustive search. Not surprisingly, we shall

solve this problem with dynamic programming.

Step 1: The structure of an optimal binary search tree

To characterize the optimal substructure of optimal binary search trees, we start

with an observation about subtrees. Consider any subtree of a binary search tree.

It must contain keys in a contiguous range k i ; : : : ; k j , for some 1 i j n.

In addition, a subtree that contains keys k i ; : : : ; k j must also have as its leaves the

dummy keys d i1 ; : : : ; d j .

Now we can state the optimal substructure: if an optimal binary search tree T

has a subtree T 0

containing keys k i ; : : : ; k j , then this subtree T 0

must be optimal as

400 Chapter 15 Dynamic Programming

well for the subproblem with keys k i ; : : : ; k j and dummy keys d i1 ; : : : ; d j . The

usual cut-and-paste argument applies. If there were a subtree T 00

whose expected

cost is lower than that of T 0

, then we could cut T 0

out of T and paste in T 00

,

resulting in a binary search tree of lower expected cost than T , thus contradicting

the optimality of T .

We need to use the optimal substructure to show that we can construct an opti-

mal solution to the problem from optimal solutions to subproblems. Given keys

k i ; : : : ; k j , one of these keys, say k r (i r j ), is the root of an optimal

subtree containing these keys. The left subtree of the root k r contains the keys

k i ; : : : ; k r1 (and dummy keys d i1 ; : : : ; d r1 ), and the right subtree contains the

keys k rC1 ; : : : ; k j (and dummy keys d r ; : : : ; d j ). As long as we examine all candi-

date roots k r , where i r j , and we determine all optimal binary search trees

containing k i ; : : : ; k r1 and those containing k rC1 ; : : : ; k j , we are guaranteed that

we will ﬁnd an optimal binary search tree.

There is one detail worth noting about “empty” subtrees. Suppose that in a

subtree with keys k i ; : : : ; k j , we select k i as the root. By the above argument, k i ’s

left subtree contains the keys k i ; : : : ; k i1 . We interpret this sequence as containing

no keys. Bear in mind, however, that subtrees also contain dummy keys. We adopt

the convention that a subtree containing keys k i ; : : : ; k i1 has no actual keys but

does contain the single dummy key d i1 . Symmetrically, if we select k j as the root,

then k j ’s right subtree contains the keys k j C1 ; : : : ; k j ; this right subtree contains

no actual keys, but it does contain the dummy key d j .

Step 2: A recursive solution

We are ready to deﬁne the value of an optimal solution recursively. We pick our

subproblem domain as ﬁnding an optimal binary search tree containing the keys

k i ; : : : ; k j , where i 1, j n, and j i 1. (When j D i 1, there

are no actual keys; we have just the dummy key d i1 .) Let us deﬁne eŒi; j as

the expected cost of searching an optimal binary search tree containing the keys

k i ; : : : ; k j . Ultimately, we wish to compute eŒ1; n.

The easy case occurs when j D i 1. Then we have just the dummy key d i1 .

The expected search cost is eŒi; i 1 D q i1 .

When j i, we need to select a root k r from among k i ; : : : ; k j and then make an

optimal binary search tree with keys k i ; : : : ; k r1 as its left subtree and an optimal

binary search tree with keys k rC1 ; : : : ; k j as its right subtree. What happens to the

expected search cost of a subtree when it becomes a subtree of a node? The depth

of each node in the subtree increases by 1. By equation (15.11), the expected search

cost of this subtree increases by the sum of all the probabilities in the subtree. For

a subtree with keys k i ; : : : ; k j , let us denote this sum of probabilities as

15.5 Optimal binary search trees 401

w.i; j / D

j X

lDi

p l C

j X

lDi1

q l : (15.12)

Thus, if k r is the root of an optimal subtree containing keys k i ; : : : ; k j , we have

eŒi; j D p r C .eŒi; r 1 C w.i; r 1// C .eŒr C 1; j C w.r C 1; j // :

Noting that

w.i; j / D w.i; r 1/ C p r C w.r C 1; j / ;

we rewrite eŒi; j as

eŒi; j D eŒi; r 1 C eŒr C 1; j C w.i; j / : (15.13)

The recursive equation (15.13) assumes that we know which node k r to use as

the root. We choose the root that gives the lowest expected search cost, giving us

our ﬁnal recursive formulation:

eŒi; j D

(

q i1 if j D i 1 ;

min

irj

feŒi; r 1 C eŒr C 1; j C w.i; j /g if i j :

(15.14)

The eŒi; j values give the expected search costs in optimal binary search trees.

To help us keep track of the structure of optimal binary search trees, we deﬁne

rootŒi; j , for 1 i j n, to be the index r for which k r is the root of an

optimal binary search tree containing keys k i ; : : : ; k j . Although we will see how

to compute the values of rootŒi; j , we leave the construction of an optimal binary

search tree from these values as Exercise 15.5-1.

Step 3: Computing the expected search cost of an optimal binary search tree

At this point, you may have noticed some similarities between our characterizations

of optimal binary search trees and matrix-chain multiplication. For both problem

domains, our subproblems consist of contiguous index subranges. A direct, recur-

sive implementation of equation (15.14) would be as inefﬁcient as a direct, recur-

sive matrix-chain multiplication algorithm. Instead, we store the eŒi; j values in a

table eŒ1 : : nC1; 0 : : n. The ﬁrst index needs to run to nC1 rather than n because

in order to have a subtree containing only the dummy key d n , we need to compute

and store eŒn C 1; n. The second index needs to start from 0 because in order to

have a subtree containing only the dummy key d 0 , we need to compute and store

eŒ1; 0. We use only the entries eŒi; j for which j i 1. We also use a table

rootŒi; j , for recording the root of the subtree containing keys k i ; : : : ; k j . This

table uses only the entries for which 1 i j n.

We will need one other table for efﬁciency. Rather than compute the value

of w.i; j / from scratch every time we are computing eŒi; j —which would take

402 Chapter 15 Dynamic Programming

‚.j i/ additions—we store these values in a table wŒ1 : : n C 1; 0 : : n. For the

base case, we compute wŒi; i 1 D q i1 for 1 i n C 1. For j i, we

compute

wŒi; j D wŒi; j 1 C p j C q j : (15.15)

Thus, we can compute the ‚.n 2 / values of wŒi; j in ‚.1/ time each.

The pseudocode that follows takes as inputs the probabilities p 1 ; : : : ; p n and

q 0 ; : : : ; q n and the size n, and it returns the tables e and root.

OPTIMAL-BST.p; q; n/

1 let eŒ1 : : n C 1; 0 : : n, wŒ1 : : n C 1; 0 : : n,

and rootŒ1 : : n; 1 : : n be new tables

2 for i D 1 to n C 1

3 eŒi; i 1 D q i1

4 wŒi; i 1 D q i1

5 for l D 1 to n

6 for i D 1 to n l C 1

7 j D i C l 1

8 eŒi; j D 1

9 wŒi; j D wŒi; j 1 C p j C q j

10 for r D i to j

11 t D eŒi; r 1 C eŒr C 1; j C wŒi; j

12 if t < eŒi; j

13 eŒi; j D t

14 rootŒi; j D r

15 return e and root

From the description above and the similarity to the MATRIX-CHAIN-ORDER pro-

cedure in Section 15.2, you should ﬁnd the operation of this procedure to be fairly

straightforward. The for loop of lines 2–4 initializes the values of eŒi; i 1

and wŒi; i 1. The for loop of lines 5–14 then uses the recurrences (15.14)

and (15.15) to compute eŒi; j and wŒi; j for all 1 i j n. In the ﬁrst itera-

tion, when l D 1, the loop computes eŒi; i and wŒi; i for i D 1; 2; : : : ; n. The sec-

ond iteration, with l D 2, computes eŒi; iC1 and wŒi; iC1 for i D 1; 2; : : : ; n1,

and so forth. The innermost for loop, in lines 10–14, tries each candidate index r

to determine which key k r to use as the root of an optimal binary search tree con-

taining keys k i ; : : : ; k j . This for loop saves the current value of the index r in

rootŒi; j whenever it ﬁnds a better key to use as the root.

Figure 15.10 shows the tables eŒi; j , wŒi; j , and rootŒi; j computed by the

procedure OPTIMAL-BST on the key distribution shown in Figure 15.9. As in the

matrix-chain multiplication example of Figure 15.5, the tables are rotated to make

15.5 Optimal binary search trees 403

2.75

1.75

1.25

0.90

0.45

0.05

2.00

1.20

0.70

0.40

0.10

1.30

0.60

0.25

0.05

0.90

0.30

0.05

0.50

0.05 0.10

e

0

1

2

3

4

5

6

5

4

3

2

1

j i 1.00

0.70

0.55

0.45

0.30

0.05

0.80

0.50

0.35

0.25

0.10

0.60

0.30

0.15

0.05

0.50

0.20

0.05

0.35

0.05 0.10

w

0

1

2

3

4

5

6

5

4

3

2

1

j i

2

2

2

1

1

4

2

2

2

5

4

3

5

4 5

root

1

2

3

4

5

5

4

3

2

1

j i

Figure 15.10 The tables eŒi; j , wŒi; j , and rootŒi; j computed by OPTIMAL-BST on the key

distribution shown in Figure 15.9. The tables are rotated so that the diagonals run horizontally.

the diagonals run horizontally. OPTIMAL-BST computes the rows from bottom to

top and from left to right within each row.

The OPTIMAL-BST procedure takes ‚.n 3 / time, just like MATRIX-CHAIN-

ORDER. We can easily see that its running time is O.n 3 /, since its for loops are

nested three deep and each loop index takes on at most n values. The loop indices in

OPTIMAL-BST do not have exactly the same bounds as those in MATRIX-CHAIN-

ORDER, but they are within at most 1 in all directions. Thus, like MATRIX-CHAIN-

ORDER, the OPTIMAL-BST procedure takes .n 3 / time.

Exercises

15.5-1

Write pseudocode for the procedure CONSTRUCT-OPTIMAL-BST.root/ which,

given the table root, outputs the structure of an optimal binary search tree. For the

example in Figure 15.10, your procedure should print out the structure

404 Chapter 15 Dynamic Programming

k 2 is the root

k 1 is the left child of k 2

d 0 is the left child of k 1

d 1 is the right child of k 1

k 5 is the right child of k 2

k 4 is the left child of k 5

k 3 is the left child of k 4

d 2 is the left child of k 3

d 3 is the right child of k 3

d 4 is the right child of k 4

d 5 is the right child of k 5

corresponding to the optimal binary search tree shown in Figure 15.9(b).

15.5-2

Determine the cost and structure of an optimal binary search tree for a set of n D 7

keys with the following probabilities:

i 0 1 2 3 4 5 6 7

pi 0.04 0.06 0.08 0.02 0.10 0.12 0.14

qi 0.06 0.06 0.06 0.06 0.05 0.05 0.05 0.05

15.5-3

Suppose that instead of maintaining the table wŒi; j , we computed the value

of w.i; j / directly from equation (15.12) in line 9 of OPTIMAL-BST and used this

computed value in line 11. How would this change affect the asymptotic running

time of OPTIMAL-BST?

15.5-4 ?

Knuth [212] has shown that there are always roots of optimal subtrees such that

rootŒi; j 1 rootŒi; j rootŒi C 1; j for all 1 i < j n. Use this fact to

modify the OPTIMAL-BST procedure to run in ‚.n 2 / time.

Problems

15-1 Longest simple path in a directed acyclic graph

Suppose that we are given a directed acyclic graph G D .V; E/ with real-

valued edge weights and two distinguished vertices s and t. Describe a dynamic-

programming approach for ﬁnding a longest weighted simple path from s to t.

What does the subproblem graph look like? What is the efﬁciency of your algo-

rithm?

Problems for Chapter 15 405

(a) (b)

Figure 15.11 Seven points in the plane, shown on a unit grid. (a) The shortest closed tour, with

length approximately 24:89. This tour is not bitonic. (b) The shortest bitonic tour for the same set of

points. Its length is approximately 25:58.

15-2 Longest palindrome subsequence

A palindrome is a nonempty string over some alphabet that reads the same for-

ward and backward. Examples of palindromes are all strings of length 1, civic,

racecar, and aibohphobia (fear of palindromes).

Give an efﬁcient algorithm to ﬁnd the longest palindrome that is a subsequence

of a given input string. For example, given the input character, your algorithm

should return carac. What is the running time of your algorithm?

15-3 Bitonic euclidean traveling-salesman problem

In the euclidean traveling-salesman problem, we are given a set of n points in

the plane, and we wish to ﬁnd the shortest closed tour that connects all n points.

Figure 15.11(a) shows the solution to a 7-point problem. The general problem is

NP-hard, and its solution is therefore believed to require more than polynomial

time (see Chapter 34).

J. L. Bentley has suggested that we simplify the problem by restricting our at-

tention to bitonic tours, that is, tours that start at the leftmost point, go strictly

rightward to the rightmost point, and then go strictly leftward back to the starting

point. Figure 15.11(b) shows the shortest bitonic tour of the same 7 points. In this

case, a polynomial-time algorithm is possible.

Describe an O.n 2 /-time algorithm for determining an optimal bitonic tour. You

may assume that no two points have the same x-coordinate and that all operations

on real numbers take unit time. (Hint: Scan left to right, maintaining optimal pos-

sibilities for the two parts of the tour.)

15-4 Printing neatly

Consider the problem of neatly printing a paragraph with a monospaced font (all

characters having the same width) on a printer. The input text is a sequence of n

406 Chapter 15 Dynamic Programming

words of lengths l 1 ; l 2 ; : : : ; l n , measured in characters. We want to print this para-

graph neatly on a number of lines that hold a maximum of M characters each. Our

criterion of “neatness” is as follows. If a given line contains words i through j ,

where i j , and we leave exactly one space between words, the number of extra

space characters at the end of the line is M j C i

P j

kDi

l k , which must be

nonnegative so that the words ﬁt on the line. We wish to minimize the sum, over

all lines except the last, of the cubes of the numbers of extra space characters at the

ends of lines. Give a dynamic-programming algorithm to print a paragraph of n

words neatly on a printer. Analyze the running time and space requirements of

your algorithm.

15-5 Edit distance

In order to transform one source string of text xŒ1 : : m to a target string yŒ1 : : n,

we can perform various transformation operations. Our goal is, given x and y,

to produce a series of transformations that change x to y. We use an ar-

ray ´—assumed to be large enough to hold all the characters it will need—to hold

the intermediate results. Initially, ´ is empty, and at termination, we should have

´Œj D yŒj for j D 1; 2; : : : ; n. We maintain current indices i into x and j into ´,

and the operations are allowed to alter ´ and these indices. Initially, i D j D 1.

We are required to examine every character in x during the transformation, which

means that at the end of the sequence of transformation operations, we must have

i D m C 1.

We may choose from among six transformation operations:

Copy a character from x to ´ by setting ´Œj D xŒi and then incrementing both i

and j . This operation examines xŒi.

Replace a character from x by another character c, by setting ´Œj D c, and then

incrementing both i and j . This operation examines xŒi.

Delete a character from x by incrementing i but leaving j alone. This operation

examines xŒi.

Insert the character c into ´ by setting ´Œj D c and then incrementing j , but

leaving i alone. This operation examines no characters of x.

Twiddle (i.e., exchange) the next two characters by copying them from x to ´ but

in the opposite order; we do so by setting ´Œj D xŒi C 1 and ´Œj C 1 D xŒi

and then setting i D i C 2 and j D j C 2. This operation examines xŒi

and xŒi C 1.

Kill the remainder of x by setting i D m C 1. This operation examines all char-

acters in x that have not yet been examined. This operation, if performed, must

be the ﬁnal operation.

Problems for Chapter 15 407

As an example, one way to transform the source string algorithm to the target

string altruistic is to use the following sequence of operations, where the

underlined characters are xŒi and ´Œj after the operation:

Operation x ´

initial strings algorithm

copy algorithm a

copy algorithm al

replace by t algorithm alt

delete algorithm alt

copy algorithm altr

insert u algorithm altru

insert i algorithm altrui

insert s algorithm altruis

twiddle algorithm altruisti

insert c algorithm altruistic

kill algorithm altruistic

Note that there are several other sequences of transformation operations that trans-

form algorithm to altruistic.

Each of the transformation operations has an associated cost. The cost of an

operation depends on the speciﬁc application, but we assume that each operation’s

cost is a constant that is known to us. We also assume that the individual costs of

the copy and replace operations are less than the combined costs of the delete and

insert operations; otherwise, the copy and replace operations would not be used.

The cost of a given sequence of transformation operations is the sum of the costs

of the individual operations in the sequence. For the sequence above, the cost of

transforming algorithm to altruistic is

.3 cost.copy// C cost.replace/ C cost.delete/ C .4 cost.insert//

C cost.twiddle/ C cost.kill/ :

a. Given two sequences xŒ1 : : m and yŒ1 : : n and set of transformation-operation

costs, the edit distance from x to y is the cost of the least expensive operation

sequence that transforms x to y. Describe a dynamic-programming algorithm

that ﬁnds the edit distance from xŒ1 : : m to yŒ1 : : n and prints an optimal op-

eration sequence. Analyze the running time and space requirements of your

algorithm.

The edit-distance problem generalizes the problem of aligning two DNA sequences

(see, for example, Setubal and Meidanis [310, Section 3.2]). There are several

methods for measuring the similarity of two DNA sequences by aligning them.

One such method to align two sequences x and y consists of inserting spaces at

408 Chapter 15 Dynamic Programming

arbitrary locations in the two sequences (including at either end) so that the result-

ing sequences x 0

and y 0

have the same length but do not have a space in the same

position (i.e., for no position j are both x 0 Œj and y 0 Œj a space). Then we assign a

“score” to each position. Position j receives a score as follows:

C1 if x 0 Œj D y 0 Œj and neither is a space,

1 if x 0 Œj ¤ y 0 Œj and neither is a space,

2 if either x 0 Œj or y 0 Œj is a space.

The score for the alignment is the sum of the scores of the individual positions. For

example, given the sequences x D GATCGGCAT and y D CAATGTGAATC, one

alignment is

G ATCG GCAT

CAAT GTGAATC

-\*++\*+\*+-++\*

A + under a position indicates a score of C1 for that position, a - indicates a score

of 1, and a \* indicates a score of 2, so that this alignment has a total score of

6 1 2 1 4 2 D 4.

b. Explain how to cast the problem of ﬁnding an optimal alignment as an edit

distance problem using a subset of the transformation operations copy, replace,

delete, insert, twiddle, and kill.

15-6 Planning a company party

Professor Stewart is consulting for the president of a corporation that is planning

a company party. The company has a hierarchical structure; that is, the supervisor

relation forms a tree rooted at the president. The personnel ofﬁce has ranked each

employee with a conviviality rating, which is a real number. In order to make the

party fun for all attendees, the president does not want both an employee and his

or her immediate supervisor to attend.

Professor Stewart is given the tree that describes the structure of the corporation,

using the left-child, right-sibling representation described in Section 10.4. Each

node of the tree holds, in addition to the pointers, the name of an employee and

that employee’s conviviality ranking. Describe an algorithm to make up a guest

list that maximizes the sum of the conviviality ratings of the guests. Analyze the

running time of your algorithm.

15-7 Viterbi algorithm

We can use dynamic programming on a directed graph G D .V; E/ for speech

recognition. Each edge .u; / 2 E is labeled with a sound .u; / from a ﬁ-

nite set † of sounds. The labeled graph is a formal model of a person speaking

Problems for Chapter 15 409

a restricted language. Each path in the graph starting from a distinguished ver-

tex 0 2 V corresponds to a possible sequence of sounds produced by the model.

We deﬁne the label of a directed path to be the concatenation of the labels of the

edges on that path.

a. Describe an efﬁcient algorithm that, given an edge-labeled graph G with dis-

tinguished vertex 0 and a sequence s D h 1 ; 2 ; : : : ; k i of sounds from †,

returns a path in G that begins at 0 and has s as its label, if any such path exists.

Otherwise, the algorithm should return NO-SUCH-PATH. Analyze the running

time of your algorithm. (Hint: You may ﬁnd concepts from Chapter 22 useful.)

Now, suppose that every edge .u; / 2 E has an associated nonnegative proba-

bility p.u; / of traversing the edge .u; / from vertex u and thus producing the

corresponding sound. The sum of the probabilities of the edges leaving any vertex

equals 1. The probability of a path is deﬁned to be the product of the probabil-

ities of its edges. We can view the probability of a path beginning at 0 as the

probability that a “random walk” beginning at 0 will follow the speciﬁed path,

where we randomly choose which edge to take leaving a vertex u according to the

probabilities of the available edges leaving u.

b. Extend your answer to part (a) so that if a path is returned, it is a most prob-

able path starting at 0 and having label s. Analyze the running time of your

algorithm.

15-8 Image compression by seam carving

We are given a color picture consisting of an m n array AŒ1 : : m; 1 : : n of pixels,

where each pixel speciﬁes a triple of red, green, and blue (RGB) intensities. Sup-

pose that we wish to compress this picture slightly. Speciﬁcally, we wish to remove

one pixel from each of the m rows, so that the whole picture becomes one pixel

narrower. To avoid disturbing visual effects, however, we require that the pixels

removed in two adjacent rows be in the same or adjacent columns; the pixels re-

moved form a “seam” from the top row to the bottom row where successive pixels

in the seam are adjacent vertically or diagonally.

a. Show that the number of such possible seams grows at least exponentially in m,

assuming that n > 1.

b. Suppose now that along with each pixel AŒi; j , we have calculated a real-

valued disruption measure dŒi; j , indicating how disruptive it would be to

remove pixel AŒi; j . Intuitively, the lower a pixel’s disruption measure, the

more similar the pixel is to its neighbors. Suppose further that we deﬁne the

disruption measure of a seam to be the sum of the disruption measures of its

pixels.

410 Chapter 15 Dynamic Programming

Give an algorithm to ﬁnd a seam with the lowest disruption measure. How

efﬁcient is your algorithm?

15-9 Breaking a string

A certain string-processing language allows a programmer to break a string into

two pieces. Because this operation copies the string, it costs n time units to break

a string of n characters into two pieces. Suppose a programmer wants to break

a string into many pieces. The order in which the breaks occur can affect the

total amount of time used. For example, suppose that the programmer wants to

break a 20-character string after characters 2, 8, and 10 (numbering the characters

in ascending order from the left-hand end, starting from 1). If she programs the

breaks to occur in left-to-right order, then the ﬁrst break costs 20 time units, the

second break costs 18 time units (breaking the string from characters 3 to 20 at

character 8), and the third break costs 12 time units, totaling 50 time units. If she

programs the breaks to occur in right-to-left order, however, then the ﬁrst break

costs 20 time units, the second break costs 10 time units, and the third break costs

8 time units, totaling 38 time units. In yet another order, she could break ﬁrst at 8

(costing 20), then break the left piece at 2 (costing 8), and ﬁnally the right piece

at 10 (costing 12), for a total cost of 40.

Design an algorithm that, given the numbers of characters after which to break,

determines a least-cost way to sequence those breaks. More formally, given a

string S with n characters and an array LŒ1 : : m containing the break points, com-

pute the lowest cost for a sequence of breaks, along with a sequence of breaks that

achieves this cost.

15-10 Planning an investment strategy

Your knowledge of algorithms helps you obtain an exciting job with the Acme

Computer Company, along with a $10,000 signing bonus. You decide to invest

this money with the goal of maximizing your return at the end of 10 years. You

decide to use the Amalgamated Investment Company to manage your investments.

Amalgamated Investments requires you to observe the following rules. It offers n

different investments, numbered 1 through n. In each year j , investment i provides

a return rate of r ij . In other words, if you invest d dollars in investment i in year j ,

then at the end of year j , you have dr ij dollars. The return rates are guaranteed,

that is, you are given all the return rates for the next 10 years for each investment.

You make investment decisions only once per year. At the end of each year, you

can leave the money made in the previous year in the same investments, or you

can shift money to other investments, by either shifting money between existing

investments or moving money to a new investement. If you do not move your

money between two consecutive years, you pay a fee of f 1 dollars, whereas if you

switch your money, you pay a fee of f 2 dollars, where f 2 > f 1 .

Problems for Chapter 15 411

a. The problem, as stated, allows you to invest your money in multiple investments

in each year. Prove that there exists an optimal investment strategy that, in

each year, puts all the money into a single investment. (Recall that an optimal

investment strategy maximizes the amount of money after 10 years and is not

concerned with any other objectives, such as minimizing risk.)

b. Prove that the problem of planning your optimal investment strategy exhibits

optimal substructure.

c. Design an algorithm that plans your optimal investment strategy. What is the

running time of your algorithm?

d. Suppose that Amalgamated Investments imposed the additional restriction that,

at any point, you can have no more than $15,000 in any one investment. Show

that the problem of maximizing your income at the end of 10 years no longer

exhibits optimal substructure.

15-11 Inventory planning

The Rinky Dink Company makes machines that resurface ice rinks. The demand

for such products varies from month to month, and so the company needs to de-

velop a strategy to plan its manufacturing given the ﬂuctuating, but predictable,

demand. The company wishes to design a plan for the next n months. For each

month i, the company knows the demand d i , that is, the number of machines that

it will sell. Let D D

P n

iD1

d i be the total demand over the next n months. The

company keeps a full-time staff who provide labor to manufacture up to m ma-

chines per month. If the company needs to make more than m machines in a given

month, it can hire additional, part-time labor, at a cost that works out to c dollars

per machine. Furthermore, if, at the end of a month, the company is holding any

unsold machines, it must pay inventory costs. The cost for holding j machines is

given as a function h.j / for j D 1; 2; : : : ; D, where h.j / 0 for 1 j D and

h.j / h.j C 1/ for 1 j D 1.

Give an algorithm that calculates a plan for the company that minimizes its costs

while fulﬁlling all the demand. The running time should be polyomial in n and D.

15-12 Signing free-agent baseball players

Suppose that you are the general manager for a major-league baseball team. During

the off-season, you need to sign some free-agent players for your team. The team

owner has given you a budget of $X to spend on free agents. You are allowed to

spend less than $X altogether, but the owner will ﬁre you if you spend any more

than $X.

412 Chapter 15 Dynamic Programming

You are considering N different positions, and for each position, P free-agent

players who play that position are available. 8 Because you do not want to overload

your roster with too many players at any position, for each position you may sign

at most one free agent who plays that position. (If you do not sign any players at a

particular position, then you plan to stick with the players you already have at that

position.)

To determine how valuable a player is going to be, you decide to use a sabermet-

ric statistic 9 known as “VORP,” or “value over replacement player.” A player with

a higher VORP is more valuable than a player with a lower VORP. A player with a

higher VORP is not necessarily more expensive to sign than a player with a lower

VORP, because factors other than a player’s value determine how much it costs to

sign him.

For each available free-agent player, you have three pieces of information:

the player’s position,

the amount of money it will cost to sign the player, and

the player’s VORP.

Devise an algorithm that maximizes the total VORP of the players you sign while

spending no more than $X altogether. You may assume that each player signs for a

multiple of $100,000. Your algorithm should output the total VORP of the players

you sign, the total amount of money you spend, and a list of which players you

sign. Analyze the running time and space requirement of your algorithm.

Chapter notes

R. Bellman began the systematic study of dynamic programming in 1955. The

word “programming,” both here and in linear programming, refers to using a tab-

ular solution method. Although optimization techniques incorporating elements of

dynamic programming were known earlier, Bellman provided the area with a solid

mathematical basis [37].

8

Although there are nine positions on a baseball team, N is not necesarily equal to 9 because some

general managers have particular ways of thinking about positions. For example, a general manager

might consider right-handed pitchers and left-handed pitchers to be separate “positions,” as well as

starting pitchers, long relief pitchers (relief pitchers who can pitch several innings), and short relief

pitchers (relief pitchers who normally pitch at most only one inning).

9

Sabermetrics is the application of statistical analysis to baseball records. It provides several ways

to compare the relative values of individual players.

Notes for Chapter 15 413

Galil and Park [125] classify dynamic-programming algorithms according to the

size of the table and the number of other table entries each entry depends on. They

call a dynamic-programming algorithm tD=eD if its table size is O.n t / and each

entry depends on O.n e / other entries. For example, the matrix-chain multiplication

algorithm in Section 15.2 would be 2D=1D, and the longest-common-subsequence

algorithm in Section 15.4 would be 2D=0D.

Hu and Shing [182, 183] give an O.n lg n/-time algorithm for the matrix-chain

multiplication problem.

The O.mn/-time algorithm for the longest-common-subsequence problem ap-

pears to be a folk algorithm. Knuth [70] posed the question of whether subquadratic

algorithms for the LCS problem exist. Masek and Paterson [244] answered this

question in the afﬁrmative by giving an algorithm that runs in O.mn= lg n/ time,

where n m and the sequences are drawn from a set of bounded size. For the

special case in which no element appears more than once in an input sequence,

Szymanski [326] shows how to solve the problem in O..n C m/ lg.n C m// time.

Many of these results extend to the problem of computing string edit distances

(Problem 15-5).

An early paper on variable-length binary encodings by Gilbert and Moore [133]

had applications to constructing optimal binary search trees for the case in which all

probabilities p i are 0; this paper contains an O.n 3 /-time algorithm. Aho, Hopcroft,

and Ullman [5] present the algorithm from Section 15.5. Exercise 15.5-4 is due to

Knuth [212]. Hu and Tucker [184] devised an algorithm for the case in which all

probabilities p i are 0 that uses O.n 2 / time and O.n/ space; subsequently, Knuth

[211] reduced the time to O.n lg n/.

Problem 15-8 is due to Avidan and Shamir [27], who have posted on the Web a

wonderful video illustrating this image-compression technique.

16 Greedy Algorithms

Algorithms for optimization problems typically go through a sequence of steps,

with a set of choices at each step. For many optimization problems, using dynamic

programming to determine the best choices is overkill; simpler, more efﬁcient al-

gorithms will do. A greedy algorithm always makes the choice that looks best at

the moment. That is, it makes a locally optimal choice in the hope that this choice

will lead to a globally optimal solution. This chapter explores optimization prob-

lems for which greedy algorithms provide optimal solutions. Before reading this

chapter, you should read about dynamic programming in Chapter 15, particularly

Section 15.3.

Greedy algorithms do not always yield optimal solutions, but for many problems

they do. We shall ﬁrst examine, in Section 16.1, a simple but nontrivial problem,

the activity-selection problem, for which a greedy algorithm efﬁciently computes

an optimal solution. We shall arrive at the greedy algorithm by ﬁrst consider-

ing a dynamic-programming approach and then showing that we can always make

greedy choices to arrive at an optimal solution. Section 16.2 reviews the basic

elements of the greedy approach, giving a direct approach for proving greedy al-

gorithms correct. Section 16.3 presents an important application of greedy tech-

niques: designing data-compression (Huffman) codes. In Section 16.4, we inves-

tigate some of the theory underlying combinatorial structures called “matroids,”

for which a greedy algorithm always produces an optimal solution. Finally, Sec-

tion 16.5 applies matroids to solve a problem of scheduling unit-time tasks with

deadlines and penalties.

The greedy method is quite powerful and works well for a wide range of prob-

lems. Later chapters will present many algorithms that we can view as applica-

tions of the greedy method, including minimum-spanning-tree algorithms (Chap-

ter 23), Dijkstra’s algorithm for shortest paths from a single source (Chapter 24),

and Chv´atal’s greedy set-covering heuristic (Chapter 35). Minimum-spanning-tree

algorithms furnish a classic example of the greedy method. Although you can read

16.1 An activity-selection problem 415

this chapter and Chapter 23 independently of each other, you might ﬁnd it useful

to read them together.

16.1 An activity-selection problem

Our ﬁrst example is the problem of scheduling several competing activities that re-

quire exclusive use of a common resource, with a goal of selecting a maximum-size

set of mutually compatible activities. Suppose we have a set S D fa 1 ; a 2 ; : : : ; a n g

of n proposed activities that wish to use a resource, such as a lecture hall, which

can serve only one activity at a time. Each activity a i has a start time s i and a ﬁnish

time f i , where 0 s i < f i < 1. If selected, activity a i takes place during the

half-open time interval Œs i ; f i /. Activities a i and a j are compatible if the intervals

Œs i ; f i / and Œs j ; f j / do not overlap. That is, a i and a j are compatible if s i f j

or s j f i . In the activity-selection problem, we wish to select a maximum-size

subset of mutually compatible activities. We assume that the activities are sorted

in monotonically increasing order of ﬁnish time:

f 1 f 2 f 3 f n1 f n : (16.1)

(We shall see later the advantage that this assumption provides.) For example,

consider the following set S of activities:

i 1 2 3 4 5 6 7 8 9 10 11

si 1 3 0 5 3 5 6 8 8 2 12

fi 4 5 6 7 9 9 10 11 12 14 16

For this example, the subset fa 3 ; a 9 ; a 11 g consists of mutually compatible activities.

It is not a maximum subset, however, since the subset fa 1 ; a 4 ; a 8 ; a 11 g is larger. In

fact, fa 1 ; a 4 ; a 8 ; a 11 g is a largest subset of mutually compatible activities; another

largest subset is fa 2 ; a 4 ; a 9 ; a 11 g.

We shall solve this problem in several steps. We start by thinking about a

dynamic-programming solution, in which we consider several choices when deter-

mining which subproblems to use in an optimal solution. We shall then observe that

we need to consider only one choice—the greedy choice—and that when we make

the greedy choice, only one subproblem remains. Based on these observations, we

shall develop a recursive greedy algorithm to solve the activity-scheduling prob-

lem. We shall complete the process of developing a greedy solution by converting

the recursive algorithm to an iterative one. Although the steps we shall go through

in this section are slightly more involved than is typical when developing a greedy

algorithm, they illustrate the relationship between greedy algorithms and dynamic

programming.

416 Chapter 16 Greedy Algorithms

The optimal substructure of the activity-selection problem

We can easily verify that the activity-selection problem exhibits optimal substruc-

ture. Let us denote by S ij the set of activities that start after activity a i ﬁnishes and

that ﬁnish before activity a j starts. Suppose that we wish to ﬁnd a maximum set of

mutually compatible activities in S ij , and suppose further that such a maximum set

is A ij , which includes some activity a k . By including a k in an optimal solution, we

are left with two subproblems: ﬁnding mutually compatible activities in the set S ik

(activities that start after activity a i ﬁnishes and that ﬁnish before activity a k starts)

and ﬁnding mutually compatible activities in the set S kj (activities that start after

activity a k ﬁnishes and that ﬁnish before activity a j starts). Let A ik D A ij \ S ik

and A kj D A ij \ S kj , so that A ik contains the activities in A ij that ﬁnish before a k

starts and A kj contains the activities in A ij that start after a k ﬁnishes. Thus, we

have A ij D A ik [ fa k g [ A kj , and so the maximum-size set A ij of mutually com-

patible activities in S ij consists of jA ij j D jA ik j C jA kj j C 1 activities.

The usual cut-and-paste argument shows that the optimal solution A ij must also

include optimal solutions to the two subproblems for S ik and S kj . If we could

ﬁnd a set A 0

kj

of mutually compatible activities in S kj where jA 0

kj

j > jA kj j, then

we could use A 0

kj

, rather than A kj , in a solution to the subproblem for S ij . We

would have constructed a set of jA ik j C jA 0

kj

j C 1 > jA ik j C jA kj j C 1 D jA ij j

mutually compatible activities, which contradicts the assumption that A ij is an

optimal solution. A symmetric argument applies to the activities in S ik .

This way of characterizing optimal substructure suggests that we might solve

the activity-selection problem by dynamic programming. If we denote the size of

an optimal solution for the set S ij by cŒi; j , then we would have the recurrence

cŒi; j D cŒi; k C cŒk; j C 1 :

Of course, if we did not know that an optimal solution for the set S ij includes

activity a k , we would have to examine all activities in S ij to ﬁnd which one to

choose, so that

cŒi; j D

(

0 if S ij D ; ;

max

a k 2S ij

fcŒi; k C cŒk; j C 1g if S ij ¤ ; :

(16.2)

We could then develop a recursive algorithm and memoize it, or we could work

bottom-up and ﬁll in table entries as we go along. But we would be overlooking

another important characteristic of the activity-selection problem that we can use

to great advantage.

16.1 An activity-selection problem 417

Making the greedy choice

What if we could choose an activity to add to our optimal solution without having

to ﬁrst solve all the subproblems? That could save us from having to consider all

the choices inherent in recurrence (16.2). In fact, for the activity-selection problem,

we need consider only one choice: the greedy choice.

What do we mean by the greedy choice for the activity-selection problem? Intu-

ition suggests that we should choose an activity that leaves the resource available

for as many other activities as possible. Now, of the activities we end up choos-

ing, one of them must be the ﬁrst one to ﬁnish. Our intuition tells us, therefore,

to choose the activity in S with the earliest ﬁnish time, since that would leave the

resource available for as many of the activities that follow it as possible. (If more

than one activity in S has the earliest ﬁnish time, then we can choose any such

activity.) In other words, since the activities are sorted in monotonically increasing

order by ﬁnish time, the greedy choice is activity a 1 . Choosing the ﬁrst activity

to ﬁnish is not the only way to think of making a greedy choice for this problem;

Exercise 16.1-3 asks you to explore other possibilities.

If we make the greedy choice, we have only one remaining subproblem to solve:

ﬁnding activities that start after a 1 ﬁnishes. Why don’t we have to consider ac-

tivities that ﬁnish before a 1 starts? We have that s 1 < f 1 , and f 1 is the earliest

ﬁnish time of any activity, and therefore no activity can have a ﬁnish time less than

or equal to s 1 . Thus, all activities that are compatible with activity a 1 must start

after a 1 ﬁnishes.

Furthermore, we have already established that the activity-selection problem ex-

hibits optimal substructure. Let S k D fa i 2 S W s i f k g be the set of activities that

start after activity a k ﬁnishes. If we make the greedy choice of activity a 1 , then S 1

remains as the only subproblem to solve. 1 Optimal substructure tells us that if a 1

is in the optimal solution, then an optimal solution to the original problem consists

of activity a 1 and all the activities in an optimal solution to the subproblem S 1 .

One big question remains: is our intuition correct? Is the greedy choice—in

which we choose the ﬁrst activity to ﬁnish—always part of some optimal solution?

The following theorem shows that it is.

1

We sometimes refer to the sets Sk as subproblems rather than as just sets of activities. It will always

be clear from the context whether we are referring to Sk as a set of activities or as a subproblem

whose input is that set.

418 Chapter 16 Greedy Algorithms

Theorem 16.1

Consider any nonempty subproblem S k , and let a m be an activity in S k with the

earliest ﬁnish time. Then a m is included in some maximum-size subset of mutually

compatible activities of S k .

Proof Let A k be a maximum-size subset of mutually compatible activities in S k ,

and let a j be the activity in A k with the earliest ﬁnish time. If a j D a m , we are

done, since we have shown that a m is in some maximum-size subset of mutually

compatible activities of S k . If a j ¤ a m , let the set A 0

k

D A k fa j g [ fa m g be A k

but substituting a m for a j . The activities in A 0

k

are disjoint, which follows because

the activities in A k are disjoint, a j is the ﬁrst activity in A k to ﬁnish, and f m f j .

Since jA 0

k

j D jA k j, we conclude that A 0

k

is a maximum-size subset of mutually

compatible activities of S k , and it includes a m .

Thus, we see that although we might be able to solve the activity-selection prob-

lem with dynamic programming, we don’t need to. (Besides, we have not yet

examined whether the activity-selection problem even has overlapping subprob-

lems.) Instead, we can repeatedly choose the activity that ﬁnishes ﬁrst, keep only

the activities compatible with this activity, and repeat until no activities remain.

Moreover, because we always choose the activity with the earliest ﬁnish time, the

ﬁnish times of the activities we choose must strictly increase. We can consider

each activity just once overall, in monotonically increasing order of ﬁnish times.

An algorithm to solve the activity-selection problem does not need to work

bottom-up, like a table-based dynamic-programming algorithm. Instead, it can

work top-down, choosing an activity to put into the optimal solution and then solv-

ing the subproblem of choosing activities from those that are compatible with those

already chosen. Greedy algorithms typically have this top-down design: make a

choice and then solve a subproblem, rather than the bottom-up technique of solving

subproblems before making a choice.

A recursive greedy algorithm

Now that we have seen how to bypass the dynamic-programming approach and in-

stead use a top-down, greedy algorithm, we can write a straightforward, recursive

procedure to solve the activity-selection problem. The procedure RECURSIVE-

ACTIVITY-SELECTOR takes the start and ﬁnish times of the activities, represented

as arrays s and f , 2 the index k that deﬁnes the subproblem S k it is to solve, and

2

Because the pseudocode takes s and f as arrays, it indexes into them with square brackets rather

than subscripts.

16.1 An activity-selection problem 419

the size n of the original problem. It returns a maximum-size set of mutually com-

patible activities in S k . We assume that the n input activities are already ordered

by monotonically increasing ﬁnish time, according to equation (16.1). If not, we

can sort them into this order in O.n lg n/ time, breaking ties arbitrarily. In order

to start, we add the ﬁctitious activity a 0 with f 0 D 0, so that subproblem S 0 is

the entire set of activities S. The initial call, which solves the entire problem, is

RECURSIVE-ACTIVITY-SELECTOR.s; f; 0; n/.

RECURSIVE-ACTIVITY-SELECTOR.s; f; k; n/

1 m D k C 1

2 while m n and sŒm < f Œk // ﬁnd the ﬁrst activity in S k to ﬁnish

3 m D m C 1

4 if m n

5 return fa m g [ RECURSIVE-ACTIVITY-SELECTOR.s; f; m; n/

6 else return ;

Figure 16.1 shows the operation of the algorithm. In a given recursive call

RECURSIVE-ACTIVITY-SELECTOR.s; f; k; n/, the while loop of lines 2–3 looks

for the ﬁrst activity in S k to ﬁnish. The loop examines a kC1 ; a kC2 ; : : : ; a n , un-

til it ﬁnds the ﬁrst activity a m that is compatible with a k ; such an activity has

s m f k . If the loop terminates because it ﬁnds such an activity, line 5 returns

the union of fa m g and the maximum-size subset of S m returned by the recursive

call RECURSIVE-ACTIVITY-SELECTOR.s; f; m; n/. Alternatively, the loop may

terminate because m > n, in which case we have examined all activities in S k

without ﬁnding one that is compatible with a k . In this case, S k D ;, and so the

procedure returns ; in line 6.

Assuming that the activities have already been sorted by ﬁnish times, the running

time of the call RECURSIVE-ACTIVITY-SELECTOR.s; f; 0; n/ is ‚.n/, which we

can see as follows. Over all recursive calls, each activity is examined exactly once

in the while loop test of line 2. In particular, activity a i is examined in the last call

made in which k < i.

An iterative greedy algorithm

We easily can convert our recursive procedure to an iterative one. The procedure

RECURSIVE-ACTIVITY-SELECTOR is almost “tail recursive” (see Problem 7-4):

it ends with a recursive call to itself followed by a union operation. It is usually a

straightforward task to transform a tail-recursive procedure to an iterative form; in

fact, some compilers for certain programming languages perform this task automat-

ically. As written, RECURSIVE-ACTIVITY-SELECTOR works for subproblems S k ,

i.e., subproblems that consist of the last activities to ﬁnish.

420 Chapter 16 Greedy Algorithms

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

time

2 3 5

3 0 6

4 5 7

5 3 9

6 5 9

7 6 10

8 8 11

9 8 12

10 2 14

11 12 16

1 1 4

k s

k

f

k

a

1

a

2

a

1

a

3

a

1

a

4

a

1

a

4

a

5

a

1

a

4

a

6

a

1

a

4

a

7

a

1

a

4

a

8

a

1

a

4

a

8

a

9

a

1

a

4

a

8

a

10

a

1

a

4

a

8

a

11

a

1

a

4

a

8

a

11

0 – 0

a

1

a

0

a

0

RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, 11)

RECURSIVE-ACTIVITY-SELECTOR(s, f, 1, 11)

RECURSIVE-ACTIVITY-SELECTOR(s, f, 4, 11)

RECURSIVE-ACTIVITY-SELECTOR(s, f, 8, 11)

m = 1

m = 4

m = 8

m = 11

RECURSIVE-ACTIVITY-SELECTOR(s, f, 11, 11)

15 16

Figure 16.1 The operation of RECURSIVE-ACTIVITY-SELECTOR on the 11 activities given ear-

lier. Activities considered in each recursive call appear between horizontal lines. The ﬁctitious

activity a0 ﬁnishes at time 0, and the initial call RECURSIVE-ACTIVITY-SELECTOR.s; f; 0; 11/, se-

lects activity a1. In each recursive call, the activities that have already been selected are shaded,

and the activity shown in white is being considered. If the starting time of an activity occurs before

the ﬁnish time of the most recently added activity (the arrow between them points left), it is re-

jected. Otherwise (the arrow points directly up or to the right), it is selected. The last recursive call,

RECURSIVE-ACTIVITY-SELECTOR.s; f; 11; 11/, returns ;. The resulting set of selected activities is

fa1; a4; a8; a11g.

16.1 An activity-selection problem 421

The procedure GREEDY-ACTIVITY-SELECTOR is an iterative version of the pro-

cedure RECURSIVE-ACTIVITY-SELECTOR. It also assumes that the input activi-

ties are ordered by monotonically increasing ﬁnish time. It collects selected activ-

ities into a set A and returns this set when it is done.

GREEDY-ACTIVITY-SELECTOR.s; f /

1 n D s:length

2 A D fa 1 g

3 k D 1

4 for m D 2 to n

5 if sŒm f Œk

6 A D A [ fa m g

7 k D m

8 return A

The procedure works as follows. The variable k indexes the most recent addition

to A, corresponding to the activity a k in the recursive version. Since we consider

the activities in order of monotonically increasing ﬁnish time, f k is always the

maximum ﬁnish time of any activity in A. That is,

f k D max ff i W a i 2 Ag : (16.3)

Lines 2–3 select activity a 1 , initialize A to contain just this activity, and initialize k

to index this activity. The for loop of lines 4–7 ﬁnds the earliest activity in S k to

ﬁnish. The loop considers each activity a m in turn and adds a m to A if it is compat-

ible with all previously selected activities; such an activity is the earliest in S k to

ﬁnish. To see whether activity a m is compatible with every activity currently in A,

it sufﬁces by equation (16.3) to check (in line 5) that its start time s m is not earlier

than the ﬁnish time f k of the activity most recently added to A. If activity a m is

compatible, then lines 6–7 add activity a m to A and set k to m. The set A returned

by the call GREEDY-ACTIVITY-SELECTOR.s; f / is precisely the set returned by

the call RECURSIVE-ACTIVITY-SELECTOR.s; f; 0; n/.

Like the recursive version, GREEDY-ACTIVITY-SELECTOR schedules a set of n

activities in ‚.n/ time, assuming that the activities were already sorted initially by

their ﬁnish times.

Exercises

16.1-1

Give a dynamic-programming algorithm for the activity-selection problem, based

on recurrence (16.2). Have your algorithm compute the sizes cŒi; j as deﬁned

above and also produce the maximum-size subset of mutually compatible activities.

422 Chapter 16 Greedy Algorithms

Assume that the inputs have been sorted as in equation (16.1). Compare the running

time of your solution to the running time of GREEDY-ACTIVITY-SELECTOR.

16.1-2

Suppose that instead of always selecting the ﬁrst activity to ﬁnish, we instead select

the last activity to start that is compatible with all previously selected activities. De-

scribe how this approach is a greedy algorithm, and prove that it yields an optimal

solution.

16.1-3

Not just any greedy approach to the activity-selection problem produces a max-

imum-size set of mutually compatible activities. Give an example to show that

the approach of selecting the activity of least duration from among those that are

compatible with previously selected activities does not work. Do the same for

the approaches of always selecting the compatible activity that overlaps the fewest

other remaining activities and always selecting the compatible remaining activity

with the earliest start time.

16.1-4

Suppose that we have a set of activities to schedule among a large number of lecture

halls, where any activity can take place in any lecture hall. We wish to schedule

all the activities using as few lecture halls as possible. Give an efﬁcient greedy

algorithm to determine which activity should use which lecture hall.

(This problem is also known as the interval-graph coloring problem. We can

create an interval graph whose vertices are the given activities and whose edges

connect incompatible activities. The smallest number of colors required to color

every vertex so that no two adjacent vertices have the same color corresponds to

ﬁnding the fewest lecture halls needed to schedule all of the given activities.)

16.1-5

Consider a modiﬁcation to the activity-selection problem in which each activity a i

has, in addition to a start and ﬁnish time, a value i . The objective is no longer

to maximize the number of activities scheduled, but instead to maximize the total

value of the activities scheduled. That is, we wish to choose a set A of compatible

activities such that

P

a k 2A

k is maximized. Give a polynomial-time algorithm for

this problem.

16.2 Elements of the greedy strategy 423

16.2 Elements of the greedy strategy

A greedy algorithm obtains an optimal solution to a problem by making a sequence

of choices. At each decision point, the algorithm makes choice that seems best at

the moment. This heuristic strategy does not always produce an optimal solution,

but as we saw in the activity-selection problem, sometimes it does. This section

discusses some of the general properties of greedy methods.

The process that we followed in Section 16.1 to develop a greedy algorithm was

a bit more involved than is typical. We went through the following steps:

1. Determine the optimal substructure of the problem.

2. Develop a recursive solution. (For the activity-selection problem, we formu-

lated recurrence (16.2), but we bypassed developing a recursive algorithm based

on this recurrence.)

3. Show that if we make the greedy choice, then only one subproblem remains.

4. Prove that it is always safe to make the greedy choice. (Steps 3 and 4 can occur

in either order.)

5. Develop a recursive algorithm that implements the greedy strategy.

6. Convert the recursive algorithm to an iterative algorithm.

In going through these steps, we saw in great detail the dynamic-programming un-

derpinnings of a greedy algorithm. For example, in the activity-selection problem,

we ﬁrst deﬁned the subproblems S ij , where both i and j varied. We then found

that if we always made the greedy choice, we could restrict the subproblems to be

of the form S k .

Alternatively, we could have fashioned our optimal substructure with a greedy

choice in mind, so that the choice leaves just one subproblem to solve. In the

activity-selection problem, we could have started by dropping the second subscript

and deﬁning subproblems of the form S k . Then, we could have proven that a greedy

choice (the ﬁrst activity a m to ﬁnish in S k ), combined with an optimal solution to

the remaining set S m of compatible activities, yields an optimal solution to S k .

More generally, we design greedy algorithms according to the following sequence

of steps:

1. Cast the optimization problem as one in which we make a choice and are left

with one subproblem to solve.

2. Prove that there is always an optimal solution to the original problem that makes

the greedy choice, so that the greedy choice is always safe.

424 Chapter 16 Greedy Algorithms

3. Demonstrate optimal substructure by showing that, having made the greedy

choice, what remains is a subproblem with the property that if we combine an

optimal solution to the subproblem with the greedy choice we have made, we

arrive at an optimal solution to the original problem.

We shall use this more direct process in later sections of this chapter. Neverthe-

less, beneath every greedy algorithm, there is almost always a more cumbersome

dynamic-programming solution.

How can we tell whether a greedy algorithm will solve a particular optimization

problem? No way works all the time, but the greedy-choice property and optimal

substructure are the two key ingredients. If we can demonstrate that the problem

has these properties, then we are well on the way to developing a greedy algorithm

for it.

Greedy-choice property

The ﬁrst key ingredient is the greedy-choice property: we can assemble a globally

optimal solution by making locally optimal (greedy) choices. In other words, when

we are considering which choice to make, we make the choice that looks best in

the current problem, without considering results from subproblems.

Here is where greedy algorithms differ from dynamic programming. In dynamic

programming, we make a choice at each step, but the choice usually depends on the

solutions to subproblems. Consequently, we typically solve dynamic-programming

problems in a bottom-up manner, progressing from smaller subproblems to larger

subproblems. (Alternatively, we can solve them top down, but memoizing. Of

course, even though the code works top down, we still must solve the subprob-

lems before making a choice.) In a greedy algorithm, we make whatever choice

seems best at the moment and then solve the subproblem that remains. The choice

made by a greedy algorithm may depend on choices so far, but it cannot depend on

any future choices or on the solutions to subproblems. Thus, unlike dynamic pro-

gramming, which solves the subproblems before making the ﬁrst choice, a greedy

algorithm makes its ﬁrst choice before solving any subproblems. A dynamic-

programming algorithm proceeds bottom up, whereas a greedy strategy usually

progresses in a top-down fashion, making one greedy choice after another, reduc-

ing each given problem instance to a smaller one.

Of course, we must prove that a greedy choice at each step yields a globally

optimal solution. Typically, as in the case of Theorem 16.1, the proof examines

a globally optimal solution to some subproblem. It then shows how to modify

the solution to substitute the greedy choice for some other choice, resulting in one

similar, but smaller, subproblem.

We can usually make the greedy choice more efﬁciently than when we have to

consider a wider set of choices. For example, in the activity-selection problem, as-

16.2 Elements of the greedy strategy 425

suming that we had already sorted the activities in monotonically increasing order

of ﬁnish times, we needed to examine each activity just once. By preprocessing the

input or by using an appropriate data structure (often a priority queue), we often

can make greedy choices quickly, thus yielding an efﬁcient algorithm.

Optimal substructure

A problem exhibits optimal substructure if an optimal solution to the problem

contains within it optimal solutions to subproblems. This property is a key in-

gredient of assessing the applicability of dynamic programming as well as greedy

algorithms. As an example of optimal substructure, recall how we demonstrated in

Section 16.1 that if an optimal solution to subproblem S ij includes an activity a k ,

then it must also contain optimal solutions to the subproblems S ik and S kj . Given

this optimal substructure, we argued that if we knew which activity to use as a k , we

could construct an optimal solution to S ij by selecting a k along with all activities

in optimal solutions to the subproblems S ik and S kj . Based on this observation of

optimal substructure, we were able to devise the recurrence (16.2) that described

the value of an optimal solution.

We usually use a more direct approach regarding optimal substructure when

applying it to greedy algorithms. As mentioned above, we have the luxury of

assuming that we arrived at a subproblem by having made the greedy choice in

the original problem. All we really need to do is argue that an optimal solution to

the subproblem, combined with the greedy choice already made, yields an optimal

solution to the original problem. This scheme implicitly uses induction on the

subproblems to prove that making the greedy choice at every step produces an

optimal solution.

Greedy versus dynamic programming

Because both the greedy and dynamic-programming strategies exploit optimal sub-

structure, you might be tempted to generate a dynamic-programming solution to a

problem when a greedy solution sufﬁces or, conversely, you might mistakenly think

that a greedy solution works when in fact a dynamic-programming solution is re-

quired. To illustrate the subtleties between the two techniques, let us investigate

two variants of a classical optimization problem.

The 0-1 knapsack problem is the following. A thief robbing a store ﬁnds n

items. The ith item is worth i dollars and weighs w i pounds, where i and w i are

integers. The thief wants to take as valuable a load as possible, but he can carry at

most W pounds in his knapsack, for some integer W . Which items should he take?

(We call this the 0-1 knapsack problem because for each item, the thief must either

426 Chapter 16 Greedy Algorithms

take it or leave it behind; he cannot take a fractional amount of an item or take an

item more than once.)

In the fractional knapsack problem, the setup is the same, but the thief can take

fractions of items, rather than having to make a binary (0-1) choice for each item.

You can think of an item in the 0-1 knapsack problem as being like a gold ingot

and an item in the fractional knapsack problem as more like gold dust.

Both knapsack problems exhibit the optimal-substructure property. For the 0-1

problem, consider the most valuable load that weighs at most W pounds. If we

remove item j from this load, the remaining load must be the most valuable load

weighing at most W w j that the thief can take from the n 1 original items

excluding j . For the comparable fractional problem, consider that if we remove

a weight w of one item j from the optimal load, the remaining load must be the

most valuable load weighing at most W w that the thief can take from the n 1

original items plus w j w pounds of item j .

Although the problems are similar, we can solve the fractional knapsack problem

by a greedy strategy, but we cannot solve the 0-1 problem by such a strategy. To

solve the fractional problem, we ﬁrst compute the value per pound i =w i for each

item. Obeying a greedy strategy, the thief begins by taking as much as possible of

the item with the greatest value per pound. If the supply of that item is exhausted

and he can still carry more, he takes as much as possible of the item with the next

greatest value per pound, and so forth, until he reaches his weight limit W . Thus,

by sorting the items by value per pound, the greedy algorithm runs in O.n lg n/

time. We leave the proof that the fractional knapsack problem has the greedy-

choice property as Exercise 16.2-1.

To see that this greedy strategy does not work for the 0-1 knapsack problem,

consider the problem instance illustrated in Figure 16.2(a). This example has 3

items and a knapsack that can hold 50 pounds. Item 1 weighs 10 pounds and

is worth 60 dollars. Item 2 weighs 20 pounds and is worth 100 dollars. Item 3

weighs 30 pounds and is worth 120 dollars. Thus, the value per pound of item 1 is

6 dollars per pound, which is greater than the value per pound of either item 2 (5

dollars per pound) or item 3 (4 dollars per pound). The greedy strategy, therefore,

would take item 1 ﬁrst. As you can see from the case analysis in Figure 16.2(b),

however, the optimal solution takes items 2 and 3, leaving item 1 behind. The two

possible solutions that take item 1 are both suboptimal.

For the comparable fractional problem, however, the greedy strategy, which

takes item 1 ﬁrst, does yield an optimal solution, as shown in Figure 16.2(c). Tak-

ing item 1 doesn’t work in the 0-1 problem because the thief is unable to ﬁll his

knapsack to capacity, and the empty space lowers the effective value per pound of

his load. In the 0-1 problem, when we consider whether to include an item in the

knapsack, we must compare the solution to the subproblem that includes the item

with the solution to the subproblem that excludes the item before we can make the

16.2 Elements of the greedy strategy 427

10

$60

item 1

20

$100

item 2

30

$120

item 3

50

knapsack

(a)

+

$120

$100

= $220

+

$60

$100

= $160

+

$60

$120

= $180

(b)

+

$60

$100

= $240

$80

+

(c)

20

30

10

20

10

30

10

20

20

30

Figure 16.2 An example showing that the greedy strategy does not work for the 0-1 knapsack

problem. (a) The thief must select a subset of the three items shown whose weight must not exceed

50 pounds. (b) The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal,

even though item 1 has the greatest value per pound. (c) For the fractional knapsack problem, taking

the items in order of greatest value per pound yields an optimal solution.

choice. The problem formulated in this way gives rise to many overlapping sub-

problems—a hallmark of dynamic programming, and indeed, as Exercise 16.2-2

asks you to show, we can use dynamic programming to solve the 0-1 problem.

Exercises

16.2-1

Prove that the fractional knapsack problem has the greedy-choice property.

16.2-2

Give a dynamic-programming solution to the 0-1 knapsack problem that runs in

O.n W / time, where n is the number of items and W is the maximum weight of

items that the thief can put in his knapsack.

16.2-3

Suppose that in a 0-1 knapsack problem, the order of the items when sorted by

increasing weight is the same as their order when sorted by decreasing value. Give

an efﬁcient algorithm to ﬁnd an optimal solution to this variant of the knapsack

problem, and argue that your algorithm is correct.

16.2-4

Professor Gekko has always dreamed of inline skating across North Dakota. He

plans to cross the state on highway U.S. 2, which runs from Grand Forks, on the

eastern border with Minnesota, to Williston, near the western border with Montana.

428 Chapter 16 Greedy Algorithms

The professor can carry two liters of water, and he can skate m miles before running

out of water. (Because North Dakota is relatively ﬂat, the professor does not have

to worry about drinking water at a greater rate on uphill sections than on ﬂat or

downhill sections.) The professor will start in Grand Forks with two full liters of

water. His ofﬁcial North Dakota state map shows all the places along U.S. 2 at

which he can reﬁll his water and the distances between these locations.

The professor’s goal is to minimize the number of water stops along his route

across the state. Give an efﬁcient method by which he can determine which water

stops he should make. Prove that your strategy yields an optimal solution, and give

its running time.

16.2-5

Describe an efﬁcient algorithm that, given a set fx 1 ; x 2 ; : : : ; x n g of points on the

real line, determines the smallest set of unit-length closed intervals that contains

all of the given points. Argue that your algorithm is correct.

16.2-6 ?

Show how to solve the fractional knapsack problem in O.n/ time.

16.2-7

Suppose you are given two sets A and B, each containing n positive integers. You

can choose to reorder each set however you like. After reordering, let a i be the ith

element of set A, and let b i be the ith element of set B. You then receive a payoff

of

Q n

iD1

a i

b i . Give an algorithm that will maximize your payoff. Prove that your

algorithm maximizes the payoff, and state its running time.

16.3 Huffman codes

Huffman codes compress data very effectively: savings of 20% to 90% are typical,

depending on the characteristics of the data being compressed. We consider the

data to be a sequence of characters. Huffman’s greedy algorithm uses a table giving

how often each character occurs (i.e., its frequency) to build up an optimal way of

representing each character as a binary string.

Suppose we have a 100,000-character data ﬁle that we wish to store compactly.

We observe that the characters in the ﬁle occur with the frequencies given by Fig-

ure 16.3. That is, only 6 different characters appear, and the character a occurs

45,000 times.

We have many options for how to represent such a ﬁle of information. Here,

we consider the problem of designing a binary character code (or code for short)

16.3 Huffman codes 429

a b c d e f

Frequency (in thousands) 45 13 12 16 9 5

Fixed-length codeword 000 001 010 011 100 101

Variable-length codeword 0 101 100 111 1101 1100

Figure 16.3 A character-coding problem. A data ﬁle of 100,000 characters contains only the char-

acters a–f, with the frequencies indicated. If we assign each character a 3-bit codeword, we can

encode the ﬁle in 300,000 bits. Using the variable-length code shown, we can encode the ﬁle in only

224,000 bits.

in which each character is represented by a unique binary string, which we call a

codeword. If we use a ﬁxed-length code, we need 3 bits to represent 6 characters:

a = 000, b = 001, . . . , f = 101. This method requires 300,000 bits to code the

entire ﬁle. Can we do better?

A variable-length code can do considerably better than a ﬁxed-length code, by

giving frequent characters short codewords and infrequent characters long code-

words. Figure 16.3 shows such a code; here the 1-bit string 0 represents a, and the

4-bit string 1100 represents f. This code requires

.45 1 C 13 3 C 12 3 C 16 3 C 9 4 C 5 4/ 1,000 D 224,000 bits

to represent the ﬁle, a savings of approximately 25%. In fact, this is an optimal

character code for this ﬁle, as we shall see.

Preﬁx codes

We consider here only codes in which no codeword is also a preﬁx of some other

codeword. Such codes are called preﬁx codes. 3 Although we won’t prove it here, a

preﬁx code can always achieve the optimal data compression among any character

code, and so we suffer no loss of generality by restricting our attention to preﬁx

codes.

Encoding is always simple for any binary character code; we just concatenate the

codewords representing each character of the ﬁle. For example, with the variable-

length preﬁx code of Figure 16.3, we code the 3-character ﬁle abc as 0101100 D

0101100, where “” denotes concatenation.

Preﬁx codes are desirable because they simplify decoding. Since no codeword

is a preﬁx of any other, the codeword that begins an encoded ﬁle is unambiguous.

We can simply identify the initial codeword, translate it back to the original char-

3

Perhaps “preﬁx-free codes” would be a better name, but the term “preﬁx codes” is standard in the

literature.

430 Chapter 16 Greedy Algorithms

a:45 b:13 c:12 d:16 e:9 f:5

58 28 14

86 14

100

0 1 0 1 0 1

0 1 0

0 1

e:9 f:5

14

0 1

c:12 b:13

25

0 1

d:16

30

0 1

55

0 1

a:45

100

0 1

(a) (b)

Figure 16.4 Trees corresponding to the coding schemes in Figure 16.3. Each leaf is labeled with

a character and its frequency of occurrence. Each internal node is labeled with the sum of the fre-

quencies of the leaves in its subtree. (a) The tree corresponding to the ﬁxed-length code a = 000, . . . ,

f = 101. (b) The tree corresponding to the optimal preﬁx code a = 0, b = 101, . . . , f = 1100.

acter, and repeat the decoding process on the remainder of the encoded ﬁle. In our

example, the string 001011101 parses uniquely as 0 0 101 1101, which decodes

to aabe.

The decoding process needs a convenient representation for the preﬁx code so

that we can easily pick off the initial codeword. A binary tree whose leaves are

the given characters provides one such representation. We interpret the binary

codeword for a character as the simple path from the root to that character, where 0

means “go to the left child” and 1 means “go to the right child.” Figure 16.4 shows

the trees for the two codes of our example. Note that these are not binary search

trees, since the leaves need not appear in sorted order and internal nodes do not

contain character keys.

An optimal code for a ﬁle is always represented by a full binary tree, in which

every nonleaf node has two children (see Exercise 16.3-2). The ﬁxed-length code

in our example is not optimal since its tree, shown in Figure 16.4(a), is not a full bi-

nary tree: it contains codewords beginning 10. . . , but none beginning 11. . . . Since

we can now restrict our attention to full binary trees, we can say that if C is the

alphabet from which the characters are drawn and all character frequencies are pos-

itive, then the tree for an optimal preﬁx code has exactly jC j leaves, one for each

letter of the alphabet, and exactly jC j 1 internal nodes (see Exercise B.5-3).

Given a tree T corresponding to a preﬁx code, we can easily compute the number

of bits required to encode a ﬁle. For each character c in the alphabet C , let the

attribute c:freq denote the frequency of c in the ﬁle and let d T .c/ denote the depth

16.3 Huffman codes 431

of c’s leaf in the tree. Note that d T .c/ is also the length of the codeword for

character c. The number of bits required to encode a ﬁle is thus

B.T / D

X

c2C

c:freq d T .c/ ; (16.4)

which we deﬁne as the cost of the tree T .

Constructing a Huffman code

Huffman invented a greedy algorithm that constructs an optimal preﬁx code called

a Huffman code. In line with our observations in Section 16.2, its proof of cor-

rectness relies on the greedy-choice property and optimal substructure. Rather

than demonstrating that these properties hold and then developing pseudocode, we

present the pseudocode ﬁrst. Doing so will help clarify how the algorithm makes

greedy choices.

In the pseudocode that follows, we assume that C is a set of n characters and

that each character c 2 C is an object with an attribute c:freq giving its frequency.

The algorithm builds the tree T corresponding to the optimal code in a bottom-up

manner. It begins with a set of jC j leaves and performs a sequence of jC j 1

“merging” operations to create the ﬁnal tree. The algorithm uses a min-priority

queue Q, keyed on the freq attribute, to identify the two least-frequent objects to

merge together. When we merge two objects, the result is a new object whose

frequency is the sum of the frequencies of the two objects that were merged.

HUFFMAN.C /

1 n D jC j

2 Q D C

3 for i D 1 to n 1

4 allocate a new node ´

5 ´:left D x D EXTRACT-MIN.Q/

6 ´:right D y D EXTRACT-MIN.Q/

7 ´:freq D x:freq C y:freq

8 INSERT.Q; ´/

9 return EXTRACT-MIN.Q/ // return the root of the tree

For our example, Huffman’s algorithm proceeds as shown in Figure 16.5. Since

the alphabet contains 6 letters, the initial queue size is n D 6, and 5 merge steps

build the tree. The ﬁnal tree represents the optimal preﬁx code. The codeword for

a letter is the sequence of edge labels on the simple path from the root to the letter.

Line 2 initializes the min-priority queue Q with the characters in C . The for

loop in lines 3–8 repeatedly extracts the two nodes x and y of lowest frequency

432 Chapter 16 Greedy Algorithms

e:9 f:5

14

0 1

c:12 b:13

25

0 1

d:16

30

0 1

55

0 1

a:45

100

0 1

e:9 f:5

14

0 1

c:12 b:13

25

0 1

d:16

30

0 1

55

0 1

a:45

e:9 f:5

14

0 1

c:12 b:13

25

0 1

d:16

30

0 1

a:45

e:9 f:5

14

0 1

c:12 b:13

25

0 1

d:16 a:45

e:9 f:5

14

0 1

c:12 b:13 d:16 a:45 e:9 f:5 c:12 b:13 d:16 a:45 (a)

(c)

(e)

(b)

(d)

(f)

Figure 16.5 The steps of Huffman’s algorithm for the frequencies given in Figure 16.3. Each part

shows the contents of the queue sorted into increasing order by frequency. At each step, the two

trees with lowest frequencies are merged. Leaves are shown as rectangles containing a character

and its frequency. Internal nodes are shown as circles containing the sum of the frequencies of their

children. An edge connecting an internal node with its children is labeled 0 if it is an edge to a left

child and 1 if it is an edge to a right child. The codeword for a letter is the sequence of labels on the

edges connecting the root to the leaf for that letter. (a) The initial set of n D 6 nodes, one for each

letter. (b)–(e) Intermediate stages. (f) The ﬁnal tree.

from the queue, replacing them in the queue with a new node ´ representing their

merger. The frequency of ´ is computed as the sum of the frequencies of x and y

in line 7. The node ´ has x as its left child and y as its right child. (This order is

arbitrary; switching the left and right child of any node yields a different code of

the same cost.) After n 1 mergers, line 9 returns the one node left in the queue,

which is the root of the code tree.

Although the algorithm would produce the same result if we were to excise the

variables x and y—assigning directly to ´:left and ´:right in lines 5 and 6, and

changing line 7 to ´:freq D ´:left:freq C ´:right:freq—we shall use the node

16.3 Huffman codes 433

names x and y in the proof of correctness. Therefore, we ﬁnd it convenient to

leave them in.

To analyze the running time of Huffman’s algorithm, we assume that Q is im-

plemented as a binary min-heap (see Chapter 6). For a set C of n characters, we

can initialize Q in line 2 in O.n/ time using the BUILD-MIN-HEAP procedure dis-

cussed in Section 6.3. The for loop in lines 3–8 executes exactly n 1 times, and

since each heap operation requires time O.lg n/, the loop contributes O.n lg n/ to

the running time. Thus, the total running time of HUFFMAN on a set of n charac-

ters is O.n lg n/. We can reduce the running time to O.n lg lg n/ by replacing the

binary min-heap with a van Emde Boas tree (see Chapter 20).

Correctness of Huffman’s algorithm

To prove that the greedy algorithm HUFFMAN is correct, we show that the prob-

lem of determining an optimal preﬁx code exhibits the greedy-choice and optimal-

substructure properties. The next lemma shows that the greedy-choice property

holds.

Lemma 16.2

Let C be an alphabet in which each character c 2 C has frequency c:freq. Let

x and y be two characters in C having the lowest frequencies. Then there exists

an optimal preﬁx code for C in which the codewords for x and y have the same

length and differ only in the last bit.

Proof The idea of the proof is to take the tree T representing an arbitrary optimal

preﬁx code and modify it to make a tree representing another optimal preﬁx code

such that the characters x and y appear as sibling leaves of maximum depth in the

new tree. If we can construct such a tree, then the codewords for x and y will have

the same length and differ only in the last bit.

Let a and b be two characters that are sibling leaves of maximum depth in T .

Without loss of generality, we assume that a:freq b:freq and x:freq y:freq.

Since x:freq and y:freq are the two lowest leaf frequencies, in order, and a:freq

and b:freq are two arbitrary frequencies, in order, we have x:freq a:freq and

y:freq b:freq.

In the remainder of the proof, it is possible that we could have x:freq D a:freq

or y:freq D b:freq. However, if we had x:freq D b:freq, then we would also have

a:freq D b:freq D x:freq D y:freq (see Exercise 16.3-1), and the lemma would

be trivially true. Thus, we will assume that x:freq ¤ b:freq, which means that

x ¤ b.

As Figure 16.6 shows, we exchange the positions in T of a and x to produce a

tree T 0

, and then we exchange the positions in T 0

of b and y to produce a tree T 00

434 Chapter 16 Greedy Algorithms

x

y

a b x

y

a

b x y

a

b

T

′′

T T

′

Figure 16.6 An illustration of the key step in the proof of Lemma 16.2. In the optimal tree T ,

leaves a and b are two siblings of maximum depth. Leaves x and y are the two characters with the

lowest frequencies; they appear in arbitrary positions in T . Assuming that x ¤ b, swapping leaves a

and x produces tree T

0

, and then swapping leaves b and y produces tree T

00

. Since each swap does

not increase the cost, the resulting tree T

00

is also an optimal tree.

in which x and y are sibling leaves of maximum depth. (Note that if x D b but

y ¤ a, then tree T 00

does not have x and y as sibling leaves of maximum depth.

Because we assume that x ¤ b, this situation cannot occur.) By equation (16.4),

the difference in cost between T and T 0

is

B.T / B.T

0

/

D

X

c2C

c:freq d T .c/

X

c2C

c:freq d T 0 .c/

D x:freq d T .x/ C a:freq d T .a/ x:freq d T 0 .x/ a:freq d T 0 .a/

D x:freq d T .x/ C a:freq d T .a/ x:freq d T .a/ a:freq d T .x/

D .a:freq x:freq/.d T .a/ d T .x//

0 ;

because both a:freq x:freq and d T .a/ d T .x/ are nonnegative. More speciﬁ-

cally, a:freq x:freq is nonnegative because x is a minimum-frequency leaf, and

d T .a/d T .x/ is nonnegative because a is a leaf of maximum depth in T . Similarly,

exchanging y and b does not increase the cost, and so B.T 0 / B.T 00 / is nonnega-

tive. Therefore, B.T 00 / B.T /, and since T is optimal, we have B.T / B.T 00 /,

which implies B.T 00 / D B.T /. Thus, T 00

is an optimal tree in which x and y

appear as sibling leaves of maximum depth, from which the lemma follows.

Lemma 16.2 implies that the process of building up an optimal tree by mergers

can, without loss of generality, begin with the greedy choice of merging together

those two characters of lowest frequency. Why is this a greedy choice? We can

view the cost of a single merger as being the sum of the frequencies of the two items

being merged. Exercise 16.3-4 shows that the total cost of the tree constructed

equals the sum of the costs of its mergers. Of all possible mergers at each step,

HUFFMAN chooses the one that incurs the least cost.

16.3 Huffman codes 435

The next lemma shows that the problem of constructing optimal preﬁx codes has

the optimal-substructure property.

Lemma 16.3

Let C be a given alphabet with frequency c:freq deﬁned for each character c 2 C .

Let x and y be two characters in C with minimum frequency. Let C 0

be the

alphabet C with the characters x and y removed and a new character ´ added,

so that C 0 D C fx; yg [ f´g. Deﬁne f for C 0

as for C , except that

´:freq D x:freq C y:freq. Let T 0

be any tree representing an optimal preﬁx code

for the alphabet C 0

. Then the tree T , obtained from T 0

by replacing the leaf node

for ´ with an internal node having x and y as children, represents an optimal preﬁx

code for the alphabet C .

Proof We ﬁrst show how to express the cost B.T / of tree T in terms of the

cost B.T 0 / of tree T 0

, by considering the component costs in equation (16.4).

For each character c 2 C fx; yg, we have that d T .c/ D d T 0 .c/, and hence

c:freq d T .c/ D c:freq d T 0 .c/. Since d T .x/ D d T .y/ D d T 0 .´/ C 1, we have

x:freq d T .x/ C y:freq d T .y/ D .x:freq C y:freq/.d T 0 .´/ C 1/

D ´:freq d T 0 .´/ C .x:freq C y:freq/ ;

from which we conclude that

B.T / D B.T

0

/ C x:freq C y:freq

or, equivalently,

B.T

0

/ D B.T / x:freq y:freq :

We now prove the lemma by contradiction. Suppose that T does not repre-

sent an optimal preﬁx code for C . Then there exists an optimal tree T 00

such that

B.T 00 / < B.T /. Without loss of generality (by Lemma 16.2), T 00

has x and y as

siblings. Let T 000

be the tree T 00

with the common parent of x and y replaced by a

leaf ´ with frequency ´:freq D x:freq C y:freq. Then

B.T

000

/ D B.T

00

/ x:freq y:freq

< B.T / x:freq y:freq

D B.T

0

/ ;

yielding a contradiction to the assumption that T 0

represents an optimal preﬁx code

for C 0

. Thus, T must represent an optimal preﬁx code for the alphabet C .

Theorem 16.4

Procedure HUFFMAN produces an optimal preﬁx code.

Proof Immediate from Lemmas 16.2 and 16.3.

436 Chapter 16 Greedy Algorithms

Exercises

16.3-1

Explain why, in the proof of Lemma 16.2, if x:freq D b:freq, then we must have

a:freq D b:freq D x:freq D y:freq.

16.3-2

Prove that a binary tree that is not full cannot correspond to an optimal preﬁx code.

16.3-3

What is an optimal Huffman code for the following set of frequencies, based on

the ﬁrst 8 Fibonacci numbers?

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

Can you generalize your answer to ﬁnd the optimal code when the frequencies are

the ﬁrst n Fibonacci numbers?

16.3-4

Prove that we can also express the total cost of a tree for a code as the sum, over

all internal nodes, of the combined frequencies of the two children of the node.

16.3-5

Prove that if we order the characters in an alphabet so that their frequencies

are monotonically decreasing, then there exists an optimal code whose codeword

lengths are monotonically increasing.

16.3-6

Suppose we have an optimal preﬁx code on a set C D f0; 1; : : : ; n 1g of charac-

ters and we wish to transmit this code using as few bits as possible. Show how to

represent any optimal preﬁx code on C using only 2n 1 C n dlg ne bits. (Hint:

Use 2n 1 bits to specify the structure of the tree, as discovered by a walk of the

tree.)

16.3-7

Generalize Huffman’s algorithm to ternary codewords (i.e., codewords using the

symbols 0, 1, and 2), and prove that it yields optimal ternary codes.

16.3-8

Suppose that a data ﬁle contains a sequence of 8-bit characters such that all 256

characters are about equally common: the maximum character frequency is less

than twice the minimum character frequency. Prove that Huffman coding in this

case is no more efﬁcient than using an ordinary 8-bit ﬁxed-length code.

16.4 Matroids and greedy methods 437

16.3-9

Show that no compression scheme can expect to compress a ﬁle of randomly cho-

sen 8-bit characters by even a single bit. (Hint: Compare the number of possible

ﬁles with the number of possible encoded ﬁles.)

? 16.4 Matroids and greedy methods

In this section, we sketch a beautiful theory about greedy algorithms. This theory

describes many situations in which the greedy method yields optimal solutions. It

involves combinatorial structures known as “matroids.” Although this theory does

not cover all cases for which a greedy method applies (for example, it does not

cover the activity-selection problem of Section 16.1 or the Huffman-coding prob-

lem of Section 16.3), it does cover many cases of practical interest. Furthermore,

this theory has been extended to cover many applications; see the notes at the end

of this chapter for references.

Matroids

A matroid is an ordered pair M D .S; / satisfying the following conditions.

1. S is a ﬁnite set.

2. is a nonempty family of subsets of S, called the independent subsets of S,

such that if B 2 and A B, then A 2 . We say that is hereditary if it

satisﬁes this property. Note that the empty set ; is necessarily a member of .

3. If A 2 , B 2 , and jAj < jBj, then there exists some element x 2 B A

such that A [ fxg 2 . We say that M satisﬁes the exchange property.

The word “matroid” is due to Hassler Whitney. He was studying matric ma-

troids, in which the elements of S are the rows of a given matrix and a set of rows is

independent if they are linearly independent in the usual sense. As Exercise 16.4-2

asks you to show, this structure deﬁnes a matroid.

As another example of matroids, consider the graphic matroid M G D .S G ; G /

deﬁned in terms of a given undirected graph G D .V; E/ as follows:

The set S G is deﬁned to be E, the set of edges of G.

If A is a subset of E, then A 2 G if and only if A is acyclic. That is, a set of

edges A is independent if and only if the subgraph G A D .V; A/ forms a forest.

The graphic matroid M G is closely related to the minimum-spanning-tree problem,

which Chapter 23 covers in detail.

438 Chapter 16 Greedy Algorithms

Theorem 16.5

If G D .V; E/ is an undirected graph, then M G D .S G ; G / is a matroid.

Proof Clearly, S G D E is a ﬁnite set. Furthermore, G is hereditary, since a

subset of a forest is a forest. Putting it another way, removing edges from an

acyclic set of edges cannot create cycles.

Thus, it remains to show that M G satisﬁes the exchange property. Suppose that

G A D .V; A/ and G B D .V; B/ are forests of G and that jBj > jAj. That is, A

and B are acyclic sets of edges, and B contains more edges than A does.

We claim that a forest F D .V F ; E F / contains exactly jV F j jE F j trees. To

see why, suppose that F consists of t trees, where the ith tree contains i vertices

and e i edges. Then, we have

jE F j D

t X

iD1

e i

D

t X

iD1

. i 1/ (by Theorem B.2)

D

t X

iD1

i t

D jV F j t ;

which implies that t D jV F j jE F j. Thus, forest G A contains jV j jAj trees, and

forest G B contains jV j jBj trees.

Since forest G B has fewer trees than forest G A does, forest G B must contain

some tree T whose vertices are in two different trees in forest G A . Moreover,

since T is connected, it must contain an edge .u; / such that vertices u and

are in different trees in forest G A . Since the edge .u; / connects vertices in two

different trees in forest G A , we can add the edge .u; / to forest G A without creating

a cycle. Therefore, M G satisﬁes the exchange property, completing the proof that

M G is a matroid.

Given a matroid M D .S; /, we call an element x … A an extension of A 2

if we can add x to A while preserving independence; that is, x is an extension

of A if A [ fxg 2 . As an example, consider a graphic matroid M G . If A is an

independent set of edges, then edge e is an extension of A if and only if e is not

in A and the addition of e to A does not create a cycle.

If A is an independent subset in a matroid M , we say that A is maximal if it has

no extensions. That is, A is maximal if it is not contained in any larger independent

subset of M . The following property is often useful.

16.4 Matroids and greedy methods 439

Theorem 16.6

All maximal independent subsets in a matroid have the same size.

Proof Suppose to the contrary that A is a maximal independent subset of M

and there exists another larger maximal independent subset B of M . Then, the

exchange property implies that for some x 2 B A, we can extend A to a larger

independent set A [ fxg, contradicting the assumption that A is maximal.

As an illustration of this theorem, consider a graphic matroid M G for a con-

nected, undirected graph G. Every maximal independent subset of M G must be a

free tree with exactly jV j 1 edges that connects all the vertices of G. Such a tree

is called a spanning tree of G.

We say that a matroid M D .S; / is weighted if it is associated with a weight

function w that assigns a strictly positive weight w.x/ to each element x 2 S. The

weight function w extends to subsets of S by summation:

w.A/ D

X

x2A

w.x/

for any A S. For example, if we let w.e/ denote the weight of an edge e in a

graphic matroid M G , then w.A/ is the total weight of the edges in edge set A.

Greedy algorithms on a weighted matroid

Many problems for which a greedy approach provides optimal solutions can be for-

mulated in terms of ﬁnding a maximum-weight independent subset in a weighted

matroid. That is, we are given a weighted matroid M D .S; /, and we wish to

ﬁnd an independent set A 2 such that w.A/ is maximized. We call such a sub-

set that is independent and has maximum possible weight an optimal subset of the

matroid. Because the weight w.x/ of any element x 2 S is positive, an optimal

subset is always a maximal independent subset—it always helps to make A as large

as possible.

For example, in the minimum-spanning-tree problem, we are given a connected

undirected graph G D .V; E/ and a length function w such that w.e/ is the (posi-

tive) length of edge e. (We use the term “length” here to refer to the original edge

weights for the graph, reserving the term “weight” to refer to the weights in the

associated matroid.) We wish to ﬁnd a subset of the edges that connects all of

the vertices together and has minimum total length. To view this as a problem of

ﬁnding an optimal subset of a matroid, consider the weighted matroid M G with

weight function w 0

, where w 0 .e/ D w 0 w.e/ and w 0 is larger than the maximum

length of any edge. In this weighted matroid, all weights are positive and an opti-

mal subset is a spanning tree of minimum total length in the original graph. More

speciﬁcally, each maximal independent subset A corresponds to a spanning tree

440 Chapter 16 Greedy Algorithms

with jV j 1 edges, and since

w

0

.A/ D

X

e2A

w

0

.e/

D

X

e2A

.w 0 w.e//

D .jV j 1/w 0

X

e2A

w.e/

D .jV j 1/w 0 w.A/

for any maximal independent subset A, an independent subset that maximizes the

quantity w 0 .A/ must minimize w.A/. Thus, any algorithm that can ﬁnd an optimal

subset A in an arbitrary matroid can solve the minimum-spanning-tree problem.

Chapter 23 gives algorithms for the minimum-spanning-tree problem, but here

we give a greedy algorithm that works for any weighted matroid. The algorithm

takes as input a weighted matroid M D .S; / with an associated positive weight

function w, and it returns an optimal subset A. In our pseudocode, we denote the

components of M by M:S and M: and the weight function by w. The algorithm

is greedy because it considers in turn each element x 2 S, in order of monotoni-

cally decreasing weight, and immediately adds it to the set A being accumulated if

A [ fxg is independent.

GREEDY.M; w/

1 A D ;

2 sort M:S into monotonically decreasing order by weight w

3 for each x 2 M:S, taken in monotonically decreasing order by weight w.x/

4 if A [ fxg 2 M:

5 A D A [ fxg

6 return A

Line 4 checks whether adding each element x to A would maintain A as an inde-

pendent set. If A would remain independent, then line 5 adds x to A. Otherwise, x

is discarded. Since the empty set is independent, and since each iteration of the for

loop maintains A’s independence, the subset A is always independent, by induc-

tion. Therefore, GREEDY always returns an independent subset A. We shall see in

a moment that A is a subset of maximum possible weight, so that A is an optimal

subset.

The running time of GREEDY is easy to analyze. Let n denote jSj. The sorting

phase of GREEDY takes time O.n lg n/. Line 4 executes exactly n times, once for

each element of S. Each execution of line 4 requires a check on whether or not

the set A [ fxg is independent. If each such check takes time O.f .n//, the entire

algorithm runs in time O.n lg n C nf .n//.

16.4 Matroids and greedy methods 441

We now prove that GREEDY returns an optimal subset.

Lemma 16.7 (Matroids exhibit the greedy-choice property)

Suppose that M D .S; / is a weighted matroid with weight function w and that S

is sorted into monotonically decreasing order by weight. Let x be the ﬁrst element

of S such that fxg is independent, if any such x exists. If x exists, then there exists

an optimal subset A of S that contains x.

Proof If no such x exists, then the only independent subset is the empty set and

the lemma is vacuously true. Otherwise, let B be any nonempty optimal subset.

Assume that x … B; otherwise, letting A D B gives an optimal subset of S that

contains x.

No element of B has weight greater than w.x/. To see why, observe that y 2 B

implies that fyg is independent, since B 2 and is hereditary. Our choice of x

therefore ensures that w.x/ w.y/ for any y 2 B.

Construct the set A as follows. Begin with A D fxg. By the choice of x, set A is

independent. Using the exchange property, repeatedly ﬁnd a new element of B that

we can add to A until jAj D jBj, while preserving the independence of A. At that

point, A and B are the same except that A has x and B has some other element y.

That is, A D B fyg [ fxg for some y 2 B, and so

w.A/ D w.B/ w.y/ C w.x/

w.B/ :

Because set B is optimal, set A, which contains x, must also be optimal.

We next show that if an element is not an option initially, then it cannot be an

option later.

Lemma 16.8

Let M D .S; / be any matroid. If x is an element of S that is an extension of

some independent subset A of S, then x is also an extension of ;.

Proof Since x is an extension of A, we have that A[fxg is independent. Since

is hereditary, fxg must be independent. Thus, x is an extension of ;.

Corollary 16.9

Let M D .S; / be any matroid. If x is an element of S such that x is not an

extension of ;, then x is not an extension of any independent subset A of S.

Proof This corollary is simply the contrapositive of Lemma 16.8.

442 Chapter 16 Greedy Algorithms

Corollary 16.9 says that any element that cannot be used immediately can never

be used. Therefore, GREEDY cannot make an error by passing over any initial

elements in S that are not an extension of ;, since they can never be used.

Lemma 16.10 (Matroids exhibit the optimal-substructure property)

Let x be the ﬁrst element of S chosen by GREEDY for the weighted matroid

M D .S; /. The remaining problem of ﬁnding a maximum-weight indepen-

dent subset containing x reduces to ﬁnding a maximum-weight independent subset

of the weighted matroid M 0 D .S 0 ; 0 /, where

S

0

D fy 2 S W fx; yg 2 g ;

0

D fB S fxg W B [ fxg 2 g ;

and the weight function for M 0

is the weight function for M , restricted to S 0

. (We

call M 0

the contraction of M by the element x.)

Proof If A is any maximum-weight independent subset of M containing x, then

A 0 D A fxg is an independent subset of M 0

. Conversely, any independent sub-

set A 0

of M 0

yields an independent subset A D A 0 [ fxg of M . Since we have in

both cases that w.A/ D w.A 0 /Cw.x/, a maximum-weight solution in M contain-

ing x yields a maximum-weight solution in M 0

, and vice versa.

Theorem 16.11 (Correctness of the greedy algorithm on matroids)

If M D .S; / is a weighted matroid with weight function w, then GREEDY.M; w/

returns an optimal subset.

Proof By Corollary 16.9, any elements that GREEDY passes over initially be-

cause they are not extensions of ; can be forgotten about, since they can never

be useful. Once GREEDY selects the ﬁrst element x, Lemma 16.7 implies that

the algorithm does not err by adding x to A, since there exists an optimal subset

containing x. Finally, Lemma 16.10 implies that the remaining problem is one of

ﬁnding an optimal subset in the matroid M 0

that is the contraction of M by x.

After the procedure GREEDY sets A to fxg, we can interpret all of its remaining

steps as acting in the matroid M 0 D .S 0 ; 0 /, because B is independent in M 0

if

and only if B [ fxg is independent in M , for all sets B 2 0

. Thus, the subsequent

operation of GREEDY will ﬁnd a maximum-weight independent subset for M 0

, and

the overall operation of GREEDY will ﬁnd a maximum-weight independent subset

for M .

16.5 A task-scheduling problem as a matroid 443

Exercises

16.4-1

Show that .S; k / is a matroid, where S is any ﬁnite set and k is the set of all

subsets of S of size at most k, where k jSj.

16.4-2 ?

Given an m n matrix T over some ﬁeld (such as the reals), show that .S; / is a

matroid, where S is the set of columns of T and A 2 if and only if the columns

in A are linearly independent.

16.4-3 ?

Show that if .S; / is a matroid, then .S; 0 / is a matroid, where

0

D fA

0

W S A

0

contains some maximal A 2 g :

That is, the maximal independent sets of .S; 0 / are just the complements of the

maximal independent sets of .S; /.

16.4-4 ?

Let S be a ﬁnite set and let S 1 ; S 2 ; : : : ; S k be a partition of S into nonempty disjoint

subsets. Deﬁne the structure .S; / by the condition that D fA W jA \ S i j 1

for i D 1; 2; : : : ; kg. Show that .S; / is a matroid. That is, the set of all sets A

that contain at most one member of each subset in the partition determines the

independent sets of a matroid.

16.4-5

Show how to transform the weight function of a weighted matroid problem, where

the desired optimal solution is a minimum-weight maximal independent subset, to

make it a standard weighted-matroid problem. Argue carefully that your transfor-

mation is correct.

? 16.5 A task-scheduling problem as a matroid

An interesting problem that we can solve using matroids is the problem of op-

timally scheduling unit-time tasks on a single processor, where each task has a

deadline, along with a penalty paid if the task misses its deadline. The problem

looks complicated, but we can solve it in a surprisingly simple manner by casting

it as a matroid and using a greedy algorithm.

A unit-time task is a job, such as a program to be run on a computer, that requires

exactly one unit of time to complete. Given a ﬁnite set S of unit-time tasks, a

444 Chapter 16 Greedy Algorithms

schedule for S is a permutation of S specifying the order in which to perform

these tasks. The ﬁrst task in the schedule begins at time 0 and ﬁnishes at time 1,

the second task begins at time 1 and ﬁnishes at time 2, and so on.

The problem of scheduling unit-time tasks with deadlines and penalties for a

single processor has the following inputs:

a set S D fa 1 ; a 2 ; : : : ; a n g of n unit-time tasks;

a set of n integer deadlines d 1 ; d 2 ; : : : ; d n , such that each d i satisﬁes 1 d i n

and task a i is supposed to ﬁnish by time d i ; and

a set of n nonnegative weights or penalties w 1 ; w 2 ; : : : ; w n , such that we incur

a penalty of w i if task a i is not ﬁnished by time d i , and we incur no penalty if

a task ﬁnishes by its deadline.

We wish to ﬁnd a schedule for S that minimizes the total penalty incurred for

missed deadlines.

Consider a given schedule. We say that a task is late in this schedule if it ﬁnishes

after its deadline. Otherwise, the task is early in the schedule. We can always trans-

form an arbitrary schedule into early-ﬁrst form, in which the early tasks precede

the late tasks. To see why, note that if some early task a i follows some late task a j ,

then we can switch the positions of a i and a j , and a i will still be early and a j will

still be late.

Furthermore, we claim that we can always transform an arbitrary schedule into

canonical form, in which the early tasks precede the late tasks and we schedule

the early tasks in order of monotonically increasing deadlines. To do so, we put

the schedule into early-ﬁrst form. Then, as long as there exist two early tasks a i

and a j ﬁnishing at respective times k and k C 1 in the schedule such that d j < d i ,

we swap the positions of a i and a j . Since a j is early before the swap, k C 1 d j .

Therefore, k C 1 < d i , and so a i is still early after the swap. Because task a j is

moved earlier in the schedule, it remains early after the swap.

The search for an optimal schedule thus reduces to ﬁnding a set A of tasks that

we assign to be early in the optimal schedule. Having determined A, we can create

the actual schedule by listing the elements of A in order of monotonically increas-

ing deadlines, then listing the late tasks (i.e., S A) in any order, producing a

canonical ordering of the optimal schedule.

We say that a set A of tasks is independent if there exists a schedule for these

tasks such that no tasks are late. Clearly, the set of early tasks for a schedule forms

an independent set of tasks. Let denote the set of all independent sets of tasks.

Consider the problem of determining whether a given set A of tasks is indepen-

dent. For t D 0; 1; 2; : : : ; n, let N t .A/ denote the number of tasks in A whose

deadline is t or earlier. Note that N 0 .A/ D 0 for any set A.

16.5 A task-scheduling problem as a matroid 445

Lemma 16.12

For any set of tasks A, the following statements are equivalent.

1. The set A is independent.

2. For t D 0; 1; 2; : : : ; n, we have N t .A/ t.

3. If the tasks in A are scheduled in order of monotonically increasing deadlines,

then no task is late.

Proof To show that (1) implies (2), we prove the contrapositive: if N t .A/ > t for

some t, then there is no way to make a schedule with no late tasks for set A, because

more than t tasks must ﬁnish before time t. Therefore, (1) implies (2). If (2) holds,

then (3) must follow: there is no way to “get stuck” when scheduling the tasks in

order of monotonically increasing deadlines, since (2) implies that the ith largest

deadline is at least i. Finally, (3) trivially implies (1).

Using property 2 of Lemma 16.12, we can easily compute whether or not a given

set of tasks is independent (see Exercise 16.5-2).

The problem of minimizing the sum of the penalties of the late tasks is the same

as the problem of maximizing the sum of the penalties of the early tasks. The

following theorem thus ensures that we can use the greedy algorithm to ﬁnd an

independent set A of tasks with the maximum total penalty.

Theorem 16.13

If S is a set of unit-time tasks with deadlines, and is the set of all independent

sets of tasks, then the corresponding system .S; / is a matroid.

Proof Every subset of an independent set of tasks is certainly independent. To

prove the exchange property, suppose that B and A are independent sets of tasks

and that jBj > jAj. Let k be the largest t such that N t .B/ N t .A/. (Such a value

of t exists, since N 0 .A/ D N 0 .B/ D 0.) Since N n .B/ D jBj and N n .A/ D jAj,

but jBj > jAj, we must have that k < n and that N j .B/ > N j .A/ for all j in

the range k C 1 j n. Therefore, B contains more tasks with deadline k C 1

than A does. Let a i be a task in B A with deadline k C 1. Let A 0 D A [ fa i g.

We now show that A 0

must be independent by using property 2 of Lemma 16.12.

For 0 t k, we have N t .A 0 / D N t .A/ t, since A is independent. For

k < t n, we have N t .A 0 / N t .B/ t, since B is independent. Therefore, A 0

is independent, completing our proof that .S; / is a matroid.

By Theorem 16.11, we can use a greedy algorithm to ﬁnd a maximum-weight

independent set of tasks A. We can then create an optimal schedule having the

tasks in A as its early tasks. This method is an efﬁcient algorithm for scheduling

446 Chapter 16 Greedy Algorithms

Task

ai 1 2 3 4 5 6 7

di 4 2 4 3 1 4 6

wi 70 60 50 40 30 20 10

Figure 16.7 An instance of the problem of scheduling unit-time tasks with deadlines and penalties

for a single processor.

unit-time tasks with deadlines and penalties for a single processor. The running

time is O.n 2 / using GREEDY, since each of the O.n/ independence checks made

by that algorithm takes time O.n/ (see Exercise 16.5-2). Problem 16-4 gives a

faster implementation.

Figure 16.7 demonstrates an example of the problem of scheduling unit-time

tasks with deadlines and penalties for a single processor. In this example, the

greedy algorithm selects, in order, tasks a 1 , a 2 , a 3 , and a 4 , then rejects a 5 (because

N 4 .fa 1 ; a 2 ; a 3 ; a 4 ; a 5 g/ D 5) and a 6 (because N 4 .fa 1 ; a 2 ; a 3 ; a 4 ; a 6 g/ D 5), and

ﬁnally accepts a 7 . The ﬁnal optimal schedule is

ha 2 ; a 4 ; a 1 ; a 3 ; a 7 ; a 5 ; a 6 i ;

which has a total penalty incurred of w 5 C w 6 D 50.

Exercises

16.5-1

Solve the instance of the scheduling problem given in Figure 16.7, but with each

penalty w i replaced by 80 w i .

16.5-2

Show how to use property 2 of Lemma 16.12 to determine in time O.jAj/ whether

or not a given set A of tasks is independent.

Problems

16-1 Coin changing

Consider the problem of making change for n cents using the fewest number of

coins. Assume that each coin’s value is an integer.

a. Describe a greedy algorithm to make change consisting of quarters, dimes,

nickels, and pennies. Prove that your algorithm yields an optimal solution.

Problems for Chapter 16 447

b. Suppose that the available coins are in the denominations that are powers of c,

i.e., the denominations are c 0 ; c 1 ; : : : ; c k

for some integers c > 1 and k 1.

Show that the greedy algorithm always yields an optimal solution.

c. Give a set of coin denominations for which the greedy algorithm does not yield

an optimal solution. Your set should include a penny so that there is a solution

for every value of n.

d. Give an O.nk/-time algorithm that makes change for any set of k different coin

denominations, assuming that one of the coins is a penny.

16-2 Scheduling to minimize average completion time

Suppose you are given a set S D fa 1 ; a 2 ; : : : ; a n g of tasks, where task a i re-

quires p i units of processing time to complete, once it has started. You have one

computer on which to run these tasks, and the computer can run only one task at a

time. Let c i be the completion time of task a i , that is, the time at which task a i com-

pletes processing. Your goal is to minimize the average completion time, that is,

to minimize .1=n/

P n

iD1

c i . For example, suppose there are two tasks, a 1 and a 2 ,

with p 1 D 3 and p 2 D 5, and consider the schedule in which a 2 runs ﬁrst, followed

by a 1 . Then c 2 D 5, c 1 D 8, and the average completion time is .5 C 8/=2 D 6:5.

If task a 1 runs ﬁrst, however, then c 1 D 3, c 2 D 8, and the average completion

time is .3 C 8/=2 D 5:5.

a. Give an algorithm that schedules the tasks so as to minimize the average com-

pletion time. Each task must run non-preemptively, that is, once task a i starts, it

must run continuously for p i units of time. Prove that your algorithm minimizes

the average completion time, and state the running time of your algorithm.

b. Suppose now that the tasks are not all available at once. That is, each task

cannot start until its release time r i . Suppose also that we allow preemption, so

that a task can be suspended and restarted at a later time. For example, a task a i

with processing time p i D 6 and release time r i D 1 might start running at

time 1 and be preempted at time 4. It might then resume at time 10 but be

preempted at time 11, and it might ﬁnally resume at time 13 and complete at

time 15. Task a i has run for a total of 6 time units, but its running time has been

divided into three pieces. In this scenario, a i ’s completion time is 15. Give

an algorithm that schedules the tasks so as to minimize the average completion

time in this new scenario. Prove that your algorithm minimizes the average

completion time, and state the running time of your algorithm.

448 Chapter 16 Greedy Algorithms

16-3 Acyclic subgraphs

a. The incidence matrix for an undirected graph G D .V; E/ is a jV j jEj ma-

trix M such that M e D 1 if edge e is incident on vertex , and M e D 0 other-

wise. Argue that a set of columns of M is linearly independent over the ﬁeld

of integers modulo 2 if and only if the corresponding set of edges is acyclic.

Then, use the result of Exercise 16.4-2 to provide an alternate proof that .E; /

of part (a) is a matroid.

b. Suppose that we associate a nonnegative weight w.e/ with each edge in an

undirected graph G D .V; E/. Give an efﬁcient algorithm to ﬁnd an acyclic

subset of E of maximum total weight.

c. Let G.V; E/ be an arbitrary directed graph, and let .E; / be deﬁned so that

A 2 if and only if A does not contain any directed cycles. Give an example

of a directed graph G such that the associated system .E; / is not a matroid.

Specify which deﬁning condition for a matroid fails to hold.

d. The incidence matrix for a directed graph G D .V; E/ with no self-loops is a

jV j jEj matrix M such that M e D 1 if edge e leaves vertex , M e D 1 if

edge e enters vertex , and M e D 0 otherwise. Argue that if a set of columns

of M is linearly independent, then the corresponding set of edges does not

contain a directed cycle.

e. Exercise 16.4-2 tells us that the set of linearly independent sets of columns of

any matrix M forms a matroid. Explain carefully why the results of parts (d)

and (e) are not contradictory. How can there fail to be a perfect correspon-

dence between the notion of a set of edges being acyclic and the notion of the

associated set of columns of the incidence matrix being linearly independent?

16-4 Scheduling variations

Consider the following algorithm for the problem from Section 16.5 of scheduling

unit-time tasks with deadlines and penalties. Let all n time slots be initially empty,

where time slot i is the unit-length slot of time that ﬁnishes at time i. We consider

the tasks in order of monotonically decreasing penalty. When considering task a j ,

if there exists a time slot at or before a j ’s deadline d j that is still empty, assign a j

to the latest such slot, ﬁlling it. If there is no such slot, assign task a j to the latest

of the as yet unﬁlled slots.

a. Argue that this algorithm always gives an optimal answer.

b. Use the fast disjoint-set forest presented in Section 21.3 to implement the algo-

rithm efﬁciently. Assume that the set of input tasks has already been sorted into

Problems for Chapter 16 449

monotonically decreasing order by penalty. Analyze the running time of your

implementation.

16-5 Off-line caching

Modern computers use a cache to store a small amount of data in a fast memory.

Even though a program may access large amounts of data, by storing a small subset

of the main memory in the cache—a small but faster memory—overall access time

can greatly decrease. When a computer program executes, it makes a sequence

hr 1 ; r 2 ; : : : ; r n i of n memory requests, where each request is for a particular data

element. For example, a program that accesses 4 distinct elements fa; b; c; dg

might make the sequence of requests hd; b; d; b; d; a; c; d; b; a; c; bi. Let k be the

size of the cache. When the cache contains k elements and the program requests the

.k C 1/st element, the system must decide, for this and each subsequent request,

which k elements to keep in the cache. More precisely, for each request r i , the

cache-management algorithm checks whether element r i is already in the cache. If

it is, then we have a cache hit; otherwise, we have a cache miss. Upon a cache

miss, the system retrieves r i from the main memory, and the cache-management

algorithm must decide whether to keep r i in the cache. If it decides to keep r i and

the cache already holds k elements, then it must evict one element to make room

for r i . The cache-management algorithm evicts data with the goal of minimizing

the number of cache misses over the entire sequence of requests.

Typically, caching is an on-line problem. That is, we have to make decisions

about which data to keep in the cache without knowing the future requests. Here,

however, we consider the off-line version of this problem, in which we are given

in advance the entire sequence of n requests and the cache size k, and we wish to

minimize the total number of cache misses.

We can solve this off-line problem by a greedy strategy called furthest-in-future,

which chooses to evict the item in the cache whose next access in the request

sequence comes furthest in the future.

a. Write pseudocode for a cache manager that uses the furthest-in-future strategy.

The input should be a sequence hr 1 ; r 2 ; : : : ; r n i of requests and a cache size k,

and the output should be a sequence of decisions about which data element (if

any) to evict upon each request. What is the running time of your algorithm?

b. Show that the off-line caching problem exhibits optimal substructure.

c. Prove that furthest-in-future produces the minimum possible number of cache

misses.

450 Chapter 16 Greedy Algorithms

Chapter notes

Much more material on greedy algorithms and matroids can be found in Lawler

[224] and Papadimitriou and Steiglitz [271].

The greedy algorithm ﬁrst appeared in the combinatorial optimization literature

in a 1971 article by Edmonds [101], though the theory of matroids dates back to

a 1935 article by Whitney [355].

Our proof of the correctness of the greedy algorithm for the activity-selection

problem is based on that of Gavril [131]. The task-scheduling problem is studied

in Lawler [224]; Horowitz, Sahni, and Rajasekaran [181]; and Brassard and Bratley

[54].

Huffman codes were invented in 1952 [185]; Lelewer and Hirschberg [231] sur-

veys data-compression techniques known as of 1987.

An extension of matroid theory to greedoid theory was pioneered by Korte and

Lov´asz [216, 217, 218, 219], who greatly generalize the theory presented here.

17 Amortized Analysis

In an amortized analysis, we average the time required to perform a sequence of

data-structure operations over all the operations performed. With amortized analy-

sis, we can show that the average cost of an operation is small, if we average over a

sequence of operations, even though a single operation within the sequence might

be expensive. Amortized analysis differs from average-case analysis in that prob-

ability is not involved; an amortized analysis guarantees the average performance

of each operation in the worst case.

The ﬁrst three sections of this chapter cover the three most common techniques

used in amortized analysis. Section 17.1 starts with aggregate analysis, in which

we determine an upper bound T .n/ on the total cost of a sequence of n operations.

The average cost per operation is then T .n/=n. We take the average cost as the

amortized cost of each operation, so that all operations have the same amortized

cost.

Section 17.2 covers the accounting method, in which we determine an amortized

cost of each operation. When there is more than one type of operation, each type of

operation may have a different amortized cost. The accounting method overcharges

some operations early in the sequence, storing the overcharge as “prepaid credit”

on speciﬁc objects in the data structure. Later in the sequence, the credit pays for

operations that are charged less than they actually cost.

Section 17.3 discusses the potential method, which is like the accounting method

in that we determine the amortized cost of each operation and may overcharge op-

erations early on to compensate for undercharges later. The potential method main-

tains the credit as the “potential energy” of the data structure as a whole instead of

associating the credit with individual objects within the data structure.

We shall use two examples to examine these three methods. One is a stack

with the additional operation MULTIPOP, which pops several objects at once. The

other is a binary counter that counts up from 0 by means of the single operation

INCREMENT.

452 Chapter 17 Amortized Analysis

While reading this chapter, bear in mind that the charges assigned during an

amortized analysis are for analysis purposes only. They need not—and should

not—appear in the code. If, for example, we assign a credit to an object x when

using the accounting method, we have no need to assign an appropriate amount to

some attribute, such as x:credit, in the code.

When we perform an amortized analysis, we often gain insight into a particular

data structure, and this insight can help us optimize the design. In Section 17.4,

for example, we shall use the potential method to analyze a dynamically expanding

and contracting table.

17.1 Aggregate analysis

In aggregate analysis, we show that for all n, a sequence of n operations takes

worst-case time T .n/ in total. In the worst case, the average cost, or amortized

cost, per operation is therefore T .n/=n. Note that this amortized cost applies to

each operation, even when there are several types of operations in the sequence.

The other two methods we shall study in this chapter, the accounting method and

the potential method, may assign different amortized costs to different types of

operations.

Stack operations

In our ﬁrst example of aggregate analysis, we analyze stacks that have been aug-

mented with a new operation. Section 10.1 presented the two fundamental stack

operations, each of which takes O.1/ time:

PUSH.S; x/ pushes object x onto stack S.

POP.S/ pops the top of stack S and returns the popped object. Calling POP on an

empty stack generates an error.

Since each of these operations runs in O.1/ time, let us consider the cost of each

to be 1. The total cost of a sequence of n PUSH and POP operations is therefore n,

and the actual running time for n operations is therefore ‚.n/.

Now we add the stack operation MULTIPOP.S; k/, which removes the k top ob-

jects of stack S, popping the entire stack if the stack contains fewer than k objects.

Of course, we assume that k is positive; otherwise the MULTIPOP operation leaves

the stack unchanged. In the following pseudocode, the operation STACK-EMPTY

returns TRUE if there are no objects currently on the stack, and FALSE otherwise.

17.1 Aggregate analysis 453

23

17

6

39

10

47

(a)

top

10

47

(b)

top

(c)

Figure 17.1 The action of MULTIPOP on a stack S, shown initially in (a). The top 4 objects are

popped by MULTIPOP.S; 4/, whose result is shown in (b). The next operation is MULTIPOP.S; 7/,

which empties the stack—shown in (c)—since there were fewer than 7 objects remaining.

MULTIPOP.S; k/

1 while not STACK-EMPTY.S/ and k > 0

2 POP.S/

3 k D k 1

Figure 17.1 shows an example of MULTIPOP.

What is the running time of MULTIPOP.S; k/ on a stack of s objects? The

actual running time is linear in the number of POP operations actually executed,

and thus we can analyze MULTIPOP in terms of the abstract costs of 1 each for

PUSH and POP. The number of iterations of the while loop is the number min.s; k/

of objects popped off the stack. Each iteration of the loop makes one call to POP in

line 2. Thus, the total cost of MULTIPOP is min.s; k/, and the actual running time

is a linear function of this cost.

Let us analyze a sequence of n PUSH, POP, and MULTIPOP operations on an ini-

tially empty stack. The worst-case cost of a MULTIPOP operation in the sequence

is O.n/, since the stack size is at most n. The worst-case time of any stack opera-

tion is therefore O.n/, and hence a sequence of n operations costs O.n 2 /, since we

may have O.n/ MULTIPOP operations costing O.n/ each. Although this analysis

is correct, the O.n 2 / result, which we obtained by considering the worst-case cost

of each operation individually, is not tight.

Using aggregate analysis, we can obtain a better upper bound that considers the

entire sequence of n operations. In fact, although a single MULTIPOP operation

can be expensive, any sequence of n PUSH, POP, and MULTIPOP operations on an

initially empty stack can cost at most O.n/. Why? We can pop each object from the

stack at most once for each time we have pushed it onto the stack. Therefore, the

number of times that POP can be called on a nonempty stack, including calls within

MULTIPOP, is at most the number of PUSH operations, which is at most n. For any

value of n, any sequence of n PUSH, POP, and MULTIPOP operations takes a total

of O.n/ time. The average cost of an operation is O.n/=n D O.1/. In aggregate

454 Chapter 17 Amortized Analysis

analysis, we assign the amortized cost of each operation to be the average cost. In

this example, therefore, all three stack operations have an amortized cost of O.1/.

We emphasize again that although we have just shown that the average cost, and

hence the running time, of a stack operation is O.1/, we did not use probabilistic

reasoning. We actually showed a worst-case bound of O.n/ on a sequence of n

operations. Dividing this total cost by n yielded the average cost per operation, or

the amortized cost.

Incrementing a binary counter

As another example of aggregate analysis, consider the problem of implementing

a k-bit binary counter that counts upward from 0. We use an array AŒ0 : : k 1 of

bits, where A:length D k, as the counter. A binary number x that is stored in the

counter has its lowest-order bit in AŒ0 and its highest-order bit in AŒk 1, so that

x D

P k1

iD0

AŒi 2 i

. Initially, x D 0, and thus AŒi D 0 for i D 0; 1; : : : ; k 1. To

add 1 (modulo 2 k

) to the value in the counter, we use the following procedure.

INCREMENT.A/

1 i D 0

2 while i < A:length and AŒi == 1

3 AŒi D 0

4 i D i C 1

5 if i < A:length

6 AŒi D 1

Figure 17.2 shows what happens to a binary counter as we increment it 16 times,

starting with the initial value 0 and ending with the value 16. At the start of

each iteration of the while loop in lines 2–4, we wish to add a 1 into position i.

If AŒi D 1, then adding 1 ﬂips the bit to 0 in position i and yields a carry of 1,

to be added into position i C 1 on the next iteration of the loop. Otherwise, the

loop ends, and then, if i < k, we know that AŒi D 0, so that line 6 adds a 1 into

position i, ﬂipping the 0 to a 1. The cost of each INCREMENT operation is linear

in the number of bits ﬂipped.

As with the stack example, a cursory analysis yields a bound that is correct but

not tight. A single execution of INCREMENT takes time ‚.k/ in the worst case, in

which array A contains all 1s. Thus, a sequence of n INCREMENT operations on

an initially zero counter takes time O.nk/ in the worst case.

We can tighten our analysis to yield a worst-case cost of O.n/ for a sequence of n

INCREMENT operations by observing that not all bits ﬂip each time INCREMENT

is called. As Figure 17.2 shows, AŒ0 does ﬂip each time INCREMENT is called.

The next bit up, AŒ1, ﬂips only every other time: a sequence of n INCREMENT

17.1 Aggregate analysis 455

0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 1 1

0 0 0 0 0 0 1 0 2

0 0 0 0 0 0 1 1 3

0 0 0 0 0 1 0 0 4

0 0 0 0 0 1 0 1 5

0 0 0 0 0 1 1 0 6

0 0 0 0 0 1 1 1 7

0 0 0 0 1 0 0 0 8

0 0 0 0 1 0 0 1 9

0 0 0 0 1 0 1 0 10

0 0 0 0 1 0 1 1 11

0 0 0 0 1 1 0 0 12

0 0 0 0 1 1 0 1 13

0 0 0 0 1 1 1 0 14

0 0 0 0 1 1 1 1 15

0 0 0 1 0 0 0 0 16

A[0]

A[1]

A[2]

A[3]

A[4]

A[5]

A[6]

A[7]

Counter

value

Total

cost

1

3

4

7

8

10

11

15

16

18

19

22

23

25

26

31

0

Figure 17.2 An 8-bit binary counter as its value goes from 0 to 16 by a sequence of 16 INCREMENT

operations. Bits that ﬂip to achieve the next value are shaded. The running cost for ﬂipping bits is

shown at the right. Notice that the total cost is always less than twice the total number of INCREMENT

operations.

operations on an initially zero counter causes AŒ1 to ﬂip bn=2c times. Similarly,

bit AŒ2 ﬂips only every fourth time, or bn=4c times in a sequence of n INCREMENT

operations. In general, for i D 0; 1; : : : ; k 1, bit AŒi ﬂips bn=2 i c times in a

sequence of n INCREMENT operations on an initially zero counter. For i k,

bit AŒi does not exist, and so it cannot ﬂip. The total number of ﬂips in the

sequence is thus

k1 X

iD0

j

n

2 i

k

< n

1 X

iD0

1

2 i

D 2n ;

by equation (A.6). The worst-case time for a sequence of n INCREMENT operations

on an initially zero counter is therefore O.n/. The average cost of each operation,

and therefore the amortized cost per operation, is O.n/=n D O.1/.

456 Chapter 17 Amortized Analysis

Exercises

17.1-1

If the set of stack operations included a MULTIPUSH operation, which pushes k

items onto the stack, would the O.1/ bound on the amortized cost of stack opera-

tions continue to hold?

17.1-2

Show that if a DECREMENT operation were included in the k-bit counter example,

n operations could cost as much as ‚.nk/ time.

17.1-3

Suppose we perform a sequence of n operations on a data structure in which the ith

operation costs i if i is an exact power of 2, and 1 otherwise. Use aggregate analysis

to determine the amortized cost per operation.

17.2 The accounting method

In the accounting method of amortized analysis, we assign differing charges to

different operations, with some operations charged more or less than they actu-

ally cost. We call the amount we charge an operation its amortized cost. When

an operation’s amortized cost exceeds its actual cost, we assign the difference to

speciﬁc objects in the data structure as credit. Credit can help pay for later oper-

ations whose amortized cost is less than their actual cost. Thus, we can view the

amortized cost of an operation as being split between its actual cost and credit that

is either deposited or used up. Different operations may have different amortized

costs. This method differs from aggregate analysis, in which all operations have

the same amortized cost.

We must choose the amortized costs of operations carefully. If we want to show

that in the worst case the average cost per operation is small by analyzing with

amortized costs, we must ensure that the total amortized cost of a sequence of oper-

ations provides an upper bound on the total actual cost of the sequence. Moreover,

as in aggregate analysis, this relationship must hold for all sequences of opera-

tions. If we denote the actual cost of the ith operation by c i and the amortized cost

of the ith operation by yc i , we require

n X

iD1

yc i

n X

iD1

c i (17.1)

for all sequences of n operations. The total credit stored in the data structure

is the difference between the total amortized cost and the total actual cost, or

17.2 The accounting method 457

P n

iD1

yc i

P n

iD1

c i . By inequality (17.1), the total credit associated with the data

structure must be nonnegative at all times. If we ever were to allow the total credit

to become negative (the result of undercharging early operations with the promise

of repaying the account later on), then the total amortized costs incurred at that

time would be below the total actual costs incurred; for the sequence of operations

up to that time, the total amortized cost would not be an upper bound on the total

actual cost. Thus, we must take care that the total credit in the data structure never

becomes negative.

Stack operations

To illustrate the accounting method of amortized analysis, let us return to the stack

example. Recall that the actual costs of the operations were

PUSH 1 ,

POP 1 ,

MULTIPOP min.k; s/ ,

where k is the argument supplied to MULTIPOP and s is the stack size when it is

called. Let us assign the following amortized costs:

PUSH 2 ,

POP 0 ,

MULTIPOP 0 .

Note that the amortized cost of MULTIPOP is a constant (0), whereas the actual cost

is variable. Here, all three amortized costs are constant. In general, the amortized

costs of the operations under consideration may differ from each other, and they

may even differ asymptotically.

We shall now show that we can pay for any sequence of stack operations by

charging the amortized costs. Suppose we use a dollar bill to represent each unit

of cost. We start with an empty stack. Recall the analogy of Section 10.1 between

the stack data structure and a stack of plates in a cafeteria. When we push a plate

on the stack, we use 1 dollar to pay the actual cost of the push and are left with a

credit of 1 dollar (out of the 2 dollars charged), which we leave on top of the plate.

At any point in time, every plate on the stack has a dollar of credit on it.

The dollar stored on the plate serves as prepayment for the cost of popping it

from the stack. When we execute a POP operation, we charge the operation nothing

and pay its actual cost using the credit stored in the stack. To pop a plate, we take

the dollar of credit off the plate and use it to pay the actual cost of the operation.

Thus, by charging the PUSH operation a little bit more, we can charge the POP

operation nothing.

458 Chapter 17 Amortized Analysis

Moreover, we can also charge MULTIPOP operations nothing. To pop the ﬁrst

plate, we take the dollar of credit off the plate and use it to pay the actual cost of a

POP operation. To pop a second plate, we again have a dollar of credit on the plate

to pay for the POP operation, and so on. Thus, we have always charged enough

up front to pay for MULTIPOP operations. In other words, since each plate on the

stack has 1 dollar of credit on it, and the stack always has a nonnegative number of

plates, we have ensured that the amount of credit is always nonnegative. Thus, for

any sequence of n PUSH, POP, and MULTIPOP operations, the total amortized cost

is an upper bound on the total actual cost. Since the total amortized cost is O.n/,

so is the total actual cost.

Incrementing a binary counter

As another illustration of the accounting method, we analyze the INCREMENT op-

eration on a binary counter that starts at zero. As we observed earlier, the running

time of this operation is proportional to the number of bits ﬂipped, which we shall

use as our cost for this example. Let us once again use a dollar bill to represent

each unit of cost (the ﬂipping of a bit in this example).

For the amortized analysis, let us charge an amortized cost of 2 dollars to set a

bit to 1. When a bit is set, we use 1 dollar (out of the 2 dollars charged) to pay

for the actual setting of the bit, and we place the other dollar on the bit as credit to

be used later when we ﬂip the bit back to 0. At any point in time, every 1 in the

counter has a dollar of credit on it, and thus we can charge nothing to reset a bit

to 0; we just pay for the reset with the dollar bill on the bit.

Now we can determine the amortized cost of INCREMENT. The cost of resetting

the bits within the while loop is paid for by the dollars on the bits that are reset. The

INCREMENT procedure sets at most one bit, in line 6, and therefore the amortized

cost of an INCREMENT operation is at most 2 dollars. The number of 1s in the

counter never becomes negative, and thus the amount of credit stays nonnegative

at all times. Thus, for n INCREMENT operations, the total amortized cost is O.n/,

which bounds the total actual cost.

Exercises

17.2-1

Suppose we perform a sequence of stack operations on a stack whose size never

exceeds k. After every k operations, we make a copy of the entire stack for backup

purposes. Show that the cost of n stack operations, including copying the stack,

is O.n/ by assigning suitable amortized costs to the various stack operations.

17.3 The potential method 459

17.2-2

Redo Exercise 17.1-3 using an accounting method of analysis.

17.2-3

Suppose we wish not only to increment a counter but also to reset it to zero (i.e.,

make all bits in it 0). Counting the time to examine or modify a bit as ‚.1/,

show how to implement a counter as an array of bits so that any sequence of n

INCREMENT and RESET operations takes time O.n/ on an initially zero counter.

(Hint: Keep a pointer to the high-order 1.)

17.3 The potential method

Instead of representing prepaid work as credit stored with speciﬁc objects in the

data structure, the potential method of amortized analysis represents the prepaid

work as “potential energy,” or just “potential,” which can be released to pay for

future operations. We associate the potential with the data structure as a whole

rather than with speciﬁc objects within the data structure.

The potential method works as follows. We will perform n operations, starting

with an initial data structure D 0 . For each i D 1; 2; : : : ; n, we let c i be the actual

cost of the ith operation and D i be the data structure that results after applying

the ith operation to data structure D i1 . A potential function ˆ maps each data

structure D i to a real number ˆ.D i /, which is the potential associated with data

structure D i . The amortized cost yc i of the ith operation with respect to potential

function ˆ is deﬁned by

yc i D c i C ˆ.D i / ˆ.D i1 / : (17.2)

The amortized cost of each operation is therefore its actual cost plus the change in

potential due to the operation. By equation (17.2), the total amortized cost of the n

operations is

n X

iD1

yc i D

n X

iD1

.c i C ˆ.D i / ˆ.D i1 //

D

n X

iD1

c i C ˆ.D n / ˆ.D 0 / : (17.3)

The second equality follows from equation (A.9) because the ˆ.D i / terms tele-

scope.

If we can deﬁne a potential function ˆ so that ˆ.D n / ˆ.D 0 /, then the total

amortized cost

P n

iD1

yc i gives an upper bound on the total actual cost

P n

iD1

c i .

460 Chapter 17 Amortized Analysis

In practice, we do not always know how many operations might be performed.

Therefore, if we require that ˆ.D i / ˆ.D 0 / for all i, then we guarantee, as in

the accounting method, that we pay in advance. We usually just deﬁne ˆ.D 0 / to

be 0 and then show that ˆ.D i / 0 for all i. (See Exercise 17.3-1 for an easy way

to handle cases in which ˆ.D 0 / ¤ 0.)

Intuitively, if the potential difference ˆ.D i / ˆ.D i1 / of the ith operation is

positive, then the amortized cost yc i represents an overcharge to the ith operation,

and the potential of the data structure increases. If the potential difference is neg-

ative, then the amortized cost represents an undercharge to the ith operation, and

the decrease in the potential pays for the actual cost of the operation.

The amortized costs deﬁned by equations (17.2) and (17.3) depend on the choice

of the potential function ˆ. Different potential functions may yield different amor-

tized costs yet still be upper bounds on the actual costs. We often ﬁnd trade-offs

that we can make in choosing a potential function; the best potential function to

use depends on the desired time bounds.

Stack operations

To illustrate the potential method, we return once again to the example of the stack

operations PUSH, POP, and MULTIPOP. We deﬁne the potential function ˆ on a

stack to be the number of objects in the stack. For the empty stack D 0 with which

we start, we have ˆ.D 0 / D 0. Since the number of objects in the stack is never

negative, the stack D i that results after the ith operation has nonnegative potential,

and thus

ˆ.D i / 0

D ˆ.D 0 / :

The total amortized cost of n operations with respect to ˆ therefore represents an

upper bound on the actual cost.

Let us now compute the amortized costs of the various stack operations. If the ith

operation on a stack containing s objects is a PUSH operation, then the potential

difference is

ˆ.D i / ˆ.D i1 / D .s C 1/ s

D 1 :

By equation (17.2), the amortized cost of this PUSH operation is

yc i D c i C ˆ.D i / ˆ.D i1 /

D 1 C 1

D 2 :

17.3 The potential method 461

Suppose that the ith operation on the stack is MULTIPOP.S; k/, which causes

k 0 D min.k; s/ objects to be popped off the stack. The actual cost of the opera-

tion is k 0

, and the potential difference is

ˆ.D i / ˆ.D i1 / D k

0

:

Thus, the amortized cost of the MULTIPOP operation is

yc i D c i C ˆ.D i / ˆ.D i1 /

D k

0

k

0

D 0 :

Similarly, the amortized cost of an ordinary POP operation is 0.

The amortized cost of each of the three operations is O.1/, and thus the total

amortized cost of a sequence of n operations is O.n/. Since we have already argued

that ˆ.D i / ˆ.D 0 /, the total amortized cost of n operations is an upper bound

on the total actual cost. The worst-case cost of n operations is therefore O.n/.

Incrementing a binary counter

As another example of the potential method, we again look at incrementing a binary

counter. This time, we deﬁne the potential of the counter after the ith INCREMENT

operation to be b i , the number of 1s in the counter after the ith operation.

Let us compute the amortized cost of an INCREMENT operation. Suppose that

the ith INCREMENT operation resets t i bits. The actual cost of the operation is

therefore at most t i C 1, since in addition to resetting t i bits, it sets at most one

bit to 1. If b i D 0, then the ith operation resets all k bits, and so b i1 D t i D k.

If b i > 0, then b i D b i1 t i C 1. In either case, b i b i1 t i C 1, and the

potential difference is

ˆ.D i / ˆ.D i1 / .b i1 t i C 1/ b i1

D 1 t i :

The amortized cost is therefore

yc i D c i C ˆ.D i / ˆ.D i1 /

.t i C 1/ C .1 t i /

D 2 :

If the counter starts at zero, then ˆ.D 0 / D 0. Since ˆ.D i / 0 for all i, the total

amortized cost of a sequence of n INCREMENT operations is an upper bound on the

total actual cost, and so the worst-case cost of n INCREMENT operations is O.n/.

The potential method gives us an easy way to analyze the counter even when

it does not start at zero. The counter starts with b 0 1s, and after n INCREMENT

462 Chapter 17 Amortized Analysis

operations it has b n 1s, where 0 b 0 ; b n k. (Recall that k is the number of bits

in the counter.) We can rewrite equation (17.3) as

n X

iD1

c i D

n X

iD1

yc i ˆ.D n / C ˆ.D 0 / : (17.4)

We have yc i 2 for all 1 i n. Since ˆ.D 0 / D b 0 and ˆ.D n / D b n , the total

actual cost of n INCREMENT operations is

n X

iD1

c i

n X

iD1

2 b n C b 0

D 2n b n C b 0 :

Note in particular that since b 0 k, as long as k D O.n/, the total actual cost

is O.n/. In other words, if we execute at least n D .k/ INCREMENT operations,

the total actual cost is O.n/, no matter what initial value the counter contains.

Exercises

17.3-1

Suppose we have a potential function ˆ such that ˆ.D i / ˆ.D 0 / for all i, but

ˆ.D 0 / ¤ 0. Show that there exists a potential function ˆ 0

such that ˆ 0 .D 0 / D 0,

ˆ 0 .D i / 0 for all i 1, and the amortized costs using ˆ 0

are the same as the

amortized costs using ˆ.

17.3-2

Redo Exercise 17.1-3 using a potential method of analysis.

17.3-3

Consider an ordinary binary min-heap data structure with n elements supporting

the instructions INSERT and EXTRACT-MIN in O.lg n/ worst-case time. Give a

potential function ˆ such that the amortized cost of INSERT is O.lg n/ and the

amortized cost of EXTRACT-MIN is O.1/, and show that it works.

17.3-4

What is the total cost of executing n of the stack operations PUSH, POP, and

MULTIPOP, assuming that the stack begins with s 0 objects and ﬁnishes with s n

objects?

17.3-5

Suppose that a counter begins at a number with b 1s in its binary representa-

tion, rather than at 0. Show that the cost of performing n INCREMENT operations

is O.n/ if n D .b/. (Do not assume that b is constant.)

17.4 Dynamic tables 463

17.3-6

Show how to implement a queue with two ordinary stacks (Exercise 10.1-6) so that

the amortized cost of each ENQUEUE and each DEQUEUE operation is O.1/.

17.3-7

Design a data structure to support the following two operations for a dynamic

multiset S of integers, which allows duplicate values:

INSERT.S; x/ inserts x into S.

DELETE-LARGER-HALF.S/ deletes the largest djSj =2e elements from S.

Explain how to implement this data structure so that any sequence of m INSERT

and DELETE-LARGER-HALF operations runs in O.m/ time. Your implementation

should also include a way to output the elements of S in O.jSj/ time.

17.4 Dynamic tables

We do not always know in advance how many objects some applications will store

in a table. We might allocate space for a table, only to ﬁnd out later that it is not

enough. We must then reallocate the table with a larger size and copy all objects

stored in the original table over into the new, larger table. Similarly, if many objects

have been deleted from the table, it may be worthwhile to reallocate the table with

a smaller size. In this section, we study this problem of dynamically expanding and

contracting a table. Using amortized analysis, we shall show that the amortized cost

of insertion and deletion is only O.1/, even though the actual cost of an operation

is large when it triggers an expansion or a contraction. Moreover, we shall see how

to guarantee that the unused space in a dynamic table never exceeds a constant

fraction of the total space.

We assume that the dynamic table supports the operations TABLE-INSERT and

TABLE-DELETE. TABLE-INSERT inserts into the table an item that occupies a sin-

gle slot, that is, a space for one item. Likewise, TABLE-DELETE removes an item

from the table, thereby freeing a slot. The details of the data-structuring method

used to organize the table are unimportant; we might use a stack (Section 10.1),

a heap (Chapter 6), or a hash table (Chapter 11). We might also use an array or

collection of arrays to implement object storage, as we did in Section 10.3.

We shall ﬁnd it convenient to use a concept introduced in our analysis of hashing

(Chapter 11). We deﬁne the load factor ˛.T / of a nonempty table T to be the

number of items stored in the table divided by the size (number of slots) of the

table. We assign an empty table (one with no items) size 0, and we deﬁne its load

factor to be 1. If the load factor of a dynamic table is bounded below by a constant,

464 Chapter 17 Amortized Analysis

the unused space in the table is never more than a constant fraction of the total

amount of space.

We start by analyzing a dynamic table in which we only insert items. We then

consider the more general case in which we both insert and delete items.

17.4.1 Table expansion

Let us assume that storage for a table is allocated as an array of slots. A table ﬁlls

up when all slots have been used or, equivalently, when its load factor is 1. 1 In some

software environments, upon attempting to insert an item into a full table, the only

alternative is to abort with an error. We shall assume, however, that our software

environment, like many modern ones, provides a memory-management system that

can allocate and free blocks of storage on request. Thus, upon inserting an item

into a full table, we can expand the table by allocating a new table with more slots

than the old table had. Because we always need the table to reside in contiguous

memory, we must allocate a new array for the larger table and then copy items from

the old table into the new table.

A common heuristic allocates a new table with twice as many slots as the old

one. If the only table operations are insertions, then the load factor of the table is

always at least 1=2, and thus the amount of wasted space never exceeds half the

total space in the table.

In the following pseudocode, we assume that T is an object representing the

table. The attribute T:table contains a pointer to the block of storage representing

the table, T:num contains the number of items in the table, and T:size gives the total

number of slots in the table. Initially, the table is empty: T:num D T:size D 0.

TABLE-INSERT.T; x/

1 if T:size == 0

2 allocate T:table with 1 slot

3 T:size D 1

4 if T:num == T:size

5 allocate new-table with 2 T:size slots

6 insert all items in T:table into new-table

7 free T:table

8 T:table D new-table

9 T:size D 2 T:size

10 insert x into T:table

11 T:num D T:num C 1

1

In some situations, such as an open-address hash table, we may wish to consider a table to be full if

its load factor equals some constant strictly less than 1. (See Exercise 17.4-1.)

17.4 Dynamic tables 465

Notice that we have two “insertion” procedures here: the TABLE-INSERT proce-

dure itself and the elementary insertion into a table in lines 6 and 10. We can

analyze the running time of TABLE-INSERT in terms of the number of elementary

insertions by assigning a cost of 1 to each elementary insertion. We assume that

the actual running time of TABLE-INSERT is linear in the time to insert individual

items, so that the overhead for allocating an initial table in line 2 is constant and

the overhead for allocating and freeing storage in lines 5 and 7 is dominated by

the cost of transferring items in line 6. We call the event in which lines 5–9 are

executed an expansion.

Let us analyze a sequence of n TABLE-INSERT operations on an initially empty

table. What is the cost c i of the ith operation? If the current table has room for the

new item (or if this is the ﬁrst operation), then c i D 1, since we need only perform

the one elementary insertion in line 10. If the current table is full, however, and an

expansion occurs, then c i D i: the cost is 1 for the elementary insertion in line 10

plus i 1 for the items that we must copy from the old table to the new table in

line 6. If we perform n operations, the worst-case cost of an operation is O.n/,

which leads to an upper bound of O.n 2 / on the total running time for n operations.

This bound is not tight, because we rarely expand the table in the course of n

TABLE-INSERT operations. Speciﬁcally, the ith operation causes an expansion

only when i 1 is an exact power of 2. The amortized cost of an operation is in

fact O.1/, as we can show using aggregate analysis. The cost of the ith operation

is

c i D

(

i if i 1 is an exact power of 2 ;

1 otherwise :

The total cost of n TABLE-INSERT operations is therefore

n X

iD1

c i n C

blg nc X

j D0

2

j

< n C 2n

D 3n ;

because at most n operations cost 1 and the costs of the remaining operations form

a geometric series. Since the total cost of n TABLE-INSERT operations is bounded

by 3n, the amortized cost of a single operation is at most 3.

By using the accounting method, we can gain some feeling for why the amor-

tized cost of a TABLE-INSERT operation should be 3. Intuitively, each item pays

for 3 elementary insertions: inserting itself into the current table, moving itself

when the table expands, and moving another item that has already been moved

once when the table expands. For example, suppose that the size of the table is m

immediately after an expansion. Then the table holds m=2 items, and it contains

466 Chapter 17 Amortized Analysis

no credit. We charge 3 dollars for each insertion. The elementary insertion that

occurs immediately costs 1 dollar. We place another dollar as credit on the item

inserted. We place the third dollar as credit on one of the m=2 items already in the

table. The table will not ﬁll again until we have inserted another m=2 1 items,

and thus, by the time the table contains m items and is full, we will have placed a

dollar on each item to pay to reinsert it during the expansion.

We can use the potential method to analyze a sequence of n TABLE-INSERT

operations, and we shall use it in Section 17.4.2 to design a TABLE-DELETE op-

eration that has an O.1/ amortized cost as well. We start by deﬁning a potential

function ˆ that is 0 immediately after an expansion but builds to the table size by

the time the table is full, so that we can pay for the next expansion by the potential.

The function

ˆ.T / D 2 T:num T:size (17.5)

is one possibility. Immediately after an expansion, we have T:num D T:size=2,

and thus ˆ.T / D 0, as desired. Immediately before an expansion, we have

T:num D T:size, and thus ˆ.T / D T:num, as desired. The initial value of the

potential is 0, and since the table is always at least half full, T:num T:size=2,

which implies that ˆ.T / is always nonnegative. Thus, the sum of the amortized

costs of n TABLE-INSERT operations gives an upper bound on the sum of the actual

costs.

To analyze the amortized cost of the ith TABLE-INSERT operation, we let num i

denote the number of items stored in the table after the ith operation, size i denote

the total size of the table after the ith operation, and ˆ i denote the potential after

the ith operation. Initially, we have num 0 D 0, size 0 D 0, and ˆ 0 D 0.

If the ith TABLE-INSERT operation does not trigger an expansion, then we have

size i D size i1 and the amortized cost of the operation is

yc i D c i C ˆ i ˆ i1

D 1 C .2 num i size i / .2 num i1 size i1 /

D 1 C .2 num i size i / .2.num i 1/ size i /

D 3 :

If the ith operation does trigger an expansion, then we have size i D 2 size i1 and

size i1 D num i1 D num i 1, which implies that size i D 2 .num i 1/. Thus,

the amortized cost of the operation is

yc i D c i C ˆ i ˆ i1

D num i C .2 num i size i / .2 num i1 size i1 /

D num i C .2 num i 2 .num i 1// .2.num i 1/ .num i 1//

D num i C 2 .num i 1/

D 3 :

17.4 Dynamic tables 467

Φ i

num

i

size

i

0 8 16 24 32

0

8

16

24

32

i

Figure 17.3 The effect of a sequence of n TABLE-INSERT operations on the number numi of items

in the table, the number sizei of slots in the table, and the potential ˆi D 2 numi sizei , each

being measured after the ith operation. The thin line shows numi , the dashed line shows sizei , and

the thick line shows ˆi . Notice that immediately before an expansion, the potential has built up to

the number of items in the table, and therefore it can pay for moving all the items to the new table.

Afterwards, the potential drops to 0, but it is immediately increased by 2 upon inserting the item that

caused the expansion.

Figure 17.3 plots the values of num i , size i , and ˆ i against i. Notice how the

potential builds to pay for expanding the table.

17.4.2 Table expansion and contraction

To implement a TABLE-DELETE operation, it is simple enough to remove the spec-

iﬁed item from the table. In order to limit the amount of wasted space, however,

we might wish to contract the table when the load factor becomes too small. Table

contraction is analogous to table expansion: when the number of items in the table

drops too low, we allocate a new, smaller table and then copy the items from the

old table into the new one. We can then free the storage for the old table by return-

ing it to the memory-management system. Ideally, we would like to preserve two

properties:

the load factor of the dynamic table is bounded below by a positive constant,

and

the amortized cost of a table operation is bounded above by a constant.

468 Chapter 17 Amortized Analysis

We assume that we measure the cost in terms of elementary insertions and dele-

tions.

You might think that we should double the table size upon inserting an item into

a full table and halve the size when a deleting an item would cause the table to

become less than half full. This strategy would guarantee that the load factor of

the table never drops below 1=2, but unfortunately, it can cause the amortized cost

of an operation to be quite large. Consider the following scenario. We perform n

operations on a table T , where n is an exact power of 2. The ﬁrst n=2 operations are

insertions, which by our previous analysis cost a total of ‚.n/. At the end of this

sequence of insertions, T:num D T:size D n=2. For the second n=2 operations,

we perform the following sequence:

insert, delete, delete, insert, insert, delete, delete, insert, insert, . . . .

The ﬁrst insertion causes the table to expand to size n. The two following deletions

cause the table to contract back to size n=2. Two further insertions cause another

expansion, and so forth. The cost of each expansion and contraction is ‚.n/, and

there are ‚.n/ of them. Thus, the total cost of the n operations is ‚.n 2 /, making

the amortized cost of an operation ‚.n/.

The downside of this strategy is obvious: after expanding the table, we do not

delete enough items to pay for a contraction. Likewise, after contracting the table,

we do not insert enough items to pay for an expansion.

We can improve upon this strategy by allowing the load factor of the table to

drop below 1=2. Speciﬁcally, we continue to double the table size upon inserting

an item into a full table, but we halve the table size when deleting an item causes

the table to become less than 1=4 full, rather than 1=2 full as before. The load

factor of the table is therefore bounded below by the constant 1=4.

Intuitively, we would consider a load factor of 1=2 to be ideal, and the table’s

potential would then be 0. As the load factor deviates from 1=2, the potential

increases so that by the time we expand or contract the table, the table has garnered

sufﬁcient potential to pay for copying all the items into the newly allocated table.

Thus, we will need a potential function that has grown to T:num by the time that

the load factor has either increased to 1 or decreased to 1=4. After either expanding

or contracting the table, the load factor goes back to 1=2 and the table’s potential

reduces back to 0.

We omit the code for TABLE-DELETE, since it is analogous to TABLE-INSERT.

For our analysis, we shall assume that whenever the number of items in the table

drops to 0, we free the storage for the table. That is, if T:num D 0, then T:size D 0.

We can now use the potential method to analyze the cost of a sequence of n

TABLE-INSERT and TABLE-DELETE operations. We start by deﬁning a poten-

tial function ˆ that is 0 immediately after an expansion or contraction and builds

as the load factor increases to 1 or decreases to 1=4. Let us denote the load fac-

17.4 Dynamic tables 469

num

i

Φ i

size

i

0 8 16 24 32 40 48

0

8

16

24

32

i

Figure 17.4 The effect of a sequence of n TABLE-INSERT and TABLE-DELETE operations on the

number numi of items in the table, the number sizei of slots in the table, and the potential

ˆi D

2 numi sizei if ˛i 1=2 ;

sizei =2 numi if ˛i < 1=2 ;

each measured after the ith operation. The thin line shows numi , the dashed line shows sizei , and

the thick line shows ˆi . Notice that immediately before an expansion, the potential has built up to

the number of items in the table, and therefore it can pay for moving all the items to the new table.

Likewise, immediately before a contraction, the potential has built up to the number of items in the

table.

tor of a nonempty table T by ˛.T / D T:num=T:size. Since for an empty table,

T:num D T:size D 0 and ˛.T / D 1, we always have T:num D ˛.T / T:size,

whether the table is empty or not. We shall use as our potential function

ˆ.T / D

(

2 T:num T:size if ˛.T / 1=2 ;

T:size=2 T:num if ˛.T / < 1=2 :

(17.6)

Observe that the potential of an empty table is 0 and that the potential is never

negative. Thus, the total amortized cost of a sequence of operations with respect

to ˆ provides an upper bound on the actual cost of the sequence.

Before proceeding with a precise analysis, we pause to observe some properties

of the potential function, as illustrated in Figure 17.4. Notice that when the load

factor is 1=2, the potential is 0. When the load factor is 1, we have T:size D T:num,

which implies ˆ.T / D T:num, and thus the potential can pay for an expansion if

an item is inserted. When the load factor is 1=4, we have T:size D 4T:num, which

470 Chapter 17 Amortized Analysis

implies ˆ.T / D T:num, and thus the potential can pay for a contraction if an item

is deleted.

To analyze a sequence of n TABLE-INSERT and TABLE-DELETE operations,

we let c i denote the actual cost of the ith operation, yc i denote its amortized cost

with respect to ˆ, num i denote the number of items stored in the table after the ith

operation, size i denote the total size of the table after the ith operation, ˛ i denote

the load factor of the table after the ith operation, and ˆ i denote the potential after

the ith operation. Initially, num 0 D 0, size 0 D 0, ˛ 0 D 1, and ˆ 0 D 0.

We start with the case in which the ith operation is TABLE-INSERT. The analy-

sis is identical to that for table expansion in Section 17.4.1 if ˛ i1 1=2. Whether

the table expands or not, the amortized cost yc i of the operation is at most 3.

If ˛ i1 < 1=2, the table cannot expand as a result of the operation, since the ta-

ble expands only when ˛ i1 D 1. If ˛ i < 1=2 as well, then the amortized cost of

the ith operation is

yc i D c i C ˆ i ˆ i1

D 1 C .size i =2 num i / .size i1 =2 num i1 /

D 1 C .size i =2 num i / .size i =2 .num i 1//

D 0 :

If ˛ i1 < 1=2 but ˛ i 1=2, then

yc i D c i C ˆ i ˆ i1

D 1 C .2 num i size i / .size i1 =2 num i1 /

D 1 C .2.num i1 C 1/ size i1 / .size i1 =2 num i1 /

D 3 num i1

3

2

size i1 C 3

D 3˛ i1 size i1

3

2

size i1 C 3

<

3

2

size i1

3

2

size i1 C 3

D 3 :

Thus, the amortized cost of a TABLE-INSERT operation is at most 3.

We now turn to the case in which the ith operation is TABLE-DELETE. In this

case, num i D num i1 1. If ˛ i1 < 1=2, then we must consider whether the

operation causes the table to contract. If it does not, then size i D size i1 and the

amortized cost of the operation is

yc i D c i C ˆ i ˆ i1

D 1 C .size i =2 num i / .size i1 =2 num i1 /

D 1 C .size i =2 num i / .size i =2 .num i C 1//

D 2 :

17.4 Dynamic tables 471

If ˛ i1 < 1=2 and the ith operation does trigger a contraction, then the actual cost

of the operation is c i D num i C 1, since we delete one item and move num i items.

We have size i =2 D size i1 =4 D num i1 D num i C 1, and the amortized cost of

the operation is

yc i D c i C ˆ i ˆ i1

D .num i C 1/ C .size i =2 num i / .size i1 =2 num i1 /

D .num i C 1/ C ..num i C 1/ num i / ..2 num i C 2/ .num i C 1//

D 1 :

When the ith operation is a TABLE-DELETE and ˛ i1 1=2, the amortized cost

is also bounded above by a constant. We leave the analysis as Exercise 17.4-2.

In summary, since the amortized cost of each operation is bounded above by

a constant, the actual time for any sequence of n operations on a dynamic table

is O.n/.

Exercises

17.4-1

Suppose that we wish to implement a dynamic, open-address hash table. Why

might we consider the table to be full when its load factor reaches some value ˛

that is strictly less than 1? Describe brieﬂy how to make insertion into a dynamic,

open-address hash table run in such a way that the expected value of the amortized

cost per insertion is O.1/. Why is the expected value of the actual cost per insertion

not necessarily O.1/ for all insertions?

17.4-2

Show that if ˛ i1 1=2 and the ith operation on a dynamic table is TABLE-

DELETE, then the amortized cost of the operation with respect to the potential

function (17.6) is bounded above by a constant.

17.4-3

Suppose that instead of contracting a table by halving its size when its load factor

drops below 1=4, we contract it by multiplying its size by 2=3 when its load factor

drops below 1=3. Using the potential function

ˆ.T / D j2 T:num T:sizej ;

show that the amortized cost of a TABLE-DELETE that uses this strategy is bounded

above by a constant.

472 Chapter 17 Amortized Analysis

Problems

17-1 Bit-reversed binary counter

Chapter 30 examines an important algorithm called the fast Fourier transform,

or FFT. The ﬁrst step of the FFT algorithm performs a bit-reversal permutation on

an input array AŒ0 : : n1 whose length is n D 2 k

for some nonnegative integer k.

This permutation swaps elements whose indices have binary representations that

are the reverse of each other.

We can express each index a as a k-bit sequence ha k1 ; a k2 ; : : : ; a 0 i, where

a D

P k1

iD0

a i 2 i

. We deﬁne

rev k .ha k1 ; a k2 ; : : : ; a 0 i/ D ha 0 ; a 1 ; : : : ; a k1 i I

thus,

rev k .a/ D

k1 X

iD0

a ki1 2

i

:

For example, if n D 16 (or, equivalently, k D 4), then rev k .3/ D 12, since

the 4-bit representation of 3 is 0011, which when reversed gives 1100, the 4-bit

representation of 12.

a. Given a function rev k that runs in ‚.k/ time, write an algorithm to perform the

bit-reversal permutation on an array of length n D 2 k

in O.nk/ time.

We can use an algorithm based on an amortized analysis to improve the running

time of the bit-reversal permutation. We maintain a “bit-reversed counter” and a

procedure BIT-REVERSED-INCREMENT that, when given a bit-reversed-counter

value a, produces rev k .rev k .a/ C 1/. If k D 4, for example, and the bit-reversed

counter starts at 0, then successive calls to BIT-REVERSED-INCREMENT produce

the sequence

0000; 1000; 0100; 1100; 0010; 1010; : : : D 0; 8; 4; 12; 2; 10; : : : :

b. Assume that the words in your computer store k-bit values and that in unit time,

your computer can manipulate the binary values with operations such as shifting

left or right by arbitrary amounts, bitwise-AND, bitwise-OR, etc. Describe

an implementation of the BIT-REVERSED-INCREMENT procedure that allows

the bit-reversal permutation on an n-element array to be performed in a total

of O.n/ time.

c. Suppose that you can shift a word left or right by only one bit in unit time. Is it

still possible to implement an O.n/-time bit-reversal permutation?

Problems for Chapter 17 473

17-2 Making binary search dynamic

Binary search of a sorted array takes logarithmic search time, but the time to insert

a new element is linear in the size of the array. We can improve the time for

insertion by keeping several sorted arrays.

Speciﬁcally, suppose that we wish to support SEARCH and INSERT on a set

of n elements. Let k D dlg.n C 1/e, and let the binary representation of n

be hn k1 ; n k2 ; : : : ; n 0 i. We have k sorted arrays A 0 ; A 1 ; : : : ; A k1 , where for

i D 0; 1; : : : ; k 1, the length of array A i is 2 i

. Each array is either full or empty,

depending on whether n i D 1 or n i D 0, respectively. The total number of ele-

ments held in all k arrays is therefore

P k1

iD0

n i 2 i D n. Although each individual

array is sorted, elements in different arrays bear no particular relationship to each

other.

a. Describe how to perform the SEARCH operation for this data structure. Analyze

its worst-case running time.

b. Describe how to perform the INSERT operation. Analyze its worst-case and

amortized running times.

c. Discuss how to implement DELETE.

17-3 Amortized weight-balanced trees

Consider an ordinary binary search tree augmented by adding to each node x the

attribute x:size giving the number of keys stored in the subtree rooted at x. Let ˛

be a constant in the range 1=2 ˛ < 1. We say that a given node x is ˛-balanced

if x:left:size ˛ x:size and x:right:size ˛ x:size. The tree as a whole

is ˛-balanced if every node in the tree is ˛-balanced. The following amortized

approach to maintaining weight-balanced trees was suggested by G. Varghese.

a. A 1=2-balanced tree is, in a sense, as balanced as it can be. Given a node x

in an arbitrary binary search tree, show how to rebuild the subtree rooted at x

so that it becomes 1=2-balanced. Your algorithm should run in time ‚.x:size/,

and it can use O.x:size/ auxiliary storage.

b. Show that performing a search in an n-node ˛-balanced binary search tree

takes O.lg n/ worst-case time.

For the remainder of this problem, assume that the constant ˛ is strictly greater

than 1=2. Suppose that we implement INSERT and DELETE as usual for an n-node

binary search tree, except that after every such operation, if any node in the tree

is no longer ˛-balanced, then we “rebuild” the subtree rooted at the highest such

node in the tree so that it becomes 1=2-balanced.

474 Chapter 17 Amortized Analysis

We shall analyze this rebuilding scheme using the potential method. For a node x

in a binary search tree T , we deﬁne

.x/ D jx:left:size x:right:sizej ;

and we deﬁne the potential of T as

ˆ.T / D c

X

x2T W.x/2

.x/ ;

where c is a sufﬁciently large constant that depends on ˛.

c. Argue that any binary search tree has nonnegative potential and that a 1=2-

balanced tree has potential 0.

d. Suppose that m units of potential can pay for rebuilding an m-node subtree.

How large must c be in terms of ˛ in order for it to take O.1/ amortized time

to rebuild a subtree that is not ˛-balanced?

e. Show that inserting a node into or deleting a node from an n-node ˛-balanced

tree costs O.lg n/ amortized time.

17-4 The cost of restructuring red-black trees

There are four basic operations on red-black trees that perform structural modi-

ﬁcations: node insertions, node deletions, rotations, and color changes. We have

seen that RB-INSERT and RB-DELETE use only O.1/ rotations, node insertions,

and node deletions to maintain the red-black properties, but they may make many

more color changes.

a. Describe a legal red-black tree with n nodes such that calling RB-INSERT to

add the .n C 1/st node causes .lg n/ color changes. Then describe a legal

red-black tree with n nodes for which calling RB-DELETE on a particular node

causes .lg n/ color changes.

Although the worst-case number of color changes per operation can be logarithmic,

we shall prove that any sequence of m RB-INSERT and RB-DELETE operations on

an initially empty red-black tree causes O.m/ structural modiﬁcations in the worst

case. Note that we count each color change as a structural modiﬁcation.

b. Some of the cases handled by the main loop of the code of both RB-INSERT-

FIXUP and RB-DELETE-FIXUP are terminating: once encountered, they cause

the loop to terminate after a constant number of additional operations. For each

of the cases of RB-INSERT-FIXUP and RB-DELETE-FIXUP, specify which are

terminating and which are not. (Hint: Look at Figures 13.5, 13.6, and 13.7.)

Problems for Chapter 17 475

We shall ﬁrst analyze the structural modiﬁcations when only insertions are per-

formed. Let T be a red-black tree, and deﬁne ˆ.T / to be the number of red nodes

in T . Assume that 1 unit of potential can pay for the structural modiﬁcations per-

formed by any of the three cases of RB-INSERT-FIXUP.

c. Let T 0

be the result of applying Case 1 of RB-INSERT-FIXUP to T . Argue that

ˆ.T 0 / D ˆ.T / 1.

d. When we insert a node into a red-black tree using RB-INSERT, we can break

the operation into three parts. List the structural modiﬁcations and potential

changes resulting from lines 1–16 of RB-INSERT, from nonterminating cases

of RB-INSERT-FIXUP, and from terminating cases of RB-INSERT-FIXUP.

e. Using part (d), argue that the amortized number of structural modiﬁcations per-

formed by any call of RB-INSERT is O.1/.

We now wish to prove that there are O.m/ structural modiﬁcations when there are

both insertions and deletions. Let us deﬁne, for each node x,

w.x/ D

„

0 if x is red ;

1 if x is black and has no red children ;

0 if x is black and has one red child ;

2 if x is black and has two red children :

Now we redeﬁne the potential of a red-black tree T as

ˆ.T / D

X

x2T

w.x/ ;

and let T 0

be the tree that results from applying any nonterminating case of RB-

INSERT-FIXUP or RB-DELETE-FIXUP to T .

f. Show that ˆ.T 0 / ˆ.T / 1 for all nonterminating cases of RB-INSERT-

FIXUP. Argue that the amortized number of structural modiﬁcations performed

by any call of RB-INSERT-FIXUP is O.1/.

g. Show that ˆ.T 0 / ˆ.T / 1 for all nonterminating cases of RB-DELETE-

FIXUP. Argue that the amortized number of structural modiﬁcations performed

by any call of RB-DELETE-FIXUP is O.1/.

h. Complete the proof that in the worst case, any sequence of m RB-INSERT and

RB-DELETE operations performs O.m/ structural modiﬁcations.

476 Chapter 17 Amortized Analysis

17-5 Competitive analysis of self-organizing lists with move-to-front

A self-organizing list is a linked list of n elements, in which each element has a

unique key. When we search for an element in the list, we are given a key, and we

want to ﬁnd an element with that key.

A self-organizing list has two important properties:

1. To ﬁnd an element in the list, given its key, we must traverse the list from the

beginning until we encounter the element with the given key. If that element is

the kth element from the start of the list, then the cost to ﬁnd the element is k.

2. We may reorder the list elements after any operation, according to a given rule

with a given cost. We may choose any heuristic we like to decide how to reorder

the list.

Assume that we start with a given list of n elements, and we are given an access

sequence D h 1 ; 2 ; : : : ; m i of keys to ﬁnd, in order. The cost of the sequence

is the sum of the costs of the individual accesses in the sequence.

Out of the various possible ways to reorder the list after an operation, this prob-

lem focuses on transposing adjacent list elements—switching their positions in the

list—with a unit cost for each transpose operation. You will show, by means of a

potential function, that a particular heuristic for reordering the list, move-to-front,

entails a total cost no worse than 4 times that of any other heuristic for maintaining

the list order—even if the other heuristic knows the access sequence in advance!

We call this type of analysis a competitive analysis.

For a heuristic H and a given initial ordering of the list, denote the access cost of

sequence by C H . /. Let m be the number of accesses in .

a. Argue that if heuristic H does not know the access sequence in advance, then

the worst-case cost for H on an access sequence is C H . / D .mn/.

With the move-to-front heuristic, immediately after searching for an element x,

we move x to the ﬁrst position on the list (i.e., the front of the list).

Let rank L .x/ denote the rank of element x in list L, that is, the position of x in

list L. For example, if x is the fourth element in L, then rank L .x/ D 4. Let c i

denote the cost of access i using the move-to-front heuristic, which includes the

cost of ﬁnding the element in the list and the cost of moving it to the front of the

list by a series of transpositions of adjacent list elements.

b. Show that if i accesses element x in list L using the move-to-front heuristic,

then c i D 2 rank L .x/ 1.

Now we compare move-to-front with any other heuristic H that processes an

access sequence according to the two properties above. Heuristic H may transpose

Problems for Chapter 17 477

elements in the list in any way it wants, and it might even know the entire access

sequence in advance.

Let L i be the list after access i using move-to-front, and let L

i

be the list after

access i using heuristic H. We denote the cost of access i by c i for move-to-

front and by c

i

for heuristic H. Suppose that heuristic H performs t

i

transpositions

during access i .

c. In part (b), you showed that c i D 2 rank L i1 .x/ 1. Now show that c

i

D

rank L

i1

.x/ C t

i

.

We deﬁne an inversion in list L i as a pair of elements y and ´ such that y

precedes ´ in L i and ´ precedes y in list L

i

. Suppose that list L i has q i inversions

after processing the access sequence h 1 ; 2 ; : : : ; i i. Then, we deﬁne a potential

function ˆ that maps L i to a real number by ˆ.L i / D 2q i . For example, if L i has

the elements he; c; a; d; bi and L

i

has the elements hc; a; b; d; ei, then L i has 5

inversions (.e; c/; .e; a/; .e; d/; .e; b/; .d; b/), and so ˆ.L i / D 10. Observe that

ˆ.L i / 0 for all i and that, if move-to-front and heuristic H start with the same

list L 0 , then ˆ.L 0 / D 0.

d. Argue that a transposition either increases the potential by 2 or decreases the

potential by 2.

Suppose that access i ﬁnds the element x. To understand how the potential

changes due to i , let us partition the elements other than x into four sets, depend-

ing on where they are in the lists just before the ith access:

Set A consists of elements that precede x in both L i1 and L

i1

.

Set B consists of elements that precede x in L i1 and follow x in L

i1

.

Set C consists of elements that follow x in L i1 and precede x in L

i1

.

Set D consists of elements that follow x in both L i1 and L

i1

.

e. Argue that rank L i1 .x/ D jAj C jBj C 1 and rank L

i1

.x/ D jAj C jC j C 1.

f. Show that access i causes a change in potential of

ˆ.L i / ˆ.L i1 / 2.jAj jBj C t

i

/ ;

where, as before, heuristic H performs t

i

transpositions during access i .

Deﬁne the amortized cost yc i of access i by yc i D c i C ˆ.L i / ˆ.L i1 /.

g. Show that the amortized cost yc i of access i is bounded from above by 4c

i

.

h. Conclude that the cost C MTF . / of access sequence with move-to-front is at

most 4 times the cost C H . / of with any other heuristic H, assuming that

both heuristics start with the same list.

478 Chapter 17 Amortized Analysis

Chapter notes

Aho, Hopcroft, and Ullman [5] used aggregate analysis to determine the running

time of operations on a disjoint-set forest; we shall analyze this data structure us-

ing the potential method in Chapter 21. Tarjan [331] surveys the accounting and

potential methods of amortized analysis and presents several applications. He at-

tributes the accounting method to several authors, including M. R. Brown, R. E.

Tarjan, S. Huddleston, and K. Mehlhorn. He attributes the potential method to

D. D. Sleator. The term “amortized” is due to D. D. Sleator and R. E. Tarjan.

Potential functions are also useful for proving lower bounds for certain types of

problems. For each conﬁguration of the problem, we deﬁne a potential function

that maps the conﬁguration to a real number. Then we determine the potential ˆ init

of the initial conﬁguration, the potential ˆ ﬁnal of the ﬁnal conﬁguration, and the

maximum change in potential ˆ max due to any step. The number of steps must

therefore be at least jˆ ﬁnal ˆ init j = jˆ max j. Examples of potential functions to

prove lower bounds in I/O complexity appear in works by Cormen, Sundquist, and

Wisniewski [79]; Floyd [107]; and Aggarwal and Vitter [3]. Krumme, Cybenko,

and Venkataraman [221] applied potential functions to prove lower bounds on gos-

siping: communicating a unique item from each vertex in a graph to every other

vertex.

The move-to-front heuristic from Problem 17-5 works quite well in practice.

Moreover, if we recognize that when we ﬁnd an element, we can splice it out of its

position in the list and relocate it to the front of the list in constant time, we can

show that the cost of move-to-front is at most twice the cost of any other heuristic

including, again, one that knows the entire access sequence in advance.

V Advanced Data Structures

Introduction

This part returns to studying data structures that support operations on dynamic

sets, but at a more advanced level than Part III. Two of the chapters, for example,

make extensive use of the amortized analysis techniques we saw in Chapter 17.

Chapter 18 presents B-trees, which are balanced search trees speciﬁcally de-

signed to be stored on disks. Because disks operate much more slowly than

random-access memory, we measure the performance of B-trees not only by how

much computing time the dynamic-set operations consume but also by how many

disk accesses they perform. For each B-tree operation, the number of disk accesses

increases with the height of the B-tree, but B-tree operations keep the height low.

Chapter 19 gives an implementation of a mergeable heap, which supports the

operations INSERT, MINIMUM, EXTRACT-MIN, and UNION. 1 The UNION oper-

ation unites, or merges, two heaps. Fibonacci heaps—the data structure in Chap-

ter 19—also support the operations DELETE and DECREASE-KEY. We use amor-

tized time bounds to measure the performance of Fibonacci heaps. The opera-

tions INSERT, MINIMUM, and UNION take only O.1/ actual and amortized time

on Fibonacci heaps, and the operations EXTRACT-MIN and DELETE take O.lg n/

amortized time. The most signiﬁcant advantage of Fibonacci heaps, however, is

that DECREASE-KEY takes only O.1/ amortized time. Because the DECREASE-

1

As in Problem 10-2, we have deﬁned a mergeable heap to support MINIMUM and EXTRACT-MIN,

and so we can also refer to it as a mergeable min-heap. Alternatively, if it supported MAXIMUM

and EXTRACT-MAX, it would be a mergeable max-heap. Unless we specify otherwise, mergeable

heaps will be by default mergeable min-heaps.

482 Part V Advanced Data Structures

KEY operation takes constant amortized time, Fibonacci heaps are key components

of some of the asymptotically fastest algorithms to date for graph problems.

Noting that we can beat the .n lg n/ lower bound for sorting when the keys

are integers in a restricted range, Chapter 20 asks whether we can design a data

structure that supports the dynamic-set operations SEARCH, INSERT, DELETE,

MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in o.lg n/ time when

the keys are integers in a restricted range. The answer turns out to be that we can,

by using a recursive data structure known as a van Emde Boas tree. If the keys are

unique integers drawn from the set f0; 1; 2; : : : ; u 1g, where u is an exact power

of 2, then van Emde Boas trees support each of the above operations in O.lg lg u/

time.

Finally, Chapter 21 presents data structures for disjoint sets. We have a universe

of n elements that are partitioned into dynamic sets. Initially, each element belongs

to its own singleton set. The operation UNION unites two sets, and the query FIND-

SET identiﬁes the unique set that contains a given element at the moment. By

representing each set as a simple rooted tree, we obtain surprisingly fast operations:

a sequence of m operations runs in O.m ˛.n// time, where ˛.n/ is an incredibly

slowly growing function—˛.n/ is at most 4 in any conceivable application. The

amortized analysis that proves this time bound is as complex as the data structure

is simple.

The topics covered in this part are by no means the only examples of “advanced”

data structures. Other advanced data structures include the following:

Dynamic trees, introduced by Sleator and Tarjan [319] and discussed by Tarjan

[330], maintain a forest of disjoint rooted trees. Each edge in each tree has

a real-valued cost. Dynamic trees support queries to ﬁnd parents, roots, edge

costs, and the minimum edge cost on a simple path from a node up to a root.

Trees may be manipulated by cutting edges, updating all edge costs on a simple

path from a node up to a root, linking a root into another tree, and making a

node the root of the tree it appears in. One implementation of dynamic trees

gives an O.lg n/ amortized time bound for each operation; a more complicated

implementation yields O.lg n/ worst-case time bounds. Dynamic trees are used

in some of the asymptotically fastest network-ﬂow algorithms.

Splay trees, developed by Sleator and Tarjan [320] and, again, discussed by

Tarjan [330], are a form of binary search tree on which the standard search-

tree operations run in O.lg n/ amortized time. One application of splay trees

simpliﬁes dynamic trees.

Persistent data structures allow queries, and sometimes updates as well, on past

versions of a data structure. Driscoll, Sarnak, Sleator, and Tarjan [97] present

techniques for making linked data structures persistent with only a small time

Part V Advanced Data Structures 483

and space cost. Problem 13-1 gives a simple example of a persistent dynamic

set.

As in Chapter 20, several data structures allow a faster implementation of dic-

tionary operations (INSERT, DELETE, and SEARCH) for a restricted universe

of keys. By taking advantage of these restrictions, they are able to achieve bet-

ter worst-case asymptotic running times than comparison-based data structures.

Fredman and Willard introduced fusion trees [115], which were the ﬁrst data

structure to allow faster dictionary operations when the universe is restricted to

integers. They showed how to implement these operations in O.lg n= lg lg n/

time. Several subsequent data structures, including exponential search trees

[16], have also given improved bounds on some or all of the dictionary opera-

tions and are mentioned in the chapter notes throughout this book.

Dynamic graph data structures support various queries while allowing the

structure of a graph to change through operations that insert or delete vertices

or edges. Examples of the queries that they support include vertex connectivity

[166], edge connectivity, minimum spanning trees [165], biconnectivity, and

transitive closure [164].

Chapter notes throughout this book mention additional data structures.

18 B-Trees

B-trees are balanced search trees designed to work well on disks or other direct-

access secondary storage devices. B-trees are similar to red-black trees (Chap-

ter 13), but they are better at minimizing disk I/O operations. Many database sys-

tems use B-trees, or variants of B-trees, to store information.

B-trees differ from red-black trees in that B-tree nodes may have many children,

from a few to thousands. That is, the “branching factor” of a B-tree can be quite

large, although it usually depends on characteristics of the disk unit used. B-trees

are similar to red-black trees in that every n-node B-tree has height O.lg n/. The

exact height of a B-tree can be considerably less than that of a red-black tree,

however, because its branching factor, and hence the base of the logarithm that

expresses its height, can be much larger. Therefore, we can also use B-trees to

implement many dynamic-set operations in time O.lg n/.

B-trees generalize binary search trees in a natural manner. Figure 18.1 shows a

simple B-tree. If an internal B-tree node x contains x:n keys, then x has x:n C 1

children. The keys in node x serve as dividing points separating the range of keys

handled by x into x:n C 1 subranges, each handled by one child of x. When

searching for a key in a B-tree, we make an .x:n C 1/-way decision based on

comparisons with the x:n keys stored at node x. The structure of leaf nodes differs

from that of internal nodes; we will examine these differences in Section 18.1.

Section 18.1 gives a precise deﬁnition of B-trees and proves that the height of

a B-tree grows only logarithmically with the number of nodes it contains. Sec-

tion 18.2 describes how to search for a key and insert a key into a B-tree, and

Section 18.3 discusses deletion. Before proceeding, however, we need to ask why

we evaluate data structures designed to work on a disk differently from data struc-

tures designed to work in main random-access memory.

Data structures on secondary storage

Computer systems take advantage of various technologies that provide memory

capacity. The primary memory (or main memory) of a computer system normally

Chapter 18 B-Trees 485

B C F G J K L

D H

N P R S V W Y Z

Q T X

M

T:root

Figure 18.1 A B-tree whose keys are the consonants of English. An internal node x containing

x:n keys has x:n C 1 children. All leaves are at the same depth in the tree. The lightly shaded nodes

are examined in a search for the letter R.

platter track

arms

read/write

head

spindle

Figure 18.2 A typical disk drive. It comprises one or more platters (two platters are shown here)

that rotate around a spindle. Each platter is read and written with a head at the end of an arm. Arms

rotate around a common pivot axis. A track is the surface that passes beneath the read/write head

when the head is stationary.

consists of silicon memory chips. This technology is typically more than an order

of magnitude more expensive per bit stored than magnetic storage technology, such

as tapes or disks. Most computer systems also have secondary storage based on

magnetic disks; the amount of such secondary storage often exceeds the amount of

primary memory by at least two orders of magnitude.

Figure 18.2 shows a typical disk drive. The drive consists of one or more plat-

ters, which rotate at a constant speed around a common spindle. A magnetizable

material covers the surface of each platter. The drive reads and writes each platter

by a head at the end of an arm. The arms can move their heads toward or away

486 Chapter 18 B-Trees

from the spindle. When a given head is stationary, the surface that passes under-

neath it is called a track. Multiple platters increase only the disk drive’s capacity

and not its performance.

Although disks are cheaper and have higher capacity than main memory, they are

much, much slower because they have moving mechanical parts. 1 The mechanical

motion has two components: platter rotation and arm movement. As of this writing,

commodity disks rotate at speeds of 5400–15,000 revolutions per minute (RPM).

We typically see 15,000 RPM speeds in server-grade drives, 7200 RPM speeds

in drives for desktops, and 5400 RPM speeds in drives for laptops. Although

7200 RPM may seem fast, one rotation takes 8.33 milliseconds, which is over 5

orders of magnitude longer than the 50 nanosecond access times (more or less)

commonly found for silicon memory. In other words, if we have to wait a full rota-

tion for a particular item to come under the read/write head, we could access main

memory more than 100,000 times during that span. On average we have to wait

for only half a rotation, but still, the difference in access times for silicon memory

compared with disks is enormous. Moving the arms also takes some time. As of

this writing, average access times for commodity disks are in the range of 8 to 11

milliseconds.

In order to amortize the time spent waiting for mechanical movements, disks

access not just one item but several at a time. Information is divided into a number

of equal-sized pages of bits that appear consecutively within tracks, and each disk

read or write is of one or more entire pages. For a typical disk, a page might be 2 11

to 2 14

bytes in length. Once the read/write head is positioned correctly and the disk

has rotated to the beginning of the desired page, reading or writing a magnetic disk

is entirely electronic (aside from the rotation of the disk), and the disk can quickly

read or write large amounts of data.

Often, accessing a page of information and reading it from a disk takes longer

than examining all the information read. For this reason, in this chapter we shall

look separately at the two principal components of the running time:

the number of disk accesses, and

the CPU (computing) time.

We measure the number of disk accesses in terms of the number of pages of infor-

mation that need to be read from or written to the disk. We note that disk-access

time is not constant—it depends on the distance between the current track and

the desired track and also on the initial rotational position of the disk. We shall

1

As of this writing, solid-state drives have recently come onto the consumer market. Although they

are faster than mechanical disk drives, they cost more per gigabyte and have lower capacities than

mechanical disk drives.

Chapter 18 B-Trees 487

nonetheless use the number of pages read or written as a ﬁrst-order approximation

of the total time spent accessing the disk.

In a typical B-tree application, the amount of data handled is so large that all

the data do not ﬁt into main memory at once. The B-tree algorithms copy selected

pages from disk into main memory as needed and write back onto disk the pages

that have changed. B-tree algorithms keep only a constant number of pages in

main memory at any time; thus, the size of main memory does not limit the size of

B-trees that can be handled.

We model disk operations in our pseudocode as follows. Let x be a pointer to an

object. If the object is currently in the computer’s main memory, then we can refer

to the attributes of the object as usual: x:key, for example. If the object referred to

by x resides on disk, however, then we must perform the operation DISK-READ.x/

to read object x into main memory before we can refer to its attributes. (We as-

sume that if x is already in main memory, then DISK-READ.x/ requires no disk

accesses; it is a “no-op.”) Similarly, the operation DISK-WRITE.x/ is used to save

any changes that have been made to the attributes of object x. That is, the typical

pattern for working with an object is as follows:

x D a pointer to some object

DISK-READ.x/

operations that access and/or modify the attributes of x

DISK-WRITE.x/ // omitted if no attributes of x were changed

other operations that access but do not modify attributes of x

The system can keep only a limited number of pages in main memory at any one

time. We shall assume that the system ﬂushes from main memory pages no longer

in use; our B-tree algorithms will ignore this issue.

Since in most systems the running time of a B-tree algorithm depends primar-

ily on the number of DISK-READ and DISK-WRITE operations it performs, we

typically want each of these operations to read or write as much information as

possible. Thus, a B-tree node is usually as large as a whole disk page, and this size

limits the number of children a B-tree node can have.

For a large B-tree stored on a disk, we often see branching factors between 50

and 2000, depending on the size of a key relative to the size of a page. A large

branching factor dramatically reduces both the height of the tree and the number of

disk accesses required to ﬁnd any key. Figure 18.3 shows a B-tree with a branching

factor of 1001 and height 2 that can store over one billion keys; nevertheless, since

we can keep the root node permanently in main memory, we can ﬁnd any key in

this tree by making at most only two disk accesses.

488 Chapter 18 B-Trees

1000

1001

1000

1001

1000

1001

1000

1001

1000 1000 1000

…

1 node,

1000 keys

1001 nodes,

1,001,000 keys

1,002,001 nodes,

1,002,001,000 keys

…

T:root

Figure 18.3 A B-tree of height 2 containing over one billion keys. Shown inside each node x

is x:n, the number of keys in x. Each internal node and leaf contains 1000 keys. This B-tree has

1001 nodes at depth 1 and over one million leaves at depth 2.

18.1 Deﬁnition of B-trees

To keep things simple, we assume, as we have for binary search trees and red-black

trees, that any “satellite information” associated with a key resides in the same

node as the key. In practice, one might actually store with each key just a pointer to

another disk page containing the satellite information for that key. The pseudocode

in this chapter implicitly assumes that the satellite information associated with a

key, or the pointer to such satellite information, travels with the key whenever the

key is moved from node to node. A common variant on a B-tree, known as a

B

C

-tree, stores all the satellite information in the leaves and stores only keys and

child pointers in the internal nodes, thus maximizing the branching factor of the

internal nodes.

A B-tree T is a rooted tree (whose root is T:root) having the following proper-

ties:

1. Every node x has the following attributes:

a. x:n, the number of keys currently stored in node x,

b. the x:n keys themselves, x:key 1 ; x:key 2 ; : : : ; x:key x: n

, stored in nondecreas-

ing order, so that x:key 1 x:key 2 x:key x: n

,

c. x:leaf , a boolean value that is TRUE if x is a leaf and FALSE if x is an internal

node.

2. Each internal node x also contains x:n C 1 pointers x:c 1 ; x:c 2 ; : : : ; x:c x: nC1 to

its children. Leaf nodes have no children, and so their c i attributes are unde-

ﬁned.

18.1 Deﬁnition of B-trees 489

3. The keys x:key

i separate the ranges of keys stored in each subtree: if k i is any

key stored in the subtree with root x:c i , then

k 1 x:key 1 k 2 x:key 2 x:key x: n

k x: nC1 :

4. All leaves have the same depth, which is the tree’s height h.

5. Nodes have lower and upper bounds on the number of keys they can contain.

We express these bounds in terms of a ﬁxed integer t 2 called the minimum

degree of the B-tree:

a. Every node other than the root must have at least t 1 keys. Every internal

node other than the root thus has at least t children. If the tree is nonempty,

the root must have at least one key.

b. Every node may contain at most 2t 1 keys. Therefore, an internal node

may have at most 2t children. We say that a node is full if it contains exactly

2t 1 keys. 2

The simplest B-tree occurs when t D 2. Every internal node then has either 2,

3, or 4 children, and we have a 2-3-4 tree. In practice, however, much larger values

of t yield B-trees with smaller height.

The height of a B-tree

The number of disk accesses required for most operations on a B-tree is propor-

tional to the height of the B-tree. We now analyze the worst-case height of a B-tree.

Theorem 18.1

If n 1, then for any n-key B-tree T of height h and minimum degree t 2,

h log

t

n C 1

2

:

Proof The root of a B-tree T contains at least one key, and all other nodes contain

at least t 1 keys. Thus, T , whose height is h, has at least 2 nodes at depth 1, at

least 2t nodes at depth 2, at least 2t 2

nodes at depth 3, and so on, until at depth h

it has at least 2t h1

nodes. Figure 18.4 illustrates such a tree for h D 3. Thus, the

2

Another common variant on a B-tree, known as a B

-tree, requires each internal node to be at

least 2=3 full, rather than at least half full, as a B-tree requires.

490 Chapter 18 B-Trees

t – 1

t – 1 t – 1 …

t

t – 1

t

…

1

t – 1

t – 1 t – 1 …

t

t – 1

t – 1 t – 1 …

t

t – 1

t

… t – 1

t – 1 t – 1 …

t

depth

number

of nodes

3 2t2

1

2

0 1

2

2t

T:root

Figure 18.4 A B-tree of height 3 containing a minimum possible number of keys. Shown inside

each node x is x:n.

number n of keys satisﬁes the inequality

n 1 C .t 1/

h X

iD1

2t

i1

D 1 C 2.t 1/

t h 1

t 1

D 2t

h

1 :

By simple algebra, we get t h .n C 1/=2. Taking base-t logarithms of both sides

proves the theorem.

Here we see the power of B-trees, as compared with red-black trees. Although

the height of the tree grows as O.lg n/ in both cases (recall that t is a constant), for

B-trees the base of the logarithm can be many times larger. Thus, B-trees save a

factor of about lg t over red-black trees in the number of nodes examined for most

tree operations. Because we usually have to access the disk to examine an arbitrary

node in a tree, B-trees avoid a substantial number of disk accesses.

Exercises

18.1-1

Why don’t we allow a minimum degree of t D 1?

18.1-2

For what values of t is the tree of Figure 18.1 a legal B-tree?

18.2 Basic operations on B-trees 491

18.1-3

Show all legal B-trees of minimum degree 2 that represent f1; 2; 3; 4; 5g.

18.1-4

As a function of the minimum degree t, what is the maximum number of keys that

can be stored in a B-tree of height h?

18.1-5

Describe the data structure that would result if each black node in a red-black tree

were to absorb its red children, incorporating their children with its own.

18.2 Basic operations on B-trees

In this section, we present the details of the operations B-TREE-SEARCH, B-

TREE-CREATE, and B-TREE-INSERT. In these procedures, we adopt two con-

ventions:

The root of the B-tree is always in main memory, so that we never need to

perform a DISK-READ on the root; we do have to perform a DISK-WRITE of

the root, however, whenever the root node is changed.

Any nodes that are passed as parameters must already have had a DISK-READ

operation performed on them.

The procedures we present are all “one-pass” algorithms that proceed downward

from the root of the tree, without having to back up.

Searching a B-tree

Searching a B-tree is much like searching a binary search tree, except that instead

of making a binary, or “two-way,” branching decision at each node, we make a

multiway branching decision according to the number of the node’s children. More

precisely, at each internal node x, we make an .x:n C 1/-way branching decision.

B-TREE-SEARCH is a straightforward generalization of the TREE-SEARCH pro-

cedure deﬁned for binary search trees. B-TREE-SEARCH takes as input a pointer

to the root node x of a subtree and a key k to be searched for in that subtree. The

top-level call is thus of the form B-TREE-SEARCH.T:root; k/. If k is in the B-tree,

B-TREE-SEARCH returns the ordered pair .y; i/ consisting of a node y and an

index i such that y:key

i D k. Otherwise, the procedure returns NIL.

492 Chapter 18 B-Trees

B-TREE-SEARCH.x; k/

1 i D 1

2 while i x:n and k > x:key

i

3 i D i C 1

4 if i x:n and k == x:key i

5 return .x; i/

6 elseif x:leaf

7 return NIL

8 else DISK-READ.x:c i /

9 return B-TREE-SEARCH.x:c i ; k/

Using a linear-search procedure, lines 1–3 ﬁnd the smallest index i such that

k x:key

i , or else they set i to x:n C 1. Lines 4–5 check to see whether we

have now discovered the key, returning if we have. Otherwise, lines 6–9 either ter-

minate the search unsuccessfully (if x is a leaf) or recurse to search the appropriate

subtree of x, after performing the necessary DISK-READ on that child.

Figure 18.1 illustrates the operation of B-TREE-SEARCH. The procedure exam-

ines the lightly shaded nodes during a search for the key R.

As in the TREE-SEARCH procedure for binary search trees, the nodes encoun-

tered during the recursion form a simple path downward from the root of the

tree. The B-TREE-SEARCH procedure therefore accesses O.h/ D O.log

t

n/ disk

pages, where h is the height of the B-tree and n is the number of keys in the B-tree.

Since x:n < 2t, the while loop of lines 2–3 takes O.t/ time within each node, and

the total CPU time is O.th/ D O.t log

t

n/.

Creating an empty B-tree

To build a B-tree T , we ﬁrst use B-TREE-CREATE to create an empty root node

and then call B-TREE-INSERT to add new keys. Both of these procedures use an

auxiliary procedure ALLOCATE-NODE, which allocates one disk page to be used

as a new node in O.1/ time. We can assume that a node created by ALLOCATE-

NODE requires no DISK-READ, since there is as yet no useful information stored

on the disk for that node.

B-TREE-CREATE.T /

1 x D ALLOCATE-NODE./

2 x:leaf D TRUE

3 x:n D 0

4 DISK-WRITE.x/

5 T:root D x

B-TREE-CREATE requires O.1/ disk operations and O.1/ CPU time.

18.2 Basic operations on B-trees 493

Inserting a key into a B-tree

Inserting a key into a B-tree is signiﬁcantly more complicated than inserting a key

into a binary search tree. As with binary search trees, we search for the leaf position

at which to insert the new key. With a B-tree, however, we cannot simply create

a new leaf node and insert it, as the resulting tree would fail to be a valid B-tree.

Instead, we insert the new key into an existing leaf node. Since we cannot insert a

key into a leaf node that is full, we introduce an operation that splits a full node y

(having 2t 1 keys) around its median key y:key

t into two nodes having only t 1

keys each. The median key moves up into y’s parent to identify the dividing point

between the two new trees. But if y’s parent is also full, we must split it before we

can insert the new key, and thus we could end up splitting full nodes all the way up

the tree.

As with a binary search tree, we can insert a key into a B-tree in a single pass

down the tree from the root to a leaf. To do so, we do not wait to ﬁnd out whether

we will actually need to split a full node in order to do the insertion. Instead, as we

travel down the tree searching for the position where the new key belongs, we split

each full node we come to along the way (including the leaf itself). Thus whenever

we want to split a full node y, we are assured that its parent is not full.

Splitting a node in a B-tree

The procedure B-TREE-SPLIT-CHILD takes as input a nonfull internal node x (as-

sumed to be in main memory) and an index i such that x:c i (also assumed to be in

main memory) is a full child of x. The procedure then splits this child in two and

adjusts x so that it has an additional child. To split a full root, we will ﬁrst make the

root a child of a new empty root node, so that we can use B-TREE-SPLIT-CHILD.

The tree thus grows in height by one; splitting is the only means by which the tree

grows.

Figure 18.5 illustrates this process. We split the full node y D x:c i about its

median key S, which moves up into y’s parent node x. Those keys in y that are

greater than the median key move into a new node ´, which becomes a new child

of x.

494 Chapter 18 B-Trees

R S T Q P U V

N W

… …

R Q P T U V

N W S

… …

x x

T 1 T 1 T 2 T 2 T 3 T 3 T 4 T 4 T 5 T 5 T 6 T 6 T 7 T 7 T 8 T 8

y D x:c i y D x:c i ´ D x:c iC1

x:key i1

x:key i1

x:key i

x:key i

x:key

iC1

Figure 18.5 Splitting a node with t D 4. Node y D x:ci splits into two nodes, y and ´, and the

median key S of y moves up into y’s parent.

B-TREE-SPLIT-CHILD.x; i/

1 ´ D ALLOCATE-NODE./

2 y D x:c i

3 ´:leaf D y:leaf

4 ´:n D t 1

5 for j D 1 to t 1

6 ´:key

j D y:key

j Ct

7 if not y:leaf

8 for j D 1 to t

9 ´:c j D y:c j Ct

10 y:n D t 1

11 for j D x:n C 1 downto i C 1

12 x:c j C1 D x:c j

13 x:c iC1 D ´

14 for j D x:n downto i

15 x:key

j C1 D x:key

j

16 x:key

i D y:key

t

17 x:n D x:n C 1

18 DISK-WRITE.y/

19 DISK-WRITE.´/

20 DISK-WRITE.x/

B-TREE-SPLIT-CHILD works by straightforward “cutting and pasting.” Here, x

is the node being split, and y is x’s ith child (set in line 2). Node y originally has 2t

children (2t 1 keys) but is reduced to t children (t 1 keys) by this operation.

Node ´ takes the t largest children (t 1 keys) from y, and ´ becomes a new child

18.2 Basic operations on B-trees 495

of x, positioned just after y in x’s table of children. The median key of y moves

up to become the key in x that separates y and ´.

Lines 1–9 create node ´ and give it the largest t 1 keys and corresponding t

children of y. Line 10 adjusts the key count for y. Finally, lines 11–17 insert ´ as

a child of x, move the median key from y up to x in order to separate y from ´,

and adjust x’s key count. Lines 18–20 write out all modiﬁed disk pages. The

CPU time used by B-TREE-SPLIT-CHILD is ‚.t/, due to the loops on lines 5–6

and 8–9. (The other loops run for O.t/ iterations.) The procedure performs O.1/

disk operations.

Inserting a key into a B-tree in a single pass down the tree

We insert a key k into a B-tree T of height h in a single pass down the tree, re-

quiring O.h/ disk accesses. The CPU time required is O.th/ D O.t log

t

n/. The

B-TREE-INSERT procedure uses B-TREE-SPLIT-CHILD to guarantee that the re-

cursion never descends to a full node.

B-TREE-INSERT.T; k/

1 r D T:root

2 if r:n == 2t 1

3 s D ALLOCATE-NODE./

4 T:root D s

5 s:leaf D FALSE

6 s:n D 0

7 s:c 1 D r

8 B-TREE-SPLIT-CHILD.s; 1/

9 B-TREE-INSERT-NONFULL.s; k/

10 else B-TREE-INSERT-NONFULL.r; k/

Lines 3–9 handle the case in which the root node r is full: the root splits and a

new node s (having two children) becomes the root. Splitting the root is the only

way to increase the height of a B-tree. Figure 18.6 illustrates this case. Unlike a

binary search tree, a B-tree increases in height at the top instead of at the bottom.

The procedure ﬁnishes by calling B-TREE-INSERT-NONFULL to insert key k into

the tree rooted at the nonfull root node. B-TREE-INSERT-NONFULL recurses as

necessary down the tree, at all times guaranteeing that the node to which it recurses

is not full by calling B-TREE-SPLIT-CHILD as necessary.

The auxiliary recursive procedure B-TREE-INSERT-NONFULL inserts key k into

node x, which is assumed to be nonfull when the procedure is called. The operation

of B-TREE-INSERT and the recursive operation of B-TREE-INSERT-NONFULL

guarantee that this assumption is true.

496 Chapter 18 B-Trees

T

8

T

7

T

6

T

5

T

4

T

3

T

2

T

1

T

8

T

7

T

6

T

5

T

4

T

3

T

2

T

1

F H L D A N P F D A L N P

s

H

r r

T:root

T:root

Figure 18.6 Splitting the root with t D 4. Root node r splits in two, and a new root node s is

created. The new root contains the median key of r and has the two halves of r as children. The

B-tree grows in height by one when the root is split.

B-TREE-INSERT-NONFULL.x; k/

1 i D x:n

2 if x:leaf

3 while i 1 and k < x:key

i

4 x:key

iC1 D x:key

i

5 i D i 1

6 x:key

iC1 D k

7 x:n D x:n C 1

8 DISK-WRITE.x/

9 else while i 1 and k < x:key i

10 i D i 1

11 i D i C 1

12 DISK-READ.x:c i /

13 if x:c i :n == 2t 1

14 B-TREE-SPLIT-CHILD.x; i/

15 if k > x:key

i

16 i D i C 1

17 B-TREE-INSERT-NONFULL.x:c i ; k/

The B-TREE-INSERT-NONFULL procedure works as follows. Lines 3–8 handle

the case in which x is a leaf node by inserting key k into x. If x is not a leaf

node, then we must insert k into the appropriate leaf node in the subtree rooted

at internal node x. In this case, lines 9–11 determine the child of x to which the

recursion descends. Line 13 detects whether the recursion would descend to a full

child, in which case line 14 uses B-TREE-SPLIT-CHILD to split that child into two

nonfull children, and lines 15–16 determine which of the two children is now the

18.2 Basic operations on B-trees 497

correct one to descend to. (Note that there is no need for a DISK-READ.x:c i / after

line 16 increments i, since the recursion will descend in this case to a child that

was just created by B-TREE-SPLIT-CHILD.) The net effect of lines 13–16 is thus

to guarantee that the procedure never recurses to a full node. Line 17 then recurses

to insert k into the appropriate subtree. Figure 18.7 illustrates the various cases of

inserting into a B-tree.

For a B-tree of height h, B-TREE-INSERT performs O.h/ disk accesses, since

only O.1/ DISK-READ and DISK-WRITE operations occur between calls to

B-TREE-INSERT-NONFULL. The total CPU time used is O.th/ D O.t log

t

n/.

Since B-TREE-INSERT-NONFULL is tail-recursive, we can alternatively imple-

ment it as a while loop, thereby demonstrating that the number of pages that need

to be in main memory at any time is O.1/.

Exercises

18.2-1

Show the results of inserting the keys

F; S; Q; K; C; L; H; T; V; W; M; R; N; P; A; B; X; Y; D; Z; E

in order into an empty B-tree with minimum degree 2. Draw only the conﬁgura-

tions of the tree just before some node must split, and also draw the ﬁnal conﬁgu-

ration.

18.2-2

Explain under what circumstances, if any, redundant DISK-READ or DISK-WRITE

operations occur during the course of executing a call to B-TREE-INSERT. (A

redundant DISK-READ is a DISK-READ for a page that is already in memory.

A redundant DISK-WRITE writes to disk a page of information that is identical to

what is already stored there.)

18.2-3

Explain how to ﬁnd the minimum key stored in a B-tree and how to ﬁnd the prede-

cessor of a given key stored in a B-tree.

18.2-4 ?

Suppose that we insert the keys f1; 2; : : : ; ng into an empty B-tree with minimum

degree 2. How many nodes does the ﬁnal B-tree have?

18.2-5

Since leaf nodes require no pointers to children, they could conceivably use a dif-

ferent (larger) t value than internal nodes for the same disk page size. Show how

to modify the procedures for creating and inserting into a B-tree to handle this

variation.

498 Chapter 18 B-Trees

J K N O R S T D E C A U V Y Z

P X M G (a)

J K N O R S T D E B A U V Y Z

P X M G (b)

C

J K N O D E B A U V Y Z

P X M G (c)

C R S Q

T

J K N O D E B A U V Y Z

M G

(d)

C R S Q L

P

X T

J K N O D E B A U V Y Z

M G

(e)

C

R S Q L

P

X T

F

Q inserted

L inserted

F inserted

initial tree

B inserted

Figure 18.7 Inserting keys into a B-tree. The minimum degree t for this B-tree is 3, so a node can

hold at most 5 keys. Nodes that are modiﬁed by the insertion process are lightly shaded. (a) The

initial tree for this example. (b) The result of inserting B into the initial tree; this is a simple insertion

into a leaf node. (c) The result of inserting Q into the previous tree. The node RST U V splits into

two nodes containing RS and U V , the key T moves up to the root, and Q is inserted in the leftmost

of the two halves (the RS node). (d) The result of inserting L into the previous tree. The root

splits right away, since it is full, and the B-tree grows in height by one. Then L is inserted into the

leaf containing JK. (e) The result of inserting F into the previous tree. The node ABCDE splits

before F is inserted into the rightmost of the two halves (the DE node).

18.3 Deleting a key from a B-tree 499

18.2-6

Suppose that we were to implement B-TREE-SEARCH to use binary search rather

than linear search within each node. Show that this change makes the CPU time

required O.lg n/, independently of how t might be chosen as a function of n.

18.2-7

Suppose that disk hardware allows us to choose the size of a disk page arbitrarily,

but that the time it takes to read the disk page is aCbt, where a and b are speciﬁed

constants and t is the minimum degree for a B-tree using pages of the selected size.

Describe how to choose t so as to minimize (approximately) the B-tree search time.

Suggest an optimal value of t for the case in which a D 5 milliseconds and b D 10

microseconds.

18.3 Deleting a key from a B-tree

Deletion from a B-tree is analogous to insertion but a little more complicated, be-

cause we can delete a key from any node—not just a leaf—and when we delete a

key from an internal node, we will have to rearrange the node’s children. As in

insertion, we must guard against deletion producing a tree whose structure violates

the B-tree properties. Just as we had to ensure that a node didn’t get too big due to

insertion, we must ensure that a node doesn’t get too small during deletion (except

that the root is allowed to have fewer than the minimum number t 1 of keys).

Just as a simple insertion algorithm might have to back up if a node on the path

to where the key was to be inserted was full, a simple approach to deletion might

have to back up if a node (other than the root) along the path to where the key is to

be deleted has the minimum number of keys.

The procedure B-TREE-DELETE deletes the key k from the subtree rooted at x.

We design this procedure to guarantee that whenever it calls itself recursively on a

node x, the number of keys in x is at least the minimum degree t. Note that this

condition requires one more key than the minimum required by the usual B-tree

conditions, so that sometimes a key may have to be moved into a child node before

recursion descends to that child. This strengthened condition allows us to delete a

key from the tree in one downward pass without having to “back up” (with one ex-

ception, which we’ll explain). You should interpret the following speciﬁcation for

deletion from a B-tree with the understanding that if the root node x ever becomes

an internal node having no keys (this situation can occur in cases 2c and 3b on

pages 501–502), then we delete x, and x’s only child x:c 1 becomes the new root

of the tree, decreasing the height of the tree by one and preserving the property that

the root of the tree contains at least one key (unless the tree is empty).

500 Chapter 18 B-Trees

J K N O D E B A U V Y Z

M G

(a)

C

R S Q L

P

X T

F

initial tree

J K N O D E B A U V Y Z

M G

(b)

C

R S Q L

P

X T

F deleted: case 1

J K N O D E B A U V Y Z

G

(c)

C

R S Q

L

P

X T

M deleted: case 2a

J K N O D E B A U V Y Z

(d)

C

R S Q

L

P

X T

G deleted: case 2c

Figure 18.8 Deleting keys from a B-tree. The minimum degree for this B-tree is t D 3, so a node

(other than the root) cannot have fewer than 2 keys. Nodes that are modiﬁed are lightly shaded.

(a) The B-tree of Figure 18.7(e). (b) Deletion of F . This is case 1: simple deletion from a leaf.

(c) Deletion of M. This is case 2a: the predecessor L of M moves up to take M’s position. (d) Dele-

tion of G. This is case 2c: we push G down to make node DEGJK and then delete G from this leaf

(case 1).

We sketch how deletion works instead of presenting the pseudocode. Figure 18.8

illustrates the various cases of deleting keys from a B-tree.

1. If the key k is in node x and x is a leaf, delete the key k from x.

2. If the key k is in node x and x is an internal node, do the following:

18.3 Deleting a key from a B-tree 501

J K N O E B A U V Y Z

(e)

C

R S Q

L P X T

D deleted: case 3b

J K N O E B A U V Y Z

C

R S Q

L P X T

J K N O A U V Y Z C R S Q

L P X T (f) B deleted: case 3a E

(e ′ ) tree shrinks

in height

Figure 18.8, continued (e) Deletion of D. This is case 3b: the recursion cannot descend to

node CL because it has only 2 keys, so we push P down and merge it with CL and TX to form

CLP TX; then we delete D from a leaf (case 1). (e

0

) After (e), we delete the root and the tree shrinks

in height by one. (f) Deletion of B. This is case 3a: C moves to ﬁll B’s position and E moves to

ﬁll C ’s position.

a. If the child y that precedes k in node x has at least t keys, then ﬁnd the

predecessor k 0

of k in the subtree rooted at y. Recursively delete k 0

, and

replace k by k 0

in x. (We can ﬁnd k 0

and delete it in a single downward

pass.)

b. If y has fewer than t keys, then, symmetrically, examine the child ´ that

follows k in node x. If ´ has at least t keys, then ﬁnd the successor k 0

of k in

the subtree rooted at ´. Recursively delete k 0

, and replace k by k 0

in x. (We

can ﬁnd k 0

and delete it in a single downward pass.)

c. Otherwise, if both y and ´ have only t 1 keys, merge k and all of ´ into y,

so that x loses both k and the pointer to ´, and y now contains 2t 1 keys.

Then free ´ and recursively delete k from y.

3. If the key k is not present in internal node x, determine the root x:c i of the

appropriate subtree that must contain k, if k is in the tree at all. If x:c i has

only t 1 keys, execute step 3a or 3b as necessary to guarantee that we descend

to a node containing at least t keys. Then ﬁnish by recursing on the appropriate

child of x.

502 Chapter 18 B-Trees

a. If x:c i has only t 1 keys but has an immediate sibling with at least t keys,

give x:c i an extra key by moving a key from x down into x:c i , moving a

key from x:c i ’s immediate left or right sibling up into x, and moving the

appropriate child pointer from the sibling into x:c i .

b. If x:c i and both of x:c i ’s immediate siblings have t 1 keys, merge x:c i

with one sibling, which involves moving a key from x down into the new

merged node to become the median key for that node.

Since most of the keys in a B-tree are in the leaves, we may expect that in

practice, deletion operations are most often used to delete keys from leaves. The

B-TREE-DELETE procedure then acts in one downward pass through the tree,

without having to back up. When deleting a key in an internal node, however,

the procedure makes a downward pass through the tree but may have to return to

the node from which the key was deleted to replace the key with its predecessor or

successor (cases 2a and 2b).

Although this procedure seems complicated, it involves only O.h/ disk oper-

ations for a B-tree of height h, since only O.1/ calls to DISK-READ and DISK-

WRITE are made between recursive invocations of the procedure. The CPU time

required is O.th/ D O.t log

t

n/.

Exercises

18.3-1

Show the results of deleting C , P , and V , in order, from the tree of Figure 18.8(f).

18.3-2

Write pseudocode for B-TREE-DELETE.

Problems

18-1 Stacks on secondary storage

Consider implementing a stack in a computer that has a relatively small amount

of fast primary memory and a relatively large amount of slower disk storage. The

operations PUSH and POP work on single-word values. The stack we wish to

support can grow to be much larger than can ﬁt in memory, and thus most of it

must be stored on disk.

A simple, but inefﬁcient, stack implementation keeps the entire stack on disk.

We maintain in memory a stack pointer, which is the disk address of the top element

on the stack. If the pointer has value p, the top element is the .p mod m/th word

on page bp=mc of the disk, where m is the number of words per page.

Problems for Chapter 18 503

To implement the PUSH operation, we increment the stack pointer, read the ap-

propriate page into memory from disk, copy the element to be pushed to the ap-

propriate word on the page, and write the page back to disk. A POP operation is

similar. We decrement the stack pointer, read in the appropriate page from disk,

and return the top of the stack. We need not write back the page, since it was not

modiﬁed.

Because disk operations are relatively expensive, we count two costs for any

implementation: the total number of disk accesses and the total CPU time. Any

disk access to a page of m words incurs charges of one disk access and ‚.m/ CPU

time.

a. Asymptotically, what is the worst-case number of disk accesses for n stack

operations using this simple implementation? What is the CPU time for n stack

operations? (Express your answer in terms of m and n for this and subsequent

parts.)

Now consider a stack implementation in which we keep one page of the stack in

memory. (We also maintain a small amount of memory to keep track of which page

is currently in memory.) We can perform a stack operation only if the relevant disk

page resides in memory. If necessary, we can write the page currently in memory

to the disk and read in the new page from the disk to memory. If the relevant disk

page is already in memory, then no disk accesses are required.

b. What is the worst-case number of disk accesses required for n PUSH opera-

tions? What is the CPU time?

c. What is the worst-case number of disk accesses required for n stack operations?

What is the CPU time?

Suppose that we now implement the stack by keeping two pages in memory (in

addition to a small number of words for bookkeeping).

d. Describe how to manage the stack pages so that the amortized number of disk

accesses for any stack operation is O.1=m/ and the amortized CPU time for

any stack operation is O.1/.

18-2 Joining and splitting 2-3-4 trees

The join operation takes two dynamic sets S 0

and S 00

and an element x such that

for any x 0 2 S 0

and x 00 2 S 00

, we have x 0 :key < x:key < x 00 :key. It returns a set

S D S 0 [ fxg [ S 00

. The split operation is like an “inverse” join: given a dynamic

set S and an element x 2 S, it creates a set S 0

that consists of all elements in

S fxg whose keys are less than x:key and a set S 00

that consists of all elements

in S fxg whose keys are greater than x:key. In this problem, we investigate

504 Chapter 18 B-Trees

how to implement these operations on 2-3-4 trees. We assume for convenience that

elements consist only of keys and that all key values are distinct.

a. Show how to maintain, for every node x of a 2-3-4 tree, the height of the subtree

rooted at x as an attribute x:height. Make sure that your implementation does

not affect the asymptotic running times of searching, insertion, and deletion.

b. Show how to implement the join operation. Given two 2-3-4 trees T 0

and T 00

and a key k, the join operation should run in O.1 C jh 0 h 00 j/ time, where h 0

and h 00

are the heights of T 0

and T 00

, respectively.

c. Consider the simple path p from the root of a 2-3-4 tree T to a given key k,

the set S 0

of keys in T that are less than k, and the set S 00

of keys in T that are

greater than k. Show that p breaks S 0

into a set of trees fT 0

0

; T 0

1

; : : : ; T 0

m

g and a

set of keys fk 0

1

; k 0

2

; : : : ; k 0

m

g, where, for i D 1; 2; : : : ; m, we have y < k 0

i

< ´

for any keys y 2 T 0

i1

and ´ 2 T 0

i

. What is the relationship between the heights

of T 0

i1

and T 0

i

? Describe how p breaks S 00

into sets of trees and keys.

d. Show how to implement the split operation on T . Use the join operation to

assemble the keys in S 0

into a single 2-3-4 tree T 0

and the keys in S 00

into a

single 2-3-4 tree T 00

. The running time of the split operation should be O.lg n/,

where n is the number of keys in T . (Hint: The costs for joining should tele-

scope.)

Chapter notes

Knuth [211], Aho, Hopcroft, and Ullman [5], and Sedgewick [306] give further

discussions of balanced-tree schemes and B-trees. Comer [74] provides a compre-

hensive survey of B-trees. Guibas and Sedgewick [155] discuss the relationships

among various kinds of balanced-tree schemes, including red-black trees and 2-3-4

trees.

In 1970, J. E. Hopcroft invented 2-3 trees, a precursor to B-trees and 2-3-4

trees, in which every internal node has either two or three children. Bayer and

McCreight [35] introduced B-trees in 1972; they did not explain their choice of

name.

Bender, Demaine, and Farach-Colton [40] studied how to make B-trees perform

well in the presence of memory-hierarchy effects. Their cache-oblivious algo-

rithms work efﬁciently without explicitly knowing the data transfer sizes within

the memory hierarchy.

19 Fibonacci Heaps

The Fibonacci heap data structure serves a dual purpose. First, it supports a set of

operations that constitutes what is known as a “mergeable heap.” Second, several

Fibonacci-heap operations run in constant amortized time, which makes this data

structure well suited for applications that invoke these operations frequently.

Mergeable heaps

A mergeable heap is any data structure that supports the following ﬁve operations,

in which each element has a key:

MAKE-HEAP./ creates and returns a new heap containing no elements.

INSERT.H; x/ inserts element x, whose key has already been ﬁlled in, into heap H.

MINIMUM.H/ returns a pointer to the element in heap H whose key is minimum.

EXTRACT-MIN.H/ deletes the element from heap H whose key is minimum, re-

turning a pointer to the element.

UNION.H 1 ; H 2 / creates and returns a new heap that contains all the elements of

heaps H 1 and H 2 . Heaps H 1 and H 2 are “destroyed” by this operation.

In addition to the mergeable-heap operations above, Fibonacci heaps also support

the following two operations:

DECREASE-KEY.H; x; k/ assigns to element x within heap H the new key

value k, which we assume to be no greater than its current key value. 1

DELETE.H; x/ deletes element x from heap H.

1

As mentioned in the introduction to Part V, our default mergeable heaps are mergeable min-

heaps, and so the operations MINIMUM, EXTRACT-MIN, and DECREASE-KEY apply. Alterna-

tively, we could deﬁne a mergeable max-heap with the operations MAXIMUM, EXTRACT-MAX,

and INCREASE-KEY.

506 Chapter 19 Fibonacci Heaps

Binary heap Fibonacci heap

Procedure (worst-case) (amortized)

MAKE-HEAP ‚.1/ ‚.1/

INSERT ‚.lg n/ ‚.1/

MINIMUM ‚.1/ ‚.1/

EXTRACT-MIN ‚.lg n/ O.lg n/

UNION ‚.n/ ‚.1/

DECREASE-KEY ‚.lg n/ ‚.1/

DELETE ‚.lg n/ O.lg n/

Figure 19.1 Running times for operations on two implementations of mergeable heaps. The num-

ber of items in the heap(s) at the time of an operation is denoted by n.

As the table in Figure 19.1 shows, if we don’t need the UNION operation, ordi-

nary binary heaps, as used in heapsort (Chapter 6), work fairly well. Operations

other than UNION run in worst-case time O.lg n/ on a binary heap. If we need

to support the UNION operation, however, binary heaps perform poorly. By con-

catenating the two arrays that hold the binary heaps to be merged and then running

BUILD-MIN-HEAP (see Section 6.3), the UNION operation takes ‚.n/ time in the

worst case.

Fibonacci heaps, on the other hand, have better asymptotic time bounds than

binary heaps for the INSERT, UNION, and DECREASE-KEY operations, and they

have the same asymptotic running times for the remaining operations. Note, how-

ever, that the running times for Fibonacci heaps in Figure 19.1 are amortized time

bounds, not worst-case per-operation time bounds. The UNION operation takes

only constant amortized time in a Fibonacci heap, which is signiﬁcantly better

than the linear worst-case time required in a binary heap (assuming, of course, that

an amortized time bound sufﬁces).

Fibonacci heaps in theory and practice

From a theoretical standpoint, Fibonacci heaps are especially desirable when the

number of EXTRACT-MIN and DELETE operations is small relative to the number

of other operations performed. This situation arises in many applications. For

example, some algorithms for graph problems may call DECREASE-KEY once per

edge. For dense graphs, which have many edges, the ‚.1/ amortized time of each

call of DECREASE-KEY adds up to a big improvement over the ‚.lg n/ worst-case

time of binary heaps. Fast algorithms for problems such as computing minimum

spanning trees (Chapter 23) and ﬁnding single-source shortest paths (Chapter 24)

make essential use of Fibonacci heaps.

19.1 Structure of Fibonacci heaps 507

From a practical point of view, however, the constant factors and program-

ming complexity of Fibonacci heaps make them less desirable than ordinary binary

(or k-ary) heaps for most applications, except for certain applications that manage

large amounts of data. Thus, Fibonacci heaps are predominantly of theoretical in-

terest. If a much simpler data structure with the same amortized time bounds as

Fibonacci heaps were developed, it would be of practical use as well.

Both binary heaps and Fibonacci heaps are inefﬁcient in how they support the

operation SEARCH; it can take a while to ﬁnd an element with a given key. For this

reason, operations such as DECREASE-KEY and DELETE that refer to a given ele-

ment require a pointer to that element as part of their input. As in our discussion of

priority queues in Section 6.5, when we use a mergeable heap in an application, we

often store a handle to the corresponding application object in each mergeable-heap

element, as well as a handle to the corresponding mergeable-heap element in each

application object. The exact nature of these handles depends on the application

and its implementation.

Like several other data structures that we have seen, Fibonacci heaps are based

on rooted trees. We represent each element by a node within a tree, and each

node has a key attribute. For the remainder of this chapter, we shall use the term

“node” instead of “element.” We shall also ignore issues of allocating nodes prior

to insertion and freeing nodes following deletion, assuming instead that the code

calling the heap procedures deals with these details.

Section 19.1 deﬁnes Fibonacci heaps, discusses how we represent them, and

presents the potential function used for their amortized analysis. Section 19.2

shows how to implement the mergeable-heap operations and achieve the amortized

time bounds shown in Figure 19.1. The remaining two operations, DECREASE-

KEY and DELETE, form the focus of Section 19.3. Finally, Section 19.4 ﬁnishes a

key part of the analysis and also explains the curious name of the data structure.

19.1 Structure of Fibonacci heaps

A Fibonacci heap is a collection of rooted trees that are min-heap ordered. That

is, each tree obeys the min-heap property: the key of a node is greater than or equal

to the key of its parent. Figure 19.2(a) shows an example of a Fibonacci heap.

As Figure 19.2(b) shows, each node x contains a pointer x:p to its parent and

a pointer x:child to any one of its children. The children of x are linked together

in a circular, doubly linked list, which we call the child list of x. Each child y in

a child list has pointers y:left and y:right that point to y’s left and right siblings,

respectively. If node y is an only child, then y:left D y:right D y. Siblings may

appear in a child list in any order.

508 Chapter 19 Fibonacci Heaps

17

30 26 46

35

24

18 52 38

3

39 41

23 7

17

30 26 46

35

24

18 52 38

3

39 41

23 7

(a)

(b)

H:min

H:min

Figure 19.2 (a) A Fibonacci heap consisting of ﬁve min-heap-ordered trees and 14 nodes. The

dashed line indicates the root list. The minimum node of the heap is the node containing the key 3.

Black nodes are marked. The potential of this particular Fibonacci heap is 5C23 D 11. (b) A more

complete representation showing pointers p (up arrows), child (down arrows), and left and right

(sideways arrows). The remaining ﬁgures in this chapter omit these details, since all the information

shown here can be determined from what appears in part (a).

Circular, doubly linked lists (see Section 10.2) have two advantages for use in

Fibonacci heaps. First, we can insert a node into any location or remove a node

from anywhere in a circular, doubly linked list in O.1/ time. Second, given two

such lists, we can concatenate them (or “splice” them together) into one circular,

doubly linked list in O.1/ time. In the descriptions of Fibonacci heap operations,

we shall refer to these operations informally, letting you ﬁll in the details of their

implementations if you wish.

Each node has two other attributes. We store the number of children in the child

list of node x in x:degree. The boolean-valued attribute x:mark indicates whether

node x has lost a child since the last time x was made the child of another node.

Newly created nodes are unmarked, and a node x becomes unmarked whenever it

is made the child of another node. Until we look at the DECREASE-KEY operation

in Section 19.3, we will just set all mark attributes to FALSE.

We access a given Fibonacci heap H by a pointer H:min to the root of a tree

containing the minimum key; we call this node the minimum node of the Fibonacci

19.1 Structure of Fibonacci heaps 509

heap. If more than one root has a key with the minimum value, then any such root

may serve as the minimum node. When a Fibonacci heap H is empty, H:min

is NIL.

The roots of all the trees in a Fibonacci heap are linked together using their

left and right pointers into a circular, doubly linked list called the root list of the

Fibonacci heap. The pointer H:min thus points to the node in the root list whose

key is minimum. Trees may appear in any order within a root list.

We rely on one other attribute for a Fibonacci heap H: H:n, the number of

nodes currently in H.

Potential function

As mentioned, we shall use the potential method of Section 17.3 to analyze the

performance of Fibonacci heap operations. For a given Fibonacci heap H, we

indicate by t.H/ the number of trees in the root list of H and by m.H/ the number

of marked nodes in H. We then deﬁne the potential ˆ.H/ of Fibonacci heap H

by

ˆ.H/ D t.H/ C 2 m.H/ : (19.1)

(We will gain some intuition for this potential function in Section 19.3.) For exam-

ple, the potential of the Fibonacci heap shown in Figure 19.2 is 5C 23 D 11. The

potential of a set of Fibonacci heaps is the sum of the potentials of its constituent

Fibonacci heaps. We shall assume that a unit of potential can pay for a constant

amount of work, where the constant is sufﬁciently large to cover the cost of any of

the speciﬁc constant-time pieces of work that we might encounter.

We assume that a Fibonacci heap application begins with no heaps. The initial

potential, therefore, is 0, and by equation (19.1), the potential is nonnegative at

all subsequent times. From equation (17.3), an upper bound on the total amortized

cost provides an upper bound on the total actual cost for the sequence of operations.

Maximum degree

The amortized analyses we shall perform in the remaining sections of this chapter

assume that we know an upper bound D.n/ on the maximum degree of any node

in an n-node Fibonacci heap. We won’t prove it, but when only the mergeable-

heap operations are supported, D.n/ blg nc. (Problem 19-2(d) asks you to prove

this property.) In Sections 19.3 and 19.4, we shall show that when we support

DECREASE-KEY and DELETE as well, D.n/ D O.lg n/.

510 Chapter 19 Fibonacci Heaps

19.2 Mergeable-heap operations

The mergeable-heap operations on Fibonacci heaps delay work as long as possible.

The various operations have performance trade-offs. For example, we insert a node

by adding it to the root list, which takes just constant time. If we were to start

with an empty Fibonacci heap and then insert k nodes, the Fibonacci heap would

consist of just a root list of k nodes. The trade-off is that if we then perform

an EXTRACT-MIN operation on Fibonacci heap H, after removing the node that

H:min points to, we would have to look through each of the remaining k 1 nodes

in the root list to ﬁnd the new minimum node. As long as we have to go through

the entire root list during the EXTRACT-MIN operation, we also consolidate nodes

into min-heap-ordered trees to reduce the size of the root list. We shall see that, no

matter what the root list looks like before a EXTRACT-MIN operation, afterward

each node in the root list has a degree that is unique within the root list, which leads

to a root list of size at most D.n/ C 1.

Creating a new Fibonacci heap

To make an empty Fibonacci heap, the MAKE-FIB-HEAP procedure allocates and

returns the Fibonacci heap object H, where H:n D 0 and H:min D NIL; there

are no trees in H. Because t.H/ D 0 and m.H/ D 0, the potential of the empty

Fibonacci heap is ˆ.H/ D 0. The amortized cost of MAKE-FIB-HEAP is thus

equal to its O.1/ actual cost.

Inserting a node

The following procedure inserts node x into Fibonacci heap H, assuming that the

node has already been allocated and that x:key has already been ﬁlled in.

FIB-HEAP-INSERT.H; x/

1 x:degree D 0

2 x:p D NIL

3 x:child D NIL

4 x:mark D FALSE

5 if H:min == NIL

6 create a root list for H containing just x

7 H:min D x

8 else insert x into H’s root list

9 if x:key < H:min:key

10 H:min D x

11 n D n C 1 H: H:

19.2 Mergeable-heap operations 511

(a) (b)

17

30

24 23

26

35

46

7 21

18 52 38

39 41

3 17

30

24 23

26

35

46

7

18 52 38

39 41

3

H:min H:min

Figure 19.3 Inserting a node into a Fibonacci heap. (a) A Fibonacci heap H. (b) Fibonacci heap H

after inserting the node with key 21. The node becomes its own min-heap-ordered tree and is then

added to the root list, becoming the left sibling of the root.

Lines 1–4 initialize some of the structural attributes of node x. Line 5 tests to see

whether Fibonacci heap H is empty. If it is, then lines 6–7 make x be the only

node in H’s root list and set H:min to point to x. Otherwise, lines 8–10 insert x

into H’s root list and update H:min if necessary. Finally, line 11 increments H:n

to reﬂect the addition of the new node. Figure 19.3 shows a node with key 21

inserted into the Fibonacci heap of Figure 19.2.

To determine the amortized cost of FIB-HEAP-INSERT, let H be the input Fi-

bonacci heap and H 0

be the resulting Fibonacci heap. Then, t.H 0 / D t.H/ C 1

and m.H 0 / D m.H/, and the increase in potential is

..t.H/ C 1/ C 2 m.H// .t.H/ C 2 m.H// D 1 :

Since the actual cost is O.1/, the amortized cost is O.1/ C 1 D O.1/.

Finding the minimum node

The minimum node of a Fibonacci heap H is given by the pointer H:min, so we

can ﬁnd the minimum node in O.1/ actual time. Because the potential of H does

not change, the amortized cost of this operation is equal to its O.1/ actual cost.

Uniting two Fibonacci heaps

The following procedure unites Fibonacci heaps H 1 and H 2 , destroying H 1 and H 2

in the process. It simply concatenates the root lists of H 1 and H 2 and then deter-

mines the new minimum node. Afterward, the objects representing H 1 and H 2 will

never be used again.

512 Chapter 19 Fibonacci Heaps

FIB-HEAP-UNION.H 1 ; H 2 /

1 H D MAKE-FIB-HEAP./

2 H:min D H 1 :min

3 concatenate the root list of H 2 with the root list of H

4 if .H 1 :min == NIL/ or .H 2 :min ¤ NIL and H 2 :min:key < H 1 :min:key/

5 H:min D H 2 :min

6 H:n D H 1 :n C H 2 :n

7 return H

Lines 1–3 concatenate the root lists of H 1 and H 2 into a new root list H. Lines

2, 4, and 5 set the minimum node of H, and line 6 sets H:n to the total number

of nodes. Line 7 returns the resulting Fibonacci heap H. As in the FIB-HEAP-

INSERT procedure, all roots remain roots.

The change in potential is

ˆ.H/ .ˆ.H 1 / C ˆ.H 2 //

D .t.H/ C 2 m.H// ..t.H 1 / C 2 m.H 1 // C .t.H 2 / C 2 m.H 2 ///

D 0 ;

because t.H/ D t.H 1 / C t.H 2 / and m.H/ D m.H 1 / C m.H 2 /. The amortized

cost of FIB-HEAP-UNION is therefore equal to its O.1/ actual cost.

Extracting the minimum node

The process of extracting the minimum node is the most complicated of the oper-

ations presented in this section. It is also where the delayed work of consolidating

trees in the root list ﬁnally occurs. The following pseudocode extracts the mini-

mum node. The code assumes for convenience that when a node is removed from

a linked list, pointers remaining in the list are updated, but pointers in the extracted

node are left unchanged. It also calls the auxiliary procedure CONSOLIDATE,

which we shall see shortly.

19.2 Mergeable-heap operations 513

FIB-HEAP-EXTRACT-MIN.H/

1 ´ D H:min

2 if ´ ¤ NIL

3 for each child x of ´

4 add x to the root list of H

5 x:p D NIL

6 remove ´ from the root list of H

7 if ´ == ´:right

8 H:min D NIL

9 else H:min D ´:right

10 CONSOLIDATE.H/

11 H:n D H:n 1

12 return ´

As Figure 19.4 illustrates, FIB-HEAP-EXTRACT-MIN works by ﬁrst making a root

out of each of the minimum node’s children and removing the minimum node from

the root list. It then consolidates the root list by linking roots of equal degree until

at most one root remains of each degree.

We start in line 1 by saving a pointer ´ to the minimum node; the procedure

returns this pointer at the end. If ´ is NIL, then Fibonacci heap H is already empty

and we are done. Otherwise, we delete node ´ from H by making all of ´’s chil-

dren roots of H in lines 3–5 (putting them into the root list) and removing ´ from

the root list in line 6. If ´ is its own right sibling after line 6, then ´ was the

only node on the root list and it had no children, so all that remains is to make

the Fibonacci heap empty in line 8 before returning ´. Otherwise, we set the

pointer H:min into the root list to point to a root other than ´ (in this case, ´’s

right sibling), which is not necessarily going to be the new minimum node when

FIB-HEAP-EXTRACT-MIN is done. Figure 19.4(b) shows the Fibonacci heap of

Figure 19.4(a) after executing line 9.

The next step, in which we reduce the number of trees in the Fibonacci heap, is

consolidating the root list of H, which the call CONSOLIDATE.H/ accomplishes.

Consolidating the root list consists of repeatedly executing the following steps until

every root in the root list has a distinct degree value:

1. Find two roots x and y in the root list with the same degree. Without loss of

generality, let x:key y:key.

2. Link y to x: remove y from the root list, and make y a child of x by calling the

FIB-HEAP-LINK procedure. This procedure increments the attribute x:degree

and clears the mark on y.

514 Chapter 19 Fibonacci Heaps

A

0 1 2 3

A

0 1 2 3

A

0 1 2 3

A

0 1 2 3

A

0 1 2 3

A

0 1 2 3

(c) (d)

(e)

17

30

24 23

26

35

46

7

17

30

24 23

26

35

46

7 21

18 52 38

39 41

(a) 3 (b)

(f)

(g) 21 18 52 38

39 41

(h)

17

30

24 23

26

35

46

7 21 18 52 38

39 41

17

30

24 23

26

35

46

7 21 18 52 38

39 41

17

30

24 23

26

35

46

7 21 18 52 38

39 41

17

30

24 23

26

35

46

7 21 18 52 38

39 41

17

30

24

23 26

35

46

7 21 18 52 38

39 41

17

30

24

23 26

35

46

7 21 18 52 38

39 41

w,x w,x

w,x w,x

w,x w,x

H:min H:min

Figure 19.4 The action of FIB-HEAP-EXTRACT-MIN. (a) A Fibonacci heap H. (b) The situa-

tion after removing the minimum node ´ from the root list and adding its children to the root list.

(c)–(e) The array A and the trees after each of the ﬁrst three iterations of the for loop of lines 4–14 of

the procedure CONSOLIDATE. The procedure processes the root list by starting at the node pointed

to by H:min and following right pointers. Each part shows the values of w and x at the end of an

iteration. (f)–(h) The next iteration of the for loop, with the values of w and x shown at the end of

each iteration of the while loop of lines 7–13. Part (f) shows the situation after the ﬁrst time through

the while loop. The node with key 23 has been linked to the node with key 7, which x now points to.

In part (g), the node with key 17 has been linked to the node with key 7, which x still points to. In

part (h), the node with key 24 has been linked to the node with key 7. Since no node was previously

pointed to by AŒ3, at the end of the for loop iteration, AŒ3 is set to point to the root of the resulting

tree.

19.2 Mergeable-heap operations 515

A

0 1 2 3

A

0 1 2 3

A

0 1 2 3

A

0 1 2 3

17

30

24 23

26

35

46

7 21 18 52 38

39 41

(i)

17

30

24 23

26

35

46

7 21 18 52 38

39 41

(j)

17

30

24 23

26

35

46

7 38

41

(k)

21

18

52

39 17

30

24 23

26

35

46

7 38

41

(l)

21

18

52

39

17

30

24 23

26

35

46

7 38

41

(m)

21

18

52

39

w,x w,x

x w,x

w

H:min

Figure 19.4, continued (i)–(l) The situation after each of the next four iterations of the for loop.

(m) Fibonacci heap H after reconstructing the root list from the array A and determining the new

H:min pointer.

The procedure CONSOLIDATE uses an auxiliary array AŒ0 : : D.H:n/ to keep

track of roots according to their degrees. If AŒi D y, then y is currently a root

with y:degree D i. Of course, in order to allocate the array we have to know how

to calculate the upper bound D.H:n/ on the maximum degree, but we will see how

to do so in Section 19.4.

516 Chapter 19 Fibonacci Heaps

CONSOLIDATE.H/

1 let AŒ0 : : D.H:n/ be a new array

2 for i D 0 to D.H:n/

3 AŒi D NIL

4 for each node w in the root list of H

5 x D w

6 d D x:degree

7 while AŒd ¤ NIL

8 y D AŒd // another node with the same degree as x

9 if x:key > y:key

10 exchange x with y

11 FIB-HEAP-LINK.H; y; x/

12 AŒd D NIL

13 d D d C 1

14 AŒd D x

15 H:min D NIL

16 for i D 0 to D.H:n/

17 if AŒi ¤ NIL

18 if H:min == NIL

19 create a root list for H containing just AŒi

20 H:min D AŒi

21 else insert AŒi into H’s root list

22 if AŒi:key < H:min:key

23 H:min D AŒi

FIB-HEAP-LINK.H; y; x/

1 remove y from the root list of H

2 make y a child of x, incrementing x:degree

3 y:mark D FALSE

In detail, the CONSOLIDATE procedure works as follows. Lines 1–3 allocate

and initialize the array A by making each entry NIL. The for loop of lines 4–14

processes each root w in the root list. As we link roots together, w may be linked

to some other node and no longer be a root. Nevertheless, w is always in a tree

rooted at some node x, which may or may not be w itself. Because we want at

most one root with each degree, we look in the array A to see whether it contains

a root y with the same degree as x. If it does, then we link the roots x and y but

guaranteeing that x remains a root after linking. That is, we link y to x after ﬁrst

exchanging the pointers to the two roots if y’s key is smaller than x’s key. After

we link y to x, the degree of x has increased by 1, and so we continue this process,

linking x and another root whose degree equals x’s new degree, until no other root

19.2 Mergeable-heap operations 517

that we have processed has the same degree as x. We then set the appropriate entry

of A to point to x, so that as we process roots later on, we have recorded that x is

the unique root of its degree that we have already processed. When this for loop

terminates, at most one root of each degree will remain, and the array A will point

to each remaining root.

The while loop of lines 7–13 repeatedly links the root x of the tree containing

node w to another tree whose root has the same degree as x, until no other root has

the same degree. This while loop maintains the following invariant:

At the start of each iteration of the while loop, d D x:degree.

We use this loop invariant as follows:

Initialization: Line 6 ensures that the loop invariant holds the ﬁrst time we enter

the loop.

Maintenance: In each iteration of the while loop, AŒd points to some root y.

Because d D x:degree D y:degree, we want to link x and y. Whichever of

x and y has the smaller key becomes the parent of the other as a result of the

link operation, and so lines 9–10 exchange the pointers to x and y if necessary.

Next, we link y to x by the call FIB-HEAP-LINK.H; y; x/ in line 11. This

call increments x:degree but leaves y:degree as d. Node y is no longer a root,

and so line 12 removes the pointer to it in array A. Because the call of FIB-

HEAP-LINK increments the value of x:degree, line 13 restores the invariant

that d D x:degree.

Termination: We repeat the while loop until AŒd D NIL, in which case there is

no other root with the same degree as x.

After the while loop terminates, we set AŒd to x in line 14 and perform the next

iteration of the for loop.

Figures 19.4(c)–(e) show the array A and the resulting trees after the ﬁrst three

iterations of the for loop of lines 4–14. In the next iteration of the for loop, three

links occur; their results are shown in Figures 19.4(f)–(h). Figures 19.4(i)–(l) show

the result of the next four iterations of the for loop.

All that remains is to clean up. Once the for loop of lines 4–14 completes,

line 15 empties the root list, and lines 16–23 reconstruct it from the array A. The

resulting Fibonacci heap appears in Figure 19.4(m). After consolidating the root

list, FIB-HEAP-EXTRACT-MIN ﬁnishes up by decrementing H:n in line 11 and

returning a pointer to the deleted node ´ in line 12.

We are now ready to show that the amortized cost of extracting the minimum

node of an n-node Fibonacci heap is O.D.n//. Let H denote the Fibonacci heap

just prior to the FIB-HEAP-EXTRACT-MIN operation.

We start by accounting for the actual cost of extracting the minimum node.

An O.D.n// contribution comes from FIB-HEAP-EXTRACT-MIN processing at

518 Chapter 19 Fibonacci Heaps

most D.n/ children of the minimum node and from the work in lines 2–3 and

16–23 of CONSOLIDATE. It remains to analyze the contribution from the for loop

of lines 4–14 in CONSOLIDATE, for which we use an aggregate analysis. The size

of the root list upon calling CONSOLIDATE is at most D.n/ C t.H/ 1, since it

consists of the original t.H/ root-list nodes, minus the extracted root node, plus

the children of the extracted node, which number at most D.n/. Within a given

iteration of the for loop of lines 4–14, the number of iterations of the while loop of

lines 7–13 depends on the root list. But we know that every time through the while

loop, one of the roots is linked to another, and thus the total number of iterations

of the while loop over all iterations of the for loop is at most the number of roots

in the root list. Hence, the total amount of work performed in the for loop is at

most proportional to D.n/ C t.H/. Thus, the total actual work in extracting the

minimum node is O.D.n/ C t.H//.

The potential before extracting the minimum node is t.H/ C 2 m.H/, and the

potential afterward is at most .D.n/ C 1/ C 2 m.H/, since at most D.n/ C 1 roots

remain and no nodes become marked during the operation. The amortized cost is

thus at most

O.D.n/ C t.H// C ..D.n/ C 1/ C 2 m.H// .t.H/ C 2 m.H//

D O.D.n// C O.t.H// t.H/

D O.D.n// ;

since we can scale up the units of potential to dominate the constant hidden

in O.t.H//. Intuitively, the cost of performing each link is paid for by the re-

duction in potential due to the link’s reducing the number of roots by one. We shall

see in Section 19.4 that D.n/ D O.lg n/, so that the amortized cost of extracting

the minimum node is O.lg n/.

Exercises

19.2-1

Show the Fibonacci heap that results from calling FIB-HEAP-EXTRACT-MIN on

the Fibonacci heap shown in Figure 19.4(m).

19.3 Decreasing a key and deleting a node

In this section, we show how to decrease the key of a node in a Fibonacci heap

in O.1/ amortized time and how to delete any node from an n-node Fibonacci

heap in O.D.n// amortized time. In Section 19.4, we will show that the maxi-

19.3 Decreasing a key and deleting a node 519

mum degree D.n/ is O.lg n/, which will imply that FIB-HEAP-EXTRACT-MIN

and FIB-HEAP-DELETE run in O.lg n/ amortized time.

Decreasing a key

In the following pseudocode for the operation FIB-HEAP-DECREASE-KEY, we

assume as before that removing a node from a linked list does not change any of

the structural attributes in the removed node.

FIB-HEAP-DECREASE-KEY.H; x; k/

1 if k > x:key

2 error “new key is greater than current key”

3 x:key D k

4 y D x:p

5 if y ¤ NIL and x:key < y:key

6 CUT.H; x; y/

7 CASCADING-CUT.H; y/

8 if x:key < H:min:key

9 H:min D x

CUT.H; x; y/

1 remove x from the child list of y, decrementing y:degree

2 add x to the root list of H

3 x:p D NIL

4 x:mark D FALSE

CASCADING-CUT.H; y/

1 ´ D y:p

2 if ´ ¤ NIL

3 if y:mark == FALSE

4 y:mark D TRUE

5 else CUT.H; y; ´/

6 CASCADING-CUT.H; ´/

The FIB-HEAP-DECREASE-KEY procedure works as follows. Lines 1–3 ensure

that the new key is no greater than the current key of x and then assign the new key

to x. If x is a root or if x:key y:key, where y is x’s parent, then no structural

changes need occur, since min-heap order has not been violated. Lines 4–5 test for

this condition.

If min-heap order has been violated, many changes may occur. We start by

cutting x in line 6. The CUT procedure “cuts” the link between x and its parent y,

making x a root.

520 Chapter 19 Fibonacci Heaps

We use the mark attributes to obtain the desired time bounds. They record a little

piece of the history of each node. Suppose that the following events have happened

to node x:

1. at some time, x was a root,

2. then x was linked to (made the child of) another node,

3. then two children of x were removed by cuts.

As soon as the second child has been lost, we cut x from its parent, making it a new

root. The attribute x:mark is TRUE if steps 1 and 2 have occurred and one child

of x has been cut. The CUT procedure, therefore, clears x:mark in line 4, since it

performs step 1. (We can now see why line 3 of FIB-HEAP-LINK clears y:mark:

node y is being linked to another node, and so step 2 is being performed. The next

time a child of y is cut, y:mark will be set to TRUE.)

We are not yet done, because x might be the second child cut from its parent y

since the time that y was linked to another node. Therefore, line 7 of FIB-HEAP-

DECREASE-KEY attempts to perform a cascading-cut operation on y. If y is a

root, then the test in line 2 of CASCADING-CUT causes the procedure to just return.

If y is unmarked, the procedure marks it in line 4, since its ﬁrst child has just been

cut, and returns. If y is marked, however, it has just lost its second child; y is cut

in line 5, and CASCADING-CUT calls itself recursively in line 6 on y’s parent ´.

The CASCADING-CUT procedure recurses its way up the tree until it ﬁnds either a

root or an unmarked node.

Once all the cascading cuts have occurred, lines 8–9 of FIB-HEAP-DECREASE-

KEY ﬁnish up by updating H:min if necessary. The only node whose key changed

was the node x whose key decreased. Thus, the new minimum node is either the

original minimum node or node x.

Figure 19.5 shows the execution of two calls of FIB-HEAP-DECREASE-KEY,

starting with the Fibonacci heap shown in Figure 19.5(a). The ﬁrst call, shown

in Figure 19.5(b), involves no cascading cuts. The second call, shown in Fig-

ures 19.5(c)–(e), invokes two cascading cuts.

We shall now show that the amortized cost of FIB-HEAP-DECREASE-KEY is

only O.1/. We start by determining its actual cost. The FIB-HEAP-DECREASE-

KEY procedure takes O.1/ time, plus the time to perform the cascading cuts. Sup-

pose that a given invocation of FIB-HEAP-DECREASE-KEY results in c calls of

CASCADING-CUT (the call made from line 7 of FIB-HEAP-DECREASE-KEY fol-

lowed by c 1 recursive calls of CASCADING-CUT). Each call of CASCADING-

CUT takes O.1/ time exclusive of recursive calls. Thus, the actual cost of FIB-

HEAP-DECREASE-KEY, including all recursive calls, is O.c/.

We next compute the change in potential. Let H denote the Fibonacci heap just

prior to the FIB-HEAP-DECREASE-KEY operation. The call to CUT in line 6 of

19.3 Decreasing a key and deleting a node 521

17

30

24 23

26

35

15 7

21

18

52

38

39 41

(b)

17

30

24 23

26

5 15 7

21

18

52

38

39 41

(c)

17

30

24 23

26 5 15 7

21

18

52

38

39 41

(d)

17

30

24

23

26 5 15 7

21

18

52

38

39 41

(e)

17

30

24 23

26

35

46

7

21

18

52

38

39 41

(a)

H:min

H:min

H:min H:min

H:min

Figure 19.5 Two calls of FIB-HEAP-DECREASE-KEY. (a) The initial Fibonacci heap. (b) The

node with key 46 has its key decreased to 15. The node becomes a root, and its parent (with key 24),

which had previously been unmarked, becomes marked. (c)–(e) The node with key 35 has its key

decreased to 5. In part (c), the node, now with key 5, becomes a root. Its parent, with key 26,

is marked, so a cascading cut occurs. The node with key 26 is cut from its parent and made an

unmarked root in (d). Another cascading cut occurs, since the node with key 24 is marked as well.

This node is cut from its parent and made an unmarked root in part (e). The cascading cuts stop

at this point, since the node with key 7 is a root. (Even if this node were not a root, the cascading

cuts would stop, since it is unmarked.) Part (e) shows the result of the FIB-HEAP-DECREASE-KEY

operation, with H:min pointing to the new minimum node.

FIB-HEAP-DECREASE-KEY creates a new tree rooted at node x and clears x’s

mark bit (which may have already been FALSE). Each call of CASCADING-CUT,

except for the last one, cuts a marked node and clears the mark bit. Afterward, the

Fibonacci heap contains t.H/Cc trees (the original t.H/ trees, c1 trees produced

by cascading cuts, and the tree rooted at x) and at most m.H/cC2 marked nodes

(c 1 were unmarked by cascading cuts and the last call of CASCADING-CUT may

have marked a node). The change in potential is therefore at most

..t.H/ C c/ C 2.m.H/ c C 2// .t.H/ C 2 m.H// D 4 c :

522 Chapter 19 Fibonacci Heaps

Thus, the amortized cost of FIB-HEAP-DECREASE-KEY is at most

O.c/ C 4 c D O.1/ ;

since we can scale up the units of potential to dominate the constant hidden in O.c/.

You can now see why we deﬁned the potential function to include a term that is

twice the number of marked nodes. When a marked node y is cut by a cascading

cut, its mark bit is cleared, which reduces the potential by 2. One unit of potential

pays for the cut and the clearing of the mark bit, and the other unit compensates

for the unit increase in potential due to node y becoming a root.

Deleting a node

The following pseudocode deletes a node from an n-node Fibonacci heap in

O.D.n// amortized time. We assume that there is no key value of 1 currently

in the Fibonacci heap.

FIB-HEAP-DELETE.H; x/

1 FIB-HEAP-DECREASE-KEY.H; x; 1/

2 FIB-HEAP-EXTRACT-MIN.H/

FIB-HEAP-DELETE makes x become the minimum node in the Fibonacci heap by

giving it a uniquely small key of 1. The FIB-HEAP-EXTRACT-MIN procedure

then removes node x from the Fibonacci heap. The amortized time of FIB-HEAP-

DELETE is the sum of the O.1/ amortized time of FIB-HEAP-DECREASE-KEY

and the O.D.n// amortized time of FIB-HEAP-EXTRACT-MIN. Since we shall see

in Section 19.4 that D.n/ D O.lg n/, the amortized time of FIB-HEAP-DELETE

is O.lg n/.

Exercises

19.3-1

Suppose that a root x in a Fibonacci heap is marked. Explain how x came to be

a marked root. Argue that it doesn’t matter to the analysis that x is marked, even

though it is not a root that was ﬁrst linked to another node and then lost one child.

19.3-2

Justify the O.1/ amortized time of FIB-HEAP-DECREASE-KEY as an average cost

per operation by using aggregate analysis.

19.4 Bounding the maximum degree 523

19.4 Bounding the maximum degree

To prove that the amortized time of FIB-HEAP-EXTRACT-MIN and FIB-HEAP-

DELETE is O.lg n/, we must show that the upper bound D.n/ on the degree of

any node of an n-node Fibonacci heap is O.lg n/. In particular, we shall show that

D.n/

log n

˘

, where is the golden ratio, deﬁned in equation (3.24) as

D .1 C

p

5/=2 D 1:61803 : : : :

The key to the analysis is as follows. For each node x within a Fibonacci heap,

deﬁne size.x/ to be the number of nodes, including x itself, in the subtree rooted

at x. (Note that x need not be in the root list—it can be any node at all.) We shall

show that size.x/ is exponential in x:degree. Bear in mind that x:degree is always

maintained as an accurate count of the degree of x.

Lemma 19.1

Let x be any node in a Fibonacci heap, and suppose that x:degree D k. Let

y 1 ; y 2 ; : : : ; y k denote the children of x in the order in which they were linked to x,

from the earliest to the latest. Then, y 1 :degree 0 and y i :degree i 2 for

i D 2; 3; : : : ; k.

Proof Obviously, y 1 :degree 0.

For i 2, we note that when y i was linked to x, all of y 1 ; y 2 ; : : : ; y i1 were

children of x, and so we must have had x:degree i 1. Because node y i is

linked to x (by CONSOLIDATE) only if x:degree D y i :degree, we must have also

had y i :degree i 1 at that time. Since then, node y i has lost at most one

child, since it would have been cut from x (by CASCADING-CUT) if it had lost

two children. We conclude that y i :degree i 2.

We ﬁnally come to the part of the analysis that explains the name “Fibonacci

heaps.” Recall from Section 3.2 that for k D 0; 1; 2; : : :, the kth Fibonacci number

is deﬁned by the recurrence

F k D

0 if k D 0 ;

1 if k D 1 ;

F k1 C F k2 if k 2 :

The following lemma gives another way to express F k .

524 Chapter 19 Fibonacci Heaps

Lemma 19.2

For all integers k 0,

F kC2 D 1 C

k X

iD0

F i :

Proof The proof is by induction on k. When k D 0,

1 C

0 X

iD0

F i D 1 C F 0

D 1 C 0

D F 2 :

We now assume the inductive hypothesis that F kC1 D 1 C

P k1

iD0

F i , and we

have

F kC2 D F k C F kC1

D F k C

1 C

k1 X

iD0

F i

!

D 1 C

k X

iD0

F i :

Lemma 19.3

For all integers k 0, the .k C 2/nd Fibonacci number satisﬁes F kC2 k

.

Proof The proof is by induction on k. The base cases are for k D 0 and k D 1.

When k D 0 we have F 2 D 1 D 0

, and when k D 1 we have F 3 D 2 >

1:619 > 1

. The inductive step is for k 2, and we assume that F iC2 > i

for

i D 0; 1; : : : ; k1. Recall that is the positive root of equation (3.23), x 2 D xC1.

Thus, we have

F kC2 D F kC1 C F k

k1

C

k2

(by the inductive hypothesis)

D

k2

. C 1/

D

k2

2

(by equation (3.23))

D

k

:

The following lemma and its corollary complete the analysis.

19.4 Bounding the maximum degree 525

Lemma 19.4

Let x be any node in a Fibonacci heap, and let k D x:degree. Then size.x/

F kC2 k

, where D .1 C

p

5/=2.

Proof Let s k denote the minimum possible size of any node of degree k in any

Fibonacci heap. Trivially, s 0 D 1 and s 1 D 2. The number s k is at most size.x/

and, because adding children to a node cannot decrease the node’s size, the value

of s k increases monotonically with k. Consider some node ´, in any Fibonacci

heap, such that ´:degree D k and size.´/ D s k . Because s k size.x/, we

compute a lower bound on size.x/ by computing a lower bound on s k . As in

Lemma 19.1, let y 1 ; y 2 ; : : : ; y k denote the children of ´ in the order in which they

were linked to ´. To bound s k , we count one for ´ itself and one for the ﬁrst child y 1

(for which size.y 1 / 1), giving

size.x/ s k

2 C

k X

iD2

s y i : degree

2 C

k X

iD2

s i2 ;

where the last line follows from Lemma 19.1 (so that y i :degree i 2) and the

monotonicity of s k (so that s y i : degree s i2 ).

We now show by induction on k that s k F kC2 for all nonnegative integers k.

The bases, for k D 0 and k D 1, are trivial. For the inductive step, we assume that

k 2 and that s i F iC2 for i D 0; 1; : : : ; k 1. We have

s k 2 C

k X

iD2

s i2

2 C

k X

iD2

F i

D 1 C

k X

iD0

F i

D F kC2 (by Lemma 19.2)

k

(by Lemma 19.3) .

Thus, we have shown that size.x/ s k F kC2 k

.

526 Chapter 19 Fibonacci Heaps

Corollary 19.5

The maximum degree D.n/ of any node in an n-node Fibonacci heap is O.lg n/.

Proof Let x be any node in an n-node Fibonacci heap, and let k D x:degree.

By Lemma 19.4, we have n size.x/ k

. Taking base- logarithms gives

us k log

n. (In fact, because k is an integer, k

log

n

˘

.) The maximum

degree D.n/ of any node is thus O.lg n/.

Exercises

19.4-1

Professor Pinocchio claims that the height of an n-node Fibonacci heap is O.lg n/.

Show that the professor is mistaken by exhibiting, for any positive integer n, a

sequence of Fibonacci-heap operations that creates a Fibonacci heap consisting of

just one tree that is a linear chain of n nodes.

19.4-2

Suppose we generalize the cascading-cut rule to cut a node x from its parent as

soon as it loses its kth child, for some integer constant k. (The rule in Section 19.3

uses k D 2.) For what values of k is D.n/ D O.lg n/?

Problems

19-1 Alternative implementation of deletion

Professor Pisano has proposed the following variant of the FIB-HEAP-DELETE

procedure, claiming that it runs faster when the node being deleted is not the node

pointed to by H:min.

PISANO-DELETE.H; x/

1 if x == H:min

2 FIB-HEAP-EXTRACT-MIN.H/

3 else y D x:p

4 if y ¤ NIL

5 CUT.H; x; y/

6 CASCADING-CUT.H; y/

7 add x’s child list to the root list of H

8 remove x from the root list of H

Problems for Chapter 19 527

a. The professor’s claim that this procedure runs faster is based partly on the as-

sumption that line 7 can be performed in O.1/ actual time. What is wrong with

this assumption?

b. Give a good upper bound on the actual time of PISANO-DELETE when x is

not H:min. Your bound should be in terms of x:degree and the number c of

calls to the CASCADING-CUT procedure.

c. Suppose that we call PISANO-DELETE.H; x/, and let H 0

be the Fibonacci heap

that results. Assuming that node x is not a root, bound the potential of H 0

in

terms of x:degree, c, t.H/, and m.H/.

d. Conclude that the amortized time for PISANO-DELETE is asymptotically no

better than for FIB-HEAP-DELETE, even when x ¤ H:min.

19-2 Binomial trees and binomial heaps

The binomial tree B k is an ordered tree (see Section B.5.2) deﬁned recursively.

As shown in Figure 19.6(a), the binomial tree B 0 consists of a single node. The

binomial tree B k consists of two binomial trees B k1 that are linked together so

that the root of one is the leftmost child of the root of the other. Figure 19.6(b)

shows the binomial trees B 0 through B 4 .

a. Show that for the binomial tree B k ,

1. there are 2 k

nodes,

2. the height of the tree is k,

3. there are exactly

k

i

nodes at depth i for i D 0; 1; : : : ; k, and

4. the root has degree k, which is greater than that of any other node; moreover,

as Figure 19.6(c) shows, if we number the children of the root from left to

right by k 1; k 2; : : : ; 0, then child i is the root of a subtree B i .

A binomial heap H is a set of binomial trees that satisﬁes the following proper-

ties:

1. Each node has a key (like a Fibonacci heap).

2. Each binomial tree in H obeys the min-heap property.

3. For any nonnegative integer k, there is at most one binomial tree in H whose

root has degree k.

b. Suppose that a binomial heap H has a total of n nodes. Discuss the relationship

between the binomial trees that H contains and the binary representation of n.

Conclude that H consists of at most blg nc C 1 binomial trees.

528 Chapter 19 Fibonacci Heaps

B

4

B

k–1

B

k–2

B

k

B

2

B

1

B

0

B

3

B

2

B

1

B

0

B

k

B

k–1

B

k–1

B

0

(a)

depth

0

1

2

3

4

(b)

(c)

Figure 19.6 (a) The recursive deﬁnition of the binomial tree Bk. Triangles represent rooted sub-

trees. (b) The binomial trees B0 through B4. Node depths in B4 are shown. (c) Another way of

looking at the binomial tree Bk.

Suppose that we represent a binomial heap as follows. The left-child, right-

sibling scheme of Section 10.4 represents each binomial tree within a binomial

heap. Each node contains its key; pointers to its parent, to its leftmost child, and

to the sibling immediately to its right (these pointers are NIL when appropriate);

and its degree (as in Fibonacci heaps, how many children it has). The roots form a

singly linked root list, ordered by the degrees of the roots (from low to high), and

we access the binomial heap by a pointer to the ﬁrst node on the root list.

c. Complete the description of how to represent a binomial heap (i.e., name the

attributes, describe when attributes have the value NIL, and deﬁne how the root

list is organized), and show how to implement the same seven operations on

binomial heaps as this chapter implemented on Fibonacci heaps. Each opera-

tion should run in O.lg n/ worst-case time, where n is the number of nodes in

Problems for Chapter 19 529

the binomial heap (or in the case of the UNION operation, in the two binomial

heaps that are being united). The MAKE-HEAP operation should take constant

time.

d. Suppose that we were to implement only the mergeable-heap operations on a

Fibonacci heap (i.e., we do not implement the DECREASE-KEY or DELETE op-

erations). How would the trees in a Fibonacci heap resemble those in a binomial

heap? How would they differ? Show that the maximum degree in an n-node

Fibonacci heap would be at most blg nc.

e. Professor McGee has devised a new data structure based on Fibonacci heaps.

A McGee heap has the same structure as a Fibonacci heap and supports just

the mergeable-heap operations. The implementations of the operations are the

same as for Fibonacci heaps, except that insertion and union consolidate the

root list as their last step. What are the worst-case running times of operations

on McGee heaps?

19-3 More Fibonacci-heap operations

We wish to augment a Fibonacci heap H to support two new operations without

changing the amortized running time of any other Fibonacci-heap operations.

a. The operation FIB-HEAP-CHANGE-KEY.H; x; k/ changes the key of node x

to the value k. Give an efﬁcient implementation of FIB-HEAP-CHANGE-KEY,

and analyze the amortized running time of your implementation for the cases

in which k is greater than, less than, or equal to x:key.

b. Give an efﬁcient implementation of FIB-HEAP-PRUNE.H; r/, which deletes

q D min.r; H:n/ nodes from H. You may choose any q nodes to delete. Ana-

lyze the amortized running time of your implementation. (Hint: You may need

to modify the data structure and potential function.)

19-4 2-3-4 heaps

Chapter 18 introduced the 2-3-4 tree, in which every internal node (other than pos-

sibly the root) has two, three, or four children and all leaves have the same depth. In

this problem, we shall implement 2-3-4 heaps, which support the mergeable-heap

operations.

The 2-3-4 heaps differ from 2-3-4 trees in the following ways. In 2-3-4 heaps,

only leaves store keys, and each leaf x stores exactly one key in the attribute x:key.

The keys in the leaves may appear in any order. Each internal node x contains

a value x:small that is equal to the smallest key stored in any leaf in the subtree

rooted at x. The root r contains an attribute r:height that gives the height of the

530 Chapter 19 Fibonacci Heaps

tree. Finally, 2-3-4 heaps are designed to be kept in main memory, so that disk

reads and writes are not needed.

Implement the following 2-3-4 heap operations. In parts (a)–(e), each operation

should run in O.lg n/ time on a 2-3-4 heap with n elements. The UNION operation

in part (f) should run in O.lg n/ time, where n is the number of elements in the two

input heaps.

a. MINIMUM, which returns a pointer to the leaf with the smallest key.

b. DECREASE-KEY, which decreases the key of a given leaf x to a given value

k x:key.

c. INSERT, which inserts leaf x with key k.

d. DELETE, which deletes a given leaf x.

e. EXTRACT-MIN, which extracts the leaf with the smallest key.

f. UNION, which unites two 2-3-4 heaps, returning a single 2-3-4 heap and de-

stroying the input heaps.

Chapter notes

Fredman and Tarjan [114] introduced Fibonacci heaps. Their paper also describes

the application of Fibonacci heaps to the problems of single-source shortest paths,

all-pairs shortest paths, weighted bipartite matching, and the minimum-spanning-

tree problem.

Subsequently, Driscoll, Gabow, Shrairman, and Tarjan [96] developed “relaxed

heaps” as an alternative to Fibonacci heaps. They devised two varieties of re-

laxed heaps. One gives the same amortized time bounds as Fibonacci heaps. The

other allows DECREASE-KEY to run in O.1/ worst-case (not amortized) time and

EXTRACT-MIN and DELETE to run in O.lg n/ worst-case time. Relaxed heaps

also have some advantages over Fibonacci heaps in parallel algorithms.

See also the chapter notes for Chapter 6 for other data structures that support fast

DECREASE-KEY operations when the sequence of values returned by EXTRACT-

MIN calls are monotonically increasing over time and the data are integers in a

speciﬁc range.

20 van Emde Boas Trees

In previous chapters, we saw data structures that support the operations of a priority

queue—binary heaps in Chapter 6, red-black trees in Chapter 13, 1 and Fibonacci

heaps in Chapter 19. In each of these data structures, at least one important op-

eration took O.lg n/ time, either worst case or amortized. In fact, because each

of these data structures bases its decisions on comparing keys, the .n lg n/ lower

bound for sorting in Section 8.1 tells us that at least one operation will have to

take .lg n/ time. Why? If we could perform both the INSERT and EXTRACT-MIN

operations in o.lg n/ time, then we could sort n keys in o.n lg n/ time by ﬁrst per-

forming n INSERT operations, followed by n EXTRACT-MIN operations.

We saw in Chapter 8, however, that sometimes we can exploit additional infor-

mation about the keys to sort in o.n lg n/ time. In particular, with counting sort

we can sort n keys, each an integer in the range 0 to k, in time ‚.n C k/, which

is ‚.n/ when k D O.n/.

Since we can circumvent the .n lg n/ lower bound for sorting when the keys are

integers in a bounded range, you might wonder whether we can perform each of the

priority-queue operations in o.lg n/ time in a similar scenario. In this chapter, we

shall see that we can: van Emde Boas trees support the priority-queue operations,

and a few others, each in O.lg lg n/ worst-case time. The hitch is that the keys

must be integers in the range 0 to n 1, with no duplicates allowed.

Speciﬁcally, van Emde Boas trees support each of the dynamic set operations

listed on page 230—SEARCH, INSERT, DELETE, MINIMUM, MAXIMUM, SUC-

CESSOR, and PREDECESSOR—in O.lg lg n/ time. In this chapter, we will omit

discussion of satellite data and focus only on storing keys. Because we concentrate

on keys and disallow duplicate keys to be stored, instead of describing the SEARCH

1

Chapter 13 does not explicitly discuss how to implement EXTRACT-MIN and DECREASE-KEY, but

we can easily build these operations for any data structure that supports MINIMUM, DELETE, and

INSERT.

532 Chapter 20 van Emde Boas Trees

operation, we will implement the simpler operation MEMBER.S; x/, which returns

a boolean indicating whether the value x is currently in dynamic set S.

So far, we have used the parameter n for two distinct purposes: the number of

elements in the dynamic set, and the range of the possible values. To avoid any

further confusion, from here on we will use n to denote the number of elements

currently in the set and u as the range of possible values, so that each van Emde

Boas tree operation runs in O.lg lg u/ time. We call the set f0; 1; 2; : : : ; u 1g

the universe of values that can be stored and u the universe size. We assume

throughout this chapter that u is an exact power of 2, i.e., u D 2 k

for some integer

k 1.

Section 20.1 starts us out by examining some simple approaches that will get

us going in the right direction. We enhance these approaches in Section 20.2,

introducing proto van Emde Boas structures, which are recursive but do not achieve

our goal of O.lg lg u/-time operations. Section 20.3 modiﬁes proto van Emde Boas

structures to develop van Emde Boas trees, and it shows how to implement each

operation in O.lg lg u/ time.

20.1 Preliminary approaches

In this section, we shall examine various approaches for storing a dynamic set.

Although none will achieve the O.lg lg u/ time bounds that we desire, we will gain

insights that will help us understand van Emde Boas trees when we see them later

in this chapter.

Direct addressing

Direct addressing, as we saw in Section 11.1, provides the simplest approach to

storing a dynamic set. Since in this chapter we are concerned only with storing

keys, we can simplify the direct-addressing approach to store the dynamic set as a

bit vector, as discussed in Exercise 11.1-2. To store a dynamic set of values from

the universe f0; 1; 2; : : : ; u 1g, we maintain an array AŒ0 : : u 1 of u bits. The

entry AŒx holds a 1 if the value x is in the dynamic set, and it holds a 0 otherwise.

Although we can perform each of the INSERT, DELETE, and MEMBER operations

in O.1/ time with a bit vector, the remaining operations—MINIMUM, MAXIMUM,

SUCCESSOR, and PREDECESSOR—each take ‚.u/ time in the worst case because

20.1 Preliminary approaches 533

0

0

0

1

1

2

1

3

1

4

1

5

0

6

1

7

0

8

0

9

0

10

0

11

0

12

0

13

1

14

1

15

0 1 1 1 0 0 0 1

1 1 0 1

1 1

1

A

Figure 20.1 A binary tree of bits superimposed on top of a bit vector representing the set

f2; 3; 4; 5; 7; 14; 15g when u D 16. Each internal node contains a 1 if and only if some leaf in

its subtree contains a 1. The arrows show the path followed to determine the predecessor of 14 in the

set.

we might have to scan through ‚.u/ elements. 2 For example, if a set contains only

the values 0 and u 1, then to ﬁnd the successor of 0, we would have to scan

entries 1 through u 2 before ﬁnding a 1 in AŒu 1.

Superimposing a binary tree structure

We can short-cut long scans in the bit vector by superimposing a binary tree of bits

on top of it. Figure 20.1 shows an example. The entries of the bit vector form the

leaves of the binary tree, and each internal node contains a 1 if and only if any leaf

in its subtree contains a 1. In other words, the bit stored in an internal node is the

logical-or of its two children.

The operations that took ‚.u/ worst-case time with an unadorned bit vector now

use the tree structure:

To ﬁnd the minimum value in the set, start at the root and head down toward

the leaves, always taking the leftmost node containing a 1.

To ﬁnd the maximum value in the set, start at the root and head down toward

the leaves, always taking the rightmost node containing a 1.

2

We assume throughout this chapter that MINIMUM and MAXIMUM return NIL if the dynamic set

is empty and that SUCCESSOR and PREDECESSOR return NIL if the element they are given has no

successor or predecessor, respectively.

534 Chapter 20 van Emde Boas Trees

To ﬁnd the successor of x, start at the leaf indexed by x, and head up toward the

root until we enter a node from the left and this node has a 1 in its right child ´.

Then head down through node ´, always taking the leftmost node containing

a 1 (i.e., ﬁnd the minimum value in the subtree rooted at the right child ´).

To ﬁnd the predecessor of x, start at the leaf indexed by x, and head up toward

the root until we enter a node from the right and this node has a 1 in its left

child ´. Then head down through node ´, always taking the rightmost node

containing a 1 (i.e., ﬁnd the maximum value in the subtree rooted at the left

child ´).

Figure 20.1 shows the path taken to ﬁnd the predecessor, 7, of the value 14.

We also augment the INSERT and DELETE operations appropriately. When in-

serting a value, we store a 1 in each node on the simple path from the appropriate

leaf up to the root. When deleting a value, we go from the appropriate leaf up to

the root, recomputing the bit in each internal node on the path as the logical-or of

its two children.

Since the height of the tree is lg u and each of the above operations makes at

most one pass up the tree and at most one pass down, each operation takes O.lg u/

time in the worst case.

This approach is only marginally better than just using a red-black tree. We can

still perform the MEMBER operation in O.1/ time, whereas searching a red-black

tree takes O.lg n/ time. Then again, if the number n of elements stored is much

smaller than the size u of the universe, a red-black tree would be faster for all the

other operations.

Superimposing a tree of constant height

What happens if we superimpose a tree with greater degree? Let us assume that

the size of the universe is u D 2 2k

for some integer k, so that

p

u is an integer.

Instead of superimposing a binary tree on top of the bit vector, we superimpose a

tree of degree

p

u. Figure 20.2(a) shows such a tree for the same bit vector as in

Figure 20.1. The height of the resulting tree is always 2.

As before, each internal node stores the logical-or of the bits within its sub-

tree, so that the

p

u internal nodes at depth 1 summarize each group of

p

u val-

ues. As Figure 20.2(b) demonstrates, we can think of these nodes as an array

summaryŒ0 : :

p

u 1, where summaryŒi contains a 1 if and only if the subar-

ray AŒi

p

u : : .i C 1/

p

u 1 contains a 1. We call this

p

u-bit subarray of A

the ith cluster. For a given value of x, the bit AŒx appears in cluster num-

ber bx=

p

uc. Now INSERT becomes an O.1/-time operation: to insert x, set

both AŒx and summaryŒbx=

p

uc to 1. We can use the summary array to perform

20.1 Preliminary approaches 535

0

0

0

1

1

2

1

3

1

4

1

5

0

6

1

7

0

8

0

9

0

10

0

11

0

12

0

13

1

14

1

15

1

1

1 0 1

(a)

0

0

0

1

1

2

1

3

1

4

1

5

0

6

1

7

0

8

0

9

0

10

0

11

0

12

0

13

1

14

1

15

(b)

1

0

1

1

0

2

1

3

A A

summary

p

u bits

p

u bits

Figure 20.2 (a) A tree of degree

p

u superimposed on top of the same bit vector as in Figure 20.1.

Each internal node stores the logical-or of the bits in its subtree. (b) A view of the same structure,

but with the internal nodes at depth 1 treated as an array summaryŒ0 : :

p

u 1, where summaryŒi is

the logical-or of the subarray AŒi

p

u : : .i C 1/

p

u 1.

each of the operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and

DELETE in O.

p

u/ time:

To ﬁnd the minimum (maximum) value, ﬁnd the leftmost (rightmost) entry in

summary that contains a 1, say summaryŒi, and then do a linear search within

the ith cluster for the leftmost (rightmost) 1.

To ﬁnd the successor (predecessor) of x, ﬁrst search to the right (left) within its

cluster. If we ﬁnd a 1, that position gives the result. Otherwise, let i D bx=

p

uc

and search to the right (left) within the summary array from index i. The ﬁrst

position that holds a 1 gives the index of a cluster. Search within that cluster

for the leftmost (rightmost) 1. That position holds the successor (predecessor).

To delete the value x, let i D bx=

p

uc. Set AŒx to 0 and then set summaryŒi

to the logical-or of the bits in the ith cluster.

In each of the above operations, we search through at most two clusters of

p

u bits

plus the summary array, and so each operation takes O.

p

u/ time.

At ﬁrst glance, it seems as though we have made negative progress. Superimpos-

ing a binary tree gave us O.lg u/-time operations, which are asymptotically faster

than O.

p

u/ time. Using a tree of degree

p

u will turn out to be a key idea of van

Emde Boas trees, however. We continue down this path in the next section.

Exercises

20.1-1

Modify the data structures in this section to support duplicate keys.

536 Chapter 20 van Emde Boas Trees

20.1-2

Modify the data structures in this section to support keys that have associated satel-

lite data.

20.1-3

Observe that, using the structures in this section, the way we ﬁnd the successor and

predecessor of a value x does not depend on whether x is in the set at the time.

Show how to ﬁnd the successor of x in a binary search tree when x is not stored in

the tree.

20.1-4

Suppose that instead of superimposing a tree of degree

p

u, we were to superim-

pose a tree of degree u 1=k

, where k > 1 is a constant. What would be the height of

such a tree, and how long would each of the operations take?

20.2 A recursive structure

In this section, we modify the idea of superimposing a tree of degree

p

u on top of

a bit vector. In the previous section, we used a summary structure of size

p

u, with

each entry pointing to another stucture of size

p

u. Now, we make the structure

recursive, shrinking the universe size by the square root at each level of recursion.

Starting with a universe of size u, we make structures holding

p

u D u 1=2

items,

which themselves hold structures of u 1=4

items, which hold structures of u 1=8

items,

and so on, down to a base size of 2.

For simplicity, in this section, we assume that u D 2 2

k

for some integer k, so

that u; u 1=2 ; u 1=4 ; : : : are integers. This restriction would be quite severe in practice,

allowing only values of u in the sequence 2; 4; 16; 256; 65536; : : :. We shall see in

the next section how to relax this assumption and assume only that u D 2 k

for

some integer k. Since the structure we examine in this section is only a precursor

to the true van Emde Boas tree structure, we tolerate this restriction in favor of

aiding our understanding.

Recalling that our goal is to achieve running times of O.lg lg u/ for the oper-

ations, let’s think about how we might obtain such running times. At the end of

Section 4.3, we saw that by changing variables, we could show that the recurrence

T .n/ D 2T

p

n

˘

C lg n (20.1)

has the solution T .n/ D O.lg n lg lg n/. Let’s consider a similar, but simpler,

recurrence:

T .u/ D T .

p

u/ C O.1/ : (20.2)

20.2 A recursive structure 537

If we use the same technique, changing variables, we can show that recur-

rence (20.2) has the solution T .u/ D O.lg lg u/. Let m D lg u, so that u D 2 m

and we have

T .2

m

/ D T .2

m=2

/ C O.1/ :

Now we rename S.m/ D T .2 m /, giving the new recurrence

S.m/ D S.m=2/ C O.1/ :

By case 2 of the master method, this recurrence has the solution S.m/ D O.lg m/.

We change back from S.m/ to T .u/, giving T .u/ D T .2 m / D S.m/ D O.lg m/ D

O.lg lg u/.

Recurrence (20.2) will guide our search for a data structure. We will design a

recursive data structure that shrinks by a factor of

p

u in each level of its recursion.

When an operation traverses this data structure, it will spend a constant amount of

time at each level before recursing to the level below. Recurrence (20.2) will then

characterize the running time of the operation.

Here is another way to think of how the term lg lg u ends up in the solution to

recurrence (20.2). As we look at the universe size in each level of the recursive data

structure, we see the sequence u; u 1=2 ; u 1=4 ; u 1=8 ; : : :. If we consider how many bits

we need to store the universe size at each level, we need lg u at the top level, and

each level needs half the bits of the previous level. In general, if we start with b

bits and halve the number of bits at each level, then after lg b levels, we get down

to just one bit. Since b D lg u, we see that after lg lg u levels, we have a universe

size of 2.

Looking back at the data structure in Figure 20.2, a given value x resides in

cluster number bx=

p

uc. If we view x as a lg u-bit binary integer, that cluster

number, bx=

p

uc, is given by the most signiﬁcant .lg u/=2 bits of x. Within its

cluster, x appears in position x mod

p

u, which is given by the least signiﬁcant

.lg u/=2 bits of x. We will need to index in this way, and so let us deﬁne some

functions that will help us do so:

high.x/ D

x=

p

u

˘

;

low.x/ D x mod

p

u ;

index.x; y/ D x

p

u C y :

The function high.x/ gives the most signiﬁcant .lg u/=2 bits of x, producing the

number of x’s cluster. The function low.x/ gives the least signiﬁcant .lg u/=2 bits

of x and provides x’s position within its cluster. The function index.x; y/ builds an

element number from x and y, treating x as the most signiﬁcant .lg u/=2 bits of the

element number and y as the least signiﬁcant .lg u/=2 bits. We have the identity

x D index.high.x/; low.x//. The value of u used by each of these functions will

538 Chapter 20 van Emde Boas Trees

… 0 1 2 3

p

u 1 proto-EB.u/

u summary cluster

proto-EB.

p

u/ structure

p

u proto-EB.

p

u/ structures

Figure 20.3 The information in a proto-EB.u/ structure when u 4. The structure contains the

universe size u, a pointer summary to a proto-EB.

p

u/ structure, and an array clusterŒ0 : :

p

u 1

of

p

u pointers to proto-EB.

p

u/ structures.

always be the universe size of the data structure in which we call the function,

which changes as we descend into the recursive structure.

20.2.1 Proto van Emde Boas structures

Taking our cue from recurrence (20.2), let us design a recursive data structure to

support the operations. Although this data structure will fail to achieve our goal of

O.lg lg u/ time for some operations, it serves as a basis for the van Emde Boas tree

structure that we will see in Section 20.3.

For the universe f0; 1; 2; : : : ; u 1g, we deﬁne a proto van Emde Boas struc-

ture, or proto-vEB structure, which we denote as proto-EB.u/, recursively as

follows. Each proto-EB.u/ structure contains an attribute u giving its universe

size. In addition, it contains the following:

If u D 2, then it is the base size, and it contains an array AŒ0 : : 1 of two bits.

Otherwise, u D 2 2

k

for some integer k 1, so that u 4. In addition

to the universe size u, the data structure proto-EB.u/ contains the following

attributes, illustrated in Figure 20.3:

a pointer named summary to a proto-EB.

p

u/ structure and

an array clusterŒ0 : :

p

u1 of

p

u pointers, each to a proto-EB.

p

u/ struc-

ture.

The element x, where 0 x < u, is recursively stored in the cluster numbered

high.x/ as element low.x/ within that cluster.

In the two-level structure of the previous section, each node stores a summary

array of size

p

u, in which each entry contains a bit. From the index of each

entry, we can compute the starting index of the subarray of size

p

u that the bit

summarizes. In the proto-vEB structure, we use explicit pointers rather than index

20.2 A recursive structure 539

0 1 2 3

cluster u 16 summary

proto-vEB(16)

0 1

cluster

u

4

summary

proto-vEB(4)

0

1

A

proto-vEB(2)

1

1

0 1

cluster

u

4

summary

proto-vEB(4)

0 1

cluster

u

4

summary

proto-vEB(4)

0 1

cluster

u

4

summary

proto-vEB(4)

0 1

cluster

u

4

summary

proto-vEB(4)

elements 0,1 elements 2,3 clusters 0,1 clusters 2,3 elements 4,5 elements 6,7

elements 8,9 elements 10,11 elements 12,13 elements 14,15

u

2

0

1

A

proto-vEB(2)

1

1

u

2

0

1

A

proto-vEB(2)

0

0

u

2

0

1

A

proto-vEB(2)

0

1

u

2

0

1

A

proto-vEB(2)

0

0

u

2

0

1

A

proto-vEB(2)

0

0

u

2

0

1

A

proto-vEB(2)

0

0

u

2

0

1

A

proto-vEB(2)

0

1

u

2

0

1

A

proto-vEB(2)

1

1

u

2

0

1

A

proto-vEB(2)

1

1

u

2

0

1

A

proto-vEB(2)

1

1

u

2

0

1

A

proto-vEB(2)

0

0

u

2

0

1

A

proto-vEB(2)

0

1

u

2

0

1

A

proto-vEB(2)

0

1

u

2

0

1

A

proto-vEB(2)

1

1

u

2

Figure 20.4 A proto-EB.16/ structure representing the set f2; 3; 4; 5; 7; 14; 15g. It points to four

proto-EB.4/ structures in clusterŒ0 : : 3, and to a summary structure, which is also a proto-EB.4/.

Each proto-EB.4/ structure points to two proto-EB.2/ structures in clusterŒ0 : : 1, and to a

proto-EB.2/ summary. Each proto-EB.2/ structure contains just an array AŒ0 : : 1 of two bits.

The proto-EB.2/ structures above “elements i,j ” store bits i and j of the actual dynamic set, and

the proto-EB.2/ structures above “clusters i,j ” store the summary bits for clusters i and j in the

top-level proto-EB.16/ structure. For clarity, heavy shading indicates the top level of a proto-vEB

structure that stores summary information for its parent structure; such a proto-vEB structure is

otherwise identical to any other proto-vEB structure with the same universe size.

540 Chapter 20 van Emde Boas Trees

calculations. The array summary contains the summary bits stored recursively in a

proto-vEB structure, and the array cluster contains

p

u pointers.

Figure 20.4 shows a fully expanded proto-EB.16/ structure representing the

set f2; 3; 4; 5; 7; 14; 15g. If the value i is in the proto-vEB structure pointed to by

summary, then the ith cluster contains some value in the set being represented.

As in the tree of constant height, clusterŒi represents the values i

p

u through

.i C 1/

p

u 1, which form the ith cluster.

At the base level, the elements of the actual dynamic sets are stored in some

of the proto-EB.2/ structures, and the remaining proto-EB.2/ structures store

summary bits. Beneath each of the non-summary base structures, the ﬁgure in-

dicates which bits it stores. For example, the proto-EB.2/ structure labeled

“elements 6,7” stores bit 6 (0, since element 6 is not in the set) in its AŒ0 and

bit 7 (1, since element 7 is in the set) in its AŒ1.

Like the clusters, each summary is just a dynamic set with universe size

p

u ,

and so we represent each summary as a proto-EB.

p

u/ structure. The four sum-

mary bits for the main proto-EB.16/ structure are in the leftmost proto-EB.4/

structure, and they ultimately appear in two proto-EB.2/ structures. For exam-

ple, the proto-EB.2/ structure labeled “clusters 2,3” has AŒ0 D 0, indicating that

cluster 2 of the proto-EB.16/ structure (containing elements 8; 9; 10; 11) is all 0,

and AŒ1 D 1, telling us that cluster 3 (containing elements 12; 13; 14; 15) has at

least one 1. Each proto-EB.4/ structure points to its own summary, which is itself

stored as a proto-EB.2/ structure. For example, look at the proto-EB.2/ struc-

ture just to the left of the one labeled “elements 0,1.” Because its AŒ0 is 0, it tells

us that the “elements 0,1” structure is all 0, and because its AŒ1 is 1, we know that

the “elements 2,3” structure contains at least one 1.

20.2.2 Operations on a proto van Emde Boas structure

We shall now describe how to perform operations on a proto-vEB structure.

We ﬁrst examine the query operations—MEMBER, MINIMUM, MAXIMUM, and

SUCCESSOR—which do not change the proto-vEB structure. We then discuss

INSERT and DELETE. We leave MAXIMUM and PREDECESSOR, which are sym-

metric to MINIMUM and SUCCESSOR, respectively, as Exercise 20.2-1.

Each of the MEMBER, SUCCESSOR, PREDECESSOR, INSERT, and DELETE op-

erations takes a parameter x, along with a proto-vEB structure V . Each of these

operations assumes that 0 x < V:u.

Determining whether a value is in the set

To perform MEMBER.x/, we need to ﬁnd the bit corresponding to x within the

appropriate proto-EB.2/ structure. We can do so in O.lg lg u/ time, bypassing

20.2 A recursive structure 541

the summary structures altogether. The following procedure takes a proto-EB

structure V and a value x, and it returns a bit indicating whether x is in the dynamic

set held by V .

PROTO-VEB-MEMBER.V; x/

1 if V:u == 2

2 return V:AŒx

3 else return PROTO-VEB-MEMBER.V:clusterŒhigh.x/; low.x//

The PROTO-VEB-MEMBER procedure works as follows. Line 1 tests whether

we are in a base case, where V is a proto-EB.2/ structure. Line 2 handles the

base case, simply returning the appropriate bit of array A. Line 3 deals with the

recursive case, “drilling down” into the appropriate smaller proto-vEB structure.

The value high.x/ says which proto-EB.

p

u/ structure we visit, and low.x/ de-

termines which element within that proto-EB.

p

u/ structure we are querying.

Let’s see what happens when we call PROTO-VEB-MEMBER.V; 6/ on the

proto-EB.16/ structure in Figure 20.4. Since high.6/ D 1 when u D 16, we

recurse into the proto-EB.4/ structure in the upper right, and we ask about ele-

ment low.6/ D 2 of that structure. In this recursive call, u D 4, and so we recurse

again. With u D 4, we have high.2/ D 1 and low.2/ D 0, and so we ask about

element 0 of the proto-EB.2/ structure in the upper right. This recursive call turns

out to be a base case, and so it returns AŒ0 D 0 back up through the chain of re-

cursive calls. Thus, we get the result that PROTO-VEB-MEMBER.V; 6/ returns 0,

indicating that 6 is not in the set.

To determine the running time of PROTO-VEB-MEMBER, let T .u/ denote

its running time on a proto-EB.u/ structure. Each recursive call takes con-

stant time, not including the time taken by the recursive calls that it makes.

When PROTO-VEB-MEMBER makes a recursive call, it makes a call on a

proto-EB.

p

u/ structure. Thus, we can characterize the running time by the recur-

rence T .u/ D T .

p

u/ C O.1/, which we have already seen as recurrence (20.2).

Its solution is T .u/ D O.lg lg u/, and so we conclude that PROTO-VEB-MEMBER

runs in time O.lg lg u/.

Finding the minimum element

Now we examine how to perform the MINIMUM operation. The procedure

PROTO-VEB-MINIMUM.V / returns the minimum element in the proto-vEB struc-

ture V , or NIL if V represents an empty set.

542 Chapter 20 van Emde Boas Trees

PROTO-VEB-MINIMUM.V /

1 if V:u == 2

2 if V:AŒ0 == 1

3 return 0

4 elseif V:AŒ1 == 1

5 return 1

6 else return NIL

7 else min-cluster D PROTO-VEB-MINIMUM.V:summary/

8 if min-cluster == NIL

9 return NIL

10 else offset D PROTO-VEB-MINIMUM.V:clusterŒmin-cluster/

11 return index.min-cluster; offset/

This procedure works as follows. Line 1 tests for the base case, which lines 2–6

handle by brute force. Lines 7–11 handle the recursive case. First, line 7 ﬁnds the

number of the ﬁrst cluster that contains an element of the set. It does so by recur-

sively calling PROTO-VEB-MINIMUM on V:summary, which is a proto-EB.

p

u/

structure. Line 7 assigns this cluster number to the variable min-cluster. If the set

is empty, then the recursive call returned NIL, and line 9 returns NIL. Otherwise,

the minimum element of the set is somewhere in cluster number min-cluster. The

recursive call in line 10 ﬁnds the offset within the cluster of the minimum element

in this cluster. Finally, line 11 constructs the value of the minimum element from

the cluster number and offset, and it returns this value.

Although querying the summary information allows us to quickly ﬁnd the clus-

ter containing the minimum element, because this procedure makes two recursive

calls on proto-EB.

p

u/ structures, it does not run in O.lg lg u/ time in the worst

case. Letting T .u/ denote the worst-case time for PROTO-VEB-MINIMUM on a

proto-EB.u/ structure, we have the recurrence

T .u/ D 2T .

p

u/ C O.1/ : (20.3)

Again, we use a change of variables to solve this recurrence, letting m D lg u,

which gives

T .2

m

/ D 2T .2

m=2

/ C O.1/ :

Renaming S.m/ D T .2 m / gives

S.m/ D 2S.m=2/ C O.1/ ;

which, by case 1 of the master method, has the solution S.m/ D ‚.m/. By chang-

ing back from S.m/ to T .u/, we have that T .u/ D T .2 m / D S.m/ D ‚.m/ D

‚.lg u/. Thus, we see that because of the second recursive call, PROTO-VEB-

MINIMUM runs in ‚.lg u/ time rather than the desired O.lg lg u/ time.

20.2 A recursive structure 543

Finding the successor

The SUCCESSOR operation is even worse. In the worst case, it makes two recursive

calls, along with a call to PROTO-VEB-MINIMUM. The procedure PROTO-VEB-

SUCCESSOR.V; x/ returns the smallest element in the proto-vEB structure V that

is greater than x, or NIL if no element in V is greater than x. It does not require x

to be a member of the set, but it does assume that 0 x < V:u.

PROTO-VEB-SUCCESSOR.V; x/

1 if V:u == 2

2 if x == 0 and V:AŒ1 == 1

3 return 1

4 else return NIL

5 else offset D PROTO-VEB-SUCCESSOR.V:clusterŒhigh.x/; low.x//

6 if offset ¤ NIL

7 return index.high.x/; offset/

8 else succ-cluster D PROTO-VEB-SUCCESSOR.V:summary; high.x//

9 if succ-cluster == NIL

10 return NIL

11 else offset D PROTO-VEB-MINIMUM.V:clusterŒsucc-cluster/

12 return index.succ-cluster; offset/

The PROTO-VEB-SUCCESSOR procedure works as follows. As usual, line 1

tests for the base case, which lines 2–4 handle by brute force: the only way that x

can have a successor within a proto-EB.2/ structure is when x D 0 and AŒ1

is 1. Lines 5–12 handle the recursive case. Line 5 searches for a successor to x

within x’s cluster, assigning the result to offset. Line 6 determines whether x has

a successor within its cluster; if it does, then line 7 computes and returns the value

of this successor. Otherwise, we have to search in other clusters. Line 8 assigns to

succ-cluster the number of the next nonempty cluster, using the summary informa-

tion to ﬁnd it. Line 9 tests whether succ-cluster is NIL, with line 10 returning NIL

if all succeeding clusters are empty. If succ-cluster is non-NIL, line 11 assigns

the ﬁrst element within that cluster to offset, and line 12 computes and returns the

minimum element in that cluster.

In the worst case, PROTO-VEB-SUCCESSOR calls itself recursively twice on

proto-EB.

p

u/ structures, and it makes one call to PROTO-VEB-MINIMUM on

a proto-EB.

p

u/ structure. Thus, the recurrence for the worst-case running

time T .u/ of PROTO-VEB-SUCCESSOR is

T .u/ D 2T .

p

u/ C ‚.lg

p

u/

D 2T .

p

u/ C ‚.lg u/ :

544 Chapter 20 van Emde Boas Trees

We can employ the same technique that we used for recurrence (20.1) to show

that this recurrence has the solution T .u/ D ‚.lg u lg lg u/. Thus, PROTO-VEB-

SUCCESSOR is asymptotically slower than PROTO-VEB-MINIMUM.

Inserting an element

To insert an element, we need to insert it into the appropriate cluster and also set

the summary bit for that cluster to 1. The procedure PROTO-VEB-INSERT.V; x/

inserts the value x into the proto-vEB structure V .

PROTO-VEB-INSERT.V; x/

1 if V:u == 2

2 V:AŒx D 1

3 else PROTO-VEB-INSERT.V:clusterŒhigh.x/; low.x//

4 PROTO-VEB-INSERT.V:summary; high.x//

In the base case, line 2 sets the appropriate bit in the array A to 1. In the recursive

case, the recursive call in line 3 inserts x into the appropriate cluster, and line 4

sets the summary bit for that cluster to 1.

Because PROTO-VEB-INSERT makes two recursive calls in the worst case, re-

currence (20.3) characterizes its running time. Hence, PROTO-VEB-INSERT runs

in ‚.lg u/ time.

Deleting an element

The DELETE operation is more complicated than insertion. Whereas we can always

set a summary bit to 1 when inserting, we cannot always reset the same summary

bit to 0 when deleting. We need to determine whether any bit in the appropriate

cluster is 1. As we have deﬁned proto-vEB structures, we would have to examine

all

p

u bits within a cluster to determine whether any of them are 1. Alternatively,

we could add an attribute n to the proto-vEB structure, counting how many el-

ements it has. We leave implementation of PROTO-VEB-DELETE as Exercises

20.2-2 and 20.2-3.

Clearly, we need to modify the proto-vEB structure to get each operation down

to making at most one recursive call. We will see in the next section how to do so.

Exercises

20.2-1

Write pseudocode for the procedures PROTO-VEB-MAXIMUM and PROTO-VEB-

PREDECESSOR.

20.3 The van Emde Boas tree 545

20.2-2

Write pseudocode for PROTO-VEB-DELETE. It should update the appropriate

summary bit by scanning the related bits within the cluster. What is the worst-

case running time of your procedure?

20.2-3

Add the attribute n to each proto-vEB structure, giving the number of elements

currently in the set it represents, and write pseudocode for PROTO-VEB-DELETE

that uses the attribute n to decide when to reset summary bits to 0. What is the

worst-case running time of your procedure? What other procedures need to change

because of the new attribute? Do these changes affect their running times?

20.2-4

Modify the proto-vEB structure to support duplicate keys.

20.2-5

Modify the proto-vEB structure to support keys that have associated satellite data.

20.2-6

Write pseudocode for a procedure that creates a proto-EB.u/ structure.

20.2-7

Argue that if line 9 of PROTO-VEB-MINIMUM is executed, then the proto-vEB

structure is empty.

20.2-8

Suppose that we designed a proto-vEB structure in which each cluster array had

only u 1=4

elements. What would the running times of each operation be?

20.3 The van Emde Boas tree

The proto-vEB structure of the previous section is close to what we need to achieve

O.lg lg u/ running times. It falls short because we have to recurse too many times

in most of the operations. In this section, we shall design a data structure that

is similar to the proto-vEB structure but stores a little more information, thereby

removing the need for some of the recursion.

In Section 20.2, we observed that the assumption that we made about the uni-

verse size—that u D 2 2

k

for some integer k—is unduly restrictive, conﬁning the

possible values of u an overly sparse set. From this point on, therefore, we will

allow the universe size u to be any exact power of 2, and when

p

u is not an inte-

546 Chapter 20 van Emde Boas Trees

… 0 1 2 3 "

p

u 1

EB.u/ u min max

summary cluster

EB. "

p

u/

"

p

u EB. #

p

u/ trees

Figure 20.5 The information in a EB.u/ tree when u > 2. The structure contains the uni-

verse size u, elements min and max, a pointer summary to a EB. "

p

u/ tree, and an array

clusterŒ0 : : "

p

u 1 of

"

p

u pointers to EB. #

p

u/ trees.

ger—that is, if u is an odd power of 2 (u D 2 2kC1

for some integer k 0)—then

we will divide the lg u bits of a number into the most signiﬁcant d.lg u/=2e bits and

the least signiﬁcant b.lg u/=2c bits. For convenience, we denote 2 d.lg u/=2e

(the “up-

per square root” of u) by "

p

u and 2 b.lg u/=2c

(the “lower square root” of u) by #

p

u,

so that u D "

p

u #

p

u and, when u is an even power of 2 (u D 2 2k

for some

integer k), "

p

u D #

p

u D

p

u. Because we now allow u to be an odd power of 2,

we must redeﬁne our helpful functions from Section 20.2:

high.x/ D

x= #

p

u

˘

;

low.x/ D x mod

#

p

u ;

index.x; y/ D x #

p

u C y :

20.3.1 van Emde Boas trees

The van Emde Boas tree, or vEB tree, modiﬁes the proto-vEB structure. We

denote a vEB tree with a universe size of u as EB.u/ and, unless u equals the

base size of 2, the attribute summary points to a EB. "

p

u/ tree and the array

clusterŒ0 : : "

p

u 1 points to "

p

u EB. #

p

u/ trees. As Figure 20.5 illustrates, a

vEB tree contains two attributes not found in a proto-vEB structure:

min stores the minimum element in the vEB tree, and

max stores the maximum element in the vEB tree.

Furthermore, the element stored in min does not appear in any of the recur-

sive EB. #

p

u/ trees that the cluster array points to. The elements stored in a

EB.u/ tree V , therefore, are V:min plus all the elements recursively stored in

the EB. #

p

u/ trees pointed to by V:clusterŒ0 : : "

p

u 1. Note that when a vEB

tree contains two or more elements, we treat min and max differently: the element

20.3 The van Emde Boas tree 547

stored in min does not appear in any of the clusters, but the element stored in max

does.

Since the base size is 2, a EB.2/ tree does not need the array A that the cor-

responding proto-EB.2/ structure has. Instead, we can determine its elements

from its min and max attributes. In a vEB tree with no elements, regardless of its

universe size u, both min and max are NIL.

Figure 20.6 shows a EB.16/ tree V holding the set f2; 3; 4; 5; 7; 14; 15g. Be-

cause the smallest element is 2, V:min equals 2, and even though high.2/ D 0, the

element 2 does not appear in the EB.4/ tree pointed to by V:clusterŒ0: notice

that V:clusterŒ0:min equals 3, and so 2 is not in this vEB tree. Similarly, since

V:clusterŒ0:min equals 3, and 2 and 3 are the only elements in V:clusterŒ0, the

EB.2/ clusters within V:clusterŒ0 are empty.

The min and max attributes will turn out to be key to reducing the number of

recursive calls within the operations on vEB trees. These attributes will help us in

four ways:

1. The MINIMUM and MAXIMUM operations do not even need to recurse, for they

can just return the values of min or max.

2. The SUCCESSOR operation can avoid making a recursive call to determine

whether the successor of a value x lies within high.x/. That is because x’s

successor lies within its cluster if and only if x is strictly less than the max

attribute of its cluster. A symmetric argument holds for PREDECESSOR and

min.

3. We can tell whether a vEB tree has no elements, exactly one element, or at least

two elements in constant time from its min and max values. This ability will

help in the INSERT and DELETE operations. If min and max are both NIL, then

the vEB tree has no elements. If min and max are non-NIL but are equal to each

other, then the vEB tree has exactly one element. Otherwise, both min and max

are non-NIL but are unequal, and the vEB tree has two or more elements.

4. If we know that a vEB tree is empty, we can insert an element into it by updating

only its min and max attributes. Hence, we can insert into an empty vEB tree in

constant time. Similarly, if we know that a vEB tree has only one element, we

can delete that element in constant time by updating only min and max. These

properties will allow us to cut short the chain of recursive calls.

Even if the universe size u is an odd power of 2, the difference in the sizes

of the summary vEB tree and the clusters will not turn out to affect the asymptotic

running times of the vEB-tree operations. The recursive procedures that implement

the vEB-tree operations will all have running times characterized by the recurrence

T .u/ T . "

p

u/ C O.1/ : (20.4)

548 Chapter 20 van Emde Boas Trees

0 1 2 3

cluster

u 16

summary

vEB(16) min 2 max 15

0 1

cluster

u 4

summary

vEB(4) min 0 max 3

u 2

min 0

max 1

vEB(2)

u 2

min 1

max 1

vEB(2)

u 2

min 1

max 1

vEB(2)

0 1

cluster summary

u 2

min

max

vEB(2)

u 2

min

max

vEB(2)

u 2

min

max

vEB(2)

0 1

cluster summary

u 2

min 0

max 1

vEB(2)

u 2

min 1

max 1

vEB(2)

u 2

min 1

max 1

vEB(2)

0 1

cluster summary

u 2

min

max

vEB(2)

u 2

min

max

vEB(2)

u 2

min

max

vEB(2)

0 1

cluster summary

u 2

min 1

max 1

vEB(2)

u 2

min

max

vEB(2)

u 2

min 1

max 1

vEB(2)

u 4 vEB(4) min 3 max 3 u 4 vEB(4) min 0 max 3

u 4 vEB(4) min max u 4 vEB(4) min 2 max 3

Figure 20.6 A EB.16/ tree corresponding to the proto-vEB tree in Figure 20.4. It stores the set

f2; 3; 4; 5; 7; 14; 15g. Slashes indicate NIL values. The value stored in the min attribute of a vEB tree

does not appear in any of its clusters. Heavy shading serves the same purpose here as in Figure 20.4.

20.3 The van Emde Boas tree 549

This recurrence looks similar to recurrence (20.2), and we will solve it in a similar

fashion. Letting m D lg u, we rewrite it as

T .2

m

/ T .2

dm=2e

/ C O.1/ :

Noting that dm=2e 2m=3 for all m 2, we have

T .2

m

/ T .2

2m=3

/ C O.1/ :

Letting S.m/ D T .2 m /, we rewrite this last recurrence as

S.m/ S.2m=3/ C O.1/ ;

which, by case 2 of the master method, has the solution S.m/ D O.lg m/. (In

terms of the asymptotic solution, the fraction 2=3 does not make any difference

compared with the fraction 1=2, because when we apply the master method, we

ﬁnd that log 3=2 1 D log 2 1 D 0:) Thus, we have T .u/ D T .2 m / D S.m/ D

O.lg m/ D O.lg lg u/.

Before using a van Emde Boas tree, we must know the universe size u, so that

we can create a van Emde Boas tree of the appropriate size that initially represents

an empty set. As Problem 20-1 asks you to show, the total space requirement of

a van Emde Boas tree is O.u/, and it is straightforward to create an empty tree

in O.u/ time. In contrast, we can create an empty red-black tree in constant time.

Therefore, we might not want to use a van Emde Boas tree when we perform only

a small number of operations, since the time to create the data structure would

exceed the time saved in the individual operations. This drawback is usually not

signiﬁcant, since we typically use a simple data structure, such as an array or linked

list, to represent a set with only a few elements.

20.3.2 Operations on a van Emde Boas tree

We are now ready to see how to perform operations on a van Emde Boas tree. As

we did for the proto van Emde Boas structure, we will consider the querying oper-

ations ﬁrst, and then INSERT and DELETE. Due to the slight asymmetry between

the minimum and maximum elements in a vEB tree—when a vEB tree contains

at least two elements, the minumum element does not appear within a cluster but

the maximum element does—we will provide pseudocode for all ﬁve querying op-

erations. As in the operations on proto van Emde Boas structures, the operations

here that take parameters V and x, where V is a van Emde Boas tree and x is an

element, assume that 0 x < V:u.

Finding the minimum and maximum elements

Because we store the minimum and maximum in the attributes min and max, two

of the operations are one-liners, taking constant time:

550 Chapter 20 van Emde Boas Trees

VEB-TREE-MINIMUM.V /

1 return V:min

VEB-TREE-MAXIMUM.V /

1 return V:max

Determining whether a value is in the set

The procedure VEB-TREE-MEMBER.V; x/ has a recursive case like that of

PROTO-VEB-MEMBER, but the base case is a little different. We also check di-

rectly whether x equals the minimum or maximum element. Since a vEB tree

doesn’t store bits as a proto-vEB structure does, we design VEB-TREE-MEMBER

to return TRUE or FALSE rather than 1 or 0.

VEB-TREE-MEMBER.V; x/

1 if x == V:min or x == V:max

2 return TRUE

3 elseif V:u == 2

4 return FALSE

5 else return VEB-TREE-MEMBER.V:clusterŒhigh.x/; low.x//

Line 1 checks to see whether x equals either the minimum or maximum element.

If it does, line 2 returns TRUE. Otherwise, line 3 tests for the base case. Since

a EB.2/ tree has no elements other than those in min and max, if it is the base

case, line 4 returns FALSE. The other possibility—it is not a base case and x equals

neither min nor max—is handled by the recursive call in line 5.

Recurrence (20.4) characterizes the running time of the VEB-TREE-MEMBER

procedure, and so this procedure takes O.lg lg u/ time.

Finding the successor and predecessor

Next we see how to implement the SUCCESSOR operation. Recall that the pro-

cedure PROTO-VEB-SUCCESSOR.V; x/ could make two recursive calls: one to

determine whether x’s successor resides in the same cluster as x and, if it does

not, one to ﬁnd the cluster containing x’s successor. Because we can access the

maximum value in a vEB tree quickly, we can avoid making two recursive calls,

and instead make one recursive call on either a cluster or on the summary, but not

on both.

20.3 The van Emde Boas tree 551

VEB-TREE-SUCCESSOR.V; x/

1 if V:u == 2

2 if x == 0 and V:max == 1

3 return 1

4 else return NIL

5 elseif V:min ¤ NIL and x < V:min

6 return V:min

7 else max-low D VEB-TREE-MAXIMUM.V:clusterŒhigh.x//

8 if max-low ¤ NIL and low.x/ < max-low

9 offset D VEB-TREE-SUCCESSOR.V:clusterŒhigh.x/; low.x//

10 return index.high.x/; offset/

11 else succ-cluster D VEB-TREE-SUCCESSOR.V:summary; high.x//

12 if succ-cluster == NIL

13 return NIL

14 else offset D VEB-TREE-MINIMUM.V:clusterŒsucc-cluster/

15 return index.succ-cluster; offset/

This procedure has six return statements and several cases. We start with the

base case in lines 2–4, which returns 1 in line 3 if we are trying to ﬁnd the successor

of 0 and 1 is in the 2-element set; otherwise, the base case returns NIL in line 4.

If we are not in the base case, we next check in line 5 whether x is strictly less

than the minimum element. If so, then we simply return the minimum element in

line 6.

If we get to line 7, then we know that we are not in a base case and that x is

greater than or equal to the minimum value in the vEB tree V . Line 7 assigns to

max-low the maximum element in x’s cluster. If x’s cluster contains some element

that is greater than x, then we know that x’s successor lies somewhere within x’s

cluster. Line 8 tests for this condition. If x’s successor is within x’s cluster, then

line 9 determines where in the cluster it is, and line 10 returns the successor in the

same way as line 7 of PROTO-VEB-SUCCESSOR.

We get to line 11 if x is greater than or equal to the greatest element in its

cluster. In this case, lines 11–15 ﬁnd x’s successor in the same way as lines 8–12

of PROTO-VEB-SUCCESSOR.

It is easy to see how recurrence (20.4) characterizes the running time of VEB-

TREE-SUCCESSOR. Depending on the result of the test in line 7, the procedure

calls itself recursively in either line 9 (on a vEB tree with universe size #

p

u) or

line 11 (on a vEB tree with universe size "

p

u). In either case, the one recursive

call is on a vEB tree with universe size at most "

p

u. The remainder of the proce-

dure, including the calls to VEB-TREE-MINIMUM and VEB-TREE-MAXIMUM,

takes O.1/ time. Hence, VEB-TREE-SUCCESSOR runs in O.lg lg u/ worst-case

time.

552 Chapter 20 van Emde Boas Trees

The VEB-TREE-PREDECESSOR procedure is symmetric to the VEB-TREE-

SUCCESSOR procedure, but with one additional case:

VEB-TREE-PREDECESSOR.V; x/

1 if V:u == 2

2 if x == 1 and V:min == 0

3 return 0

4 else return NIL

5 elseif V:max ¤ NIL and x > V:max

6 return V:max

7 else min-low D VEB-TREE-MINIMUM.V:clusterŒhigh.x//

8 if min-low ¤ NIL and low.x/ > min-low

9 offset D VEB-TREE-PREDECESSOR.V:clusterŒhigh.x/; low.x//

10 return index.high.x/; offset/

11 else pred-cluster D VEB-TREE-PREDECESSOR.V:summary; high.x//

12 if pred-cluster == NIL

13 if V:min ¤ NIL and x > V:min

14 return V:min

15 else return NIL

16 else offset D VEB-TREE-MAXIMUM.V:clusterŒpred-cluster/

17 return index.pred-cluster; offset/

Lines 13–14 form the additional case. This case occurs when x’s predecessor,

if it exists, does not reside in x’s cluster. In VEB-TREE-SUCCESSOR, we were

assured that if x’s successor resides outside of x’s cluster, then it must reside in

a higher-numbered cluster. But if x’s predecessor is the minimum value in vEB

tree V , then the successor resides in no cluster at all. Line 13 checks for this

condition, and line 14 returns the minimum value as appropriate.

This extra case does not affect the asymptotic running time of VEB-TREE-

PREDECESSOR when compared with VEB-TREE-SUCCESSOR, and so VEB-

TREE-PREDECESSOR runs in O.lg lg u/ worst-case time.

Inserting an element

Now we examine how to insert an element into a vEB tree. Recall that PROTO-

VEB-INSERT made two recursive calls: one to insert the element and one to insert

the element’s cluster number into the summary. The VEB-TREE-INSERT proce-

dure will make only one recursive call. How can we get away with just one? When

we insert an element, either the cluster that it goes into already has another element

or it does not. If the cluster already has another element, then the cluster number

is already in the summary, and so we do not need to make that recursive call. If

20.3 The van Emde Boas tree 553

the cluster does not already have another element, then the element being inserted

becomes the only element in the cluster, and we do not need to recurse to insert an

element into an empty vEB tree:

VEB-EMPTY-TREE-INSERT.V; x/

1 V:min D x

2 V:max D x

With this procedure in hand, here is the pseudocode for VEB-TREE-INSERT.V; x/,

which assumes that x is not already an element in the set represented by vEB

tree V :

VEB-TREE-INSERT.V; x/

1 if V:min == NIL

2 VEB-EMPTY-TREE-INSERT.V; x/

3 else if x < V:min

4 exchange x with V:min

5 if V:u > 2

6 if VEB-TREE-MINIMUM.V:clusterŒhigh.x// == NIL

7 VEB-TREE-INSERT.V:summary; high.x//

8 VEB-EMPTY-TREE-INSERT.V:clusterŒhigh.x/; low.x//

9 else VEB-TREE-INSERT.V:clusterŒhigh.x/; low.x//

10 if x > V:max

11 V:max D x

This procedure works as follows. Line 1 tests whether V is an empty vEB tree

and, if it is, then line 2 handles this easy case. Lines 3–11 assume that V is not

empty, and therefore some element will be inserted into one of V ’s clusters. But

that element might not necessarily be the element x passed to VEB-TREE-INSERT.

If x < min, as tested in line 3, then x needs to become the new min. We don’t

want to lose the original min, however, and so we need to insert it into one of V ’s

clusters. In this case, line 4 exchanges x with min, so that we insert the original

min into one of V ’s clusters.

We execute lines 6–9 only if V is not a base-case vEB tree. Line 6 determines

whether the cluster that x will go into is currently empty. If so, then line 7 in-

serts x’s cluster number into the summary and line 8 handles the easy case of

inserting x into an empty cluster. If x’s cluster is not currently empty, then line 9

inserts x into its cluster. In this case, we do not need to update the summary,

since x’s cluster number is already a member of the summary.

Finally, lines 10–11 take care of updating max if x > max. Note that if V is a

base-case vEB tree that is not empty, then lines 3–4 and 10–11 update min and max

properly.

554 Chapter 20 van Emde Boas Trees

Once again, we can easily see how recurrence (20.4) characterizes the running

time. Depending on the result of the test in line 6, either the recursive call in line 7

(run on a vEB tree with universe size "

p

u) or the recursive call in line 9 (run on

a vEB with universe size #

p

u) executes. In either case, the one recursive call is

on a vEB tree with universe size at most "

p

u. Because the remainder of VEB-

TREE-INSERT takes O.1/ time, recurrence (20.4) applies, and so the running time

is O.lg lg u/.

Deleting an element

Finally, we look at how to delete an element from a vEB tree. The procedure

VEB-TREE-DELETE.V; x/ assumes that x is currently an element in the set repre-

sented by the vEB tree V .

VEB-TREE-DELETE.V; x/

1 if V:min == V:max

2 V:min D NIL

3 V:max D NIL

4 elseif V:u == 2

5 if x == 0

6 V:min D 1

7 else V:min D 0

8 V:max D V:min

9 else if x == V:min

10 ﬁrst-cluster D VEB-TREE-MINIMUM.V:summary/

11 x D index.ﬁrst-cluster;

VEB-TREE-MINIMUM.V:clusterŒﬁrst-cluster//

12 V:min D x

13 VEB-TREE-DELETE.V:clusterŒhigh.x/; low.x//

14 if VEB-TREE-MINIMUM.V:clusterŒhigh.x// == NIL

15 VEB-TREE-DELETE.V:summary; high.x//

16 if x == V:max

17 summary-max D VEB-TREE-MAXIMUM.V:summary/

18 if summary-max == NIL

19 V:max D V:min

20 else V:max D index.summary-max;

VEB-TREE-MAXIMUM.V:clusterŒsummary-max//

21 elseif x == V:max

22 V:max D index.high.x/;

VEB-TREE-MAXIMUM.V:clusterŒhigh.x///

20.3 The van Emde Boas tree 555

The VEB-TREE-DELETE procedure works as follows. If the vEB tree V con-

tains only one element, then it’s just as easy to delete it as it was to insert an element

into an empty vEB tree: just set min and max to NIL. Lines 1–3 handle this case.

Otherwise, V has at least two elements. Line 4 tests whether V is a base-case vEB

tree and, if so, lines 5–8 set min and max to the one remaining element.

Lines 9–22 assume that V has two or more elements and that u 4. In this

case, we will have to delete an element from a cluster. The element we delete from

a cluster might not be x, however, because if x equals min, then once we have

deleted x, some other element within one of V ’s clusters becomes the new min,

and we have to delete that other element from its cluster. If the test in line 9 reveals

that we are in this case, then line 10 sets ﬁrst-cluster to the number of the cluster

that contains the lowest element other than min, and line 11 sets x to the value of

the lowest element in that cluster. This element becomes the new min in line 12

and, because we set x to its value, it is the element that will be deleted from its

cluster.

When we reach line 13, we know that we need to delete element x from its

cluster, whether x was the value originally passed to VEB-TREE-DELETE or x

is the element becoming the new minimum. Line 13 deletes x from its cluster.

That cluster might now become empty, which line 14 tests, and if it does, then

we need to remove x’s cluster number from the summary, which line 15 handles.

After updating the summary, we might need to update max. Line 16 checks to see

whether we are deleting the maximum element in V and, if we are, then line 17 sets

summary-max to the number of the highest-numbered nonempty cluster. (The call

VEB-TREE-MAXIMUM.V:summary/ works because we have already recursively

called VEB-TREE-DELETE on V:summary, and therefore V:summary:max has al-

ready been updated as necessary.) If all of V ’s clusters are empty, then the only

remaining element in V is min; line 18 checks for this case, and line 19 updates

max appropriately. Otherwise, line 20 sets max to the maximum element in the

highest-numbered cluster. (If this cluster is where the element has been deleted,

we again rely on the recursive call in line 13 having already corrected that cluster’s

max attribute.)

Finally, we have to handle the case in which x’s cluster did not become empty

due to x being deleted. Although we do not have to update the summary in this

case, we might have to update max. Line 21 tests for this case, and if we have to

update max, line 22 does so (again relying on the recursive call to have corrected

max in the cluster).

Now we show that VEB-TREE-DELETE runs in O.lg lg u/ time in the worst

case. At ﬁrst glance, you might think that recurrence (20.4) does not always apply,

because a single call of VEB-TREE-DELETE can make two recursive calls: one

on line 13 and one on line 15. Although the procedure can make both recursive

calls, let’s think about what happens when it does. In order for the recursive call on

556 Chapter 20 van Emde Boas Trees

line 15 to occur, the test on line 14 must show that x’s cluster is empty. The only

way that x’s cluster can be empty is if x was the only element in its cluster when

we made the recursive call on line 13. But if x was the only element in its cluster,

then that recursive call took O.1/ time, because it executed only lines 1–3. Thus,

we have two mutually exclusive possibilities:

The recursive call on line 13 took constant time.

The recursive call on line 15 did not occur.

In either case, recurrence (20.4) characterizes the running time of VEB-TREE-

DELETE, and hence its worst-case running time is O.lg lg u/.

Exercises

20.3-1

Modify vEB trees to support duplicate keys.

20.3-2

Modify vEB trees to support keys that have associated satellite data.

20.3-3

Write pseudocode for a procedure that creates an empty van Emde Boas tree.

20.3-4

What happens if you call VEB-TREE-INSERT with an element that is already in

the vEB tree? What happens if you call VEB-TREE-DELETE with an element that

is not in the vEB tree? Explain why the procedures exhibit the behavior that they

do. Show how to modify vEB trees and their operations so that we can check in

constant time whether an element is present.

20.3-5

Suppose that instead of "

p

u clusters, each with universe size #

p

u, we constructed

vEB trees to have u 1=k

clusters, each with universe size u 11=k

, where k > 1 is a

constant. If we were to modify the operations appropriately, what would be their

running times? For the purpose of analysis, assume that u 1=k

and u 11=k

are always

integers.

20.3-6

Creating a vEB tree with universe size u requires O.u/ time. Suppose we wish to

explicitly account for that time. What is the smallest number of operations n for

which the amortized time of each operation in a vEB tree is O.lg lg u/?

Problems for Chapter 20 557

Problems

20-1 Space requirements for van Emde Boas trees

This problem explores the space requirements for van Emde Boas trees and sug-

gests a way to modify the data structure to make its space requirement depend on

the number n of elements actually stored in the tree, rather than on the universe

size u. For simplicity, assume that

p

u is always an integer.

a. Explain why the following recurrence characterizes the space requirement P.u/

of a van Emde Boas tree with universe size u:

P.u/ D .

p

u C 1/P.

p

u/ C ‚.

p

u/ : (20.5)

b. Prove that recurrence (20.5) has the solution P.u/ D O.u/.

In order to reduce the space requirements, let us deﬁne a reduced-space van Emde

Boas tree, or RS-vEB tree, as a vEB tree V but with the following changes:

The attribute V:cluster, rather than being stored as a simple array of pointers to

vEB trees with universe size

p

u, is a hash table (see Chapter 11) stored as a dy-

namic table (see Section 17.4). Corresponding to the array version of V:cluster,

the hash table stores pointers to RS-vEB trees with universe size

p

u. To ﬁnd

the ith cluster, we look up the key i in the hash table, so that we can ﬁnd the

ith cluster by a single search in the hash table.

The hash table stores only pointers to nonempty clusters. A search in the hash

table for an empty cluster returns NIL, indicating that the cluster is empty.

The attribute V:summary is NIL if all clusters are empty. Otherwise, V:summary

points to an RS-vEB tree with universe size

p

u.

Because the hash table is implemented with a dynamic table, the space it requires

is proportional to the number of nonempty clusters.

When we need to insert an element into an empty RS-vEB tree, we create the RS-

vEB tree by calling the following procedure, where the parameter u is the universe

size of the RS-vEB tree:

CREATE-NEW-RS-VEB-TREE.u/

1 allocate a new vEB tree V

2 V:u D u

3 V:min D NIL

4 V:max D NIL

5 V:summary D NIL

6 create V:cluster as an empty dynamic hash table

7 return V

558 Chapter 20 van Emde Boas Trees

c. Modify the VEB-TREE-INSERT procedure to produce pseudocode for the pro-

cedure RS-VEB-TREE-INSERT.V; x/, which inserts x into the RS-vEB tree V ,

calling CREATE-NEW-RS-VEB-TREE as appropriate.

d. Modify the VEB-TREE-SUCCESSOR procedure to produce pseudocode for

the procedure RS-VEB-TREE-SUCCESSOR.V; x/, which returns the successor

of x in RS-vEB tree V , or NIL if x has no successor in V .

e. Prove that, under the assumption of simple uniform hashing, your RS-VEB-

TREE-INSERT and RS-VEB-TREE-SUCCESSOR procedures run in O.lg lg u/

expected time.

f. Assuming that elements are never deleted from a vEB tree, prove that the space

requirement for the RS-vEB tree structure is O.n/, where n is the number of

elements actually stored in the RS-vEB tree.

g. RS-vEB trees have another advantage over vEB trees: they require less time to

create. How long does it take to create an empty RS-vEB tree?

20-2 y-fast tries

This problem investigates D. Willard’s “y-fast tries” which, like van Emde Boas

trees, perform each of the operations MEMBER, MINIMUM, MAXIMUM, PRE-

DECESSOR, and SUCCESSOR on elements drawn from a universe with size u in

O.lg lg u/ worst-case time. The INSERT and DELETE operations take O.lg lg u/

amortized time. Like reduced-space van Emde Boas trees (see Problem 20-1), y-

fast tries use only O.n/ space to store n elements. The design of y-fast tries relies

on perfect hashing (see Section 11.5).

As a preliminary structure, suppose that we create a perfect hash table containing

not only every element in the dynamic set, but every preﬁx of the binary represen-

tation of every element in the set. For example, if u D 16, so that lg u D 4, and

x D 13 is in the set, then because the binary representation of 13 is 1101, the

perfect hash table would contain the strings 1, 11, 110, and 1101. In addition to

the hash table, we create a doubly linked list of the elements currently in the set, in

increasing order.

a. How much space does this structure require?

b. Show how to perform the MINIMUM and MAXIMUM operations in O.1/ time;

the MEMBER, PREDECESSOR, and SUCCESSOR operations in O.lg lg u/ time;

and the INSERT and DELETE operations in O.lg u/ time.

To reduce the space requirement to O.n/, we make the following changes to the

data structure:

Notes for Chapter 20 559

We cluster the n elements into n= lg u groups of size lg u. (Assume for now

that lg u divides n.) The ﬁrst group consists of the lg u smallest elements in the

set, the second group consists of the next lg u smallest elements, and so on.

We designate a “representative” value for each group. The representative of

the ith group is at least as large as the largest element in the ith group, and it is

smaller than every element of the .i C1/st group. (The representative of the last

group can be the maximum possible element u 1.) Note that a representative

might be a value not currently in the set.

We store the lg u elements of each group in a balanced binary search tree, such

as a red-black tree. Each representative points to the balanced binary search

tree for its group, and each balanced binary search tree points to its group’s

representative.

The perfect hash table stores only the representatives, which are also stored in

a doubly linked list in increasing order.

We call this structure a y-fast trie.

c. Show that a y-fast trie requires only O.n/ space to store n elements.

d. Show how to perform the MINIMUM and MAXIMUM operations in O.lg lg u/

time with a y-fast trie.

e. Show how to perform the MEMBER operation in O.lg lg u/ time.

f. Show how to perform the PREDECESSOR and SUCCESSOR operations in

O.lg lg u/ time.

g. Explain why the INSERT and DELETE operations take .lg lg u/ time.

h. Show how to relax the requirement that each group in a y-fast trie has exactly

lg u elements to allow INSERT and DELETE to run in O.lg lg u/ amortized time

without affecting the asymptotic running times of the other operations.

Chapter notes

The data structure in this chapter is named after P. van Emde Boas, who described

an early form of the idea in 1975 [339]. Later papers by van Emde Boas [340]

and van Emde Boas, Kaas, and Zijlstra [341] reﬁned the idea and the exposition.

Mehlhorn and N¨aher [252] subsequently extended the ideas to apply to universe

560 Chapter 20 van Emde Boas Trees

sizes that are prime. Mehlhorn’s book [249] contains a slightly different treatment

of van Emde Boas trees than the one in this chapter.

Using the ideas behind van Emde Boas trees, Dementiev et al. [83] developed

a nonrecursive, three-level search tree that ran faster than van Emde Boas trees in

their own experiments.

Wang and Lin [347] designed a hardware-pipelined version of van Emde Boas

trees, which achieves constant amortized time per operation and uses O.lg lg u/

stages in the pipeline.

A lower bound by Pˇatras¸cu and Thorup [273, 274] for ﬁnding the predecessor

shows that van Emde Boas trees are optimal for this operation, even if randomiza-

tion is allowed.

21 Data Structures for Disjoint Sets

Some applications involve grouping n distinct elements into a collection of disjoint

sets. These applications often need to perform two operations in particular: ﬁnding

the unique set that contains a given element and uniting two sets. This chapter

explores methods for maintaining a data structure that supports these operations.

Section 21.1 describes the operations supported by a disjoint-set data structure

and presents a simple application. In Section 21.2, we look at a simple linked-list

implementation for disjoint sets. Section 21.3 presents a more efﬁcient represen-

tation using rooted trees. The running time using the tree representation is theo-

retically superlinear, but for all practical purposes it is linear. Section 21.4 deﬁnes

and discusses a very quickly growing function and its very slowly growing inverse,

which appears in the running time of operations on the tree-based implementation,

and then, by a complex amortized analysis, proves an upper bound on the running

time that is just barely superlinear.

21.1 Disjoint-set operations

A disjoint-set data structure maintains a collection S D fS 1 ; S 2 ; : : : ; S k g of dis-

joint dynamic sets. We identify each set by a representative, which is some mem-

ber of the set. In some applications, it doesn’t matter which member is used as the

representative; we care only that if we ask for the representative of a dynamic set

twice without modifying the set between the requests, we get the same answer both

times. Other applications may require a prespeciﬁed rule for choosing the repre-

sentative, such as choosing the smallest member in the set (assuming, of course,

that the elements can be ordered).

As in the other dynamic-set implementations we have studied, we represent each

element of a set by an object. Letting x denote an object, we wish to support the

following operations:

562 Chapter 21 Data Structures for Disjoint Sets

MAKE-SET.x/ creates a new set whose only member (and thus representative)

is x. Since the sets are disjoint, we require that x not already be in some other

set.

UNION.x; y/ unites the dynamic sets that contain x and y, say S x and S y , into a

new set that is the union of these two sets. We assume that the two sets are dis-

joint prior to the operation. The representative of the resulting set is any member

of S x [ S y , although many implementations of UNION speciﬁcally choose the

representative of either S x or S y as the new representative. Since we require

the sets in the collection to be disjoint, conceptually we destroy sets S x and S y ,

removing them from the collection S . In practice, we often absorb the elements

of one of the sets into the other set.

FIND-SET.x/ returns a pointer to the representative of the (unique) set contain-

ing x.

Throughout this chapter, we shall analyze the running times of disjoint-set data

structures in terms of two parameters: n, the number of MAKE-SET operations,

and m, the total number of MAKE-SET, UNION, and FIND-SET operations. Since

the sets are disjoint, each UNION operation reduces the number of sets by one.

After n 1 UNION operations, therefore, only one set remains. The number of

UNION operations is thus at most n 1. Note also that since the MAKE-SET

operations are included in the total number of operations m, we have m n. We

assume that the n MAKE-SET operations are the ﬁrst n operations performed.

An application of disjoint-set data structures

One of the many applications of disjoint-set data structures arises in determin-

ing the connected components of an undirected graph (see Section B.4). Fig-

ure 21.1(a), for example, shows a graph with four connected components.

The procedure CONNECTED-COMPONENTS that follows uses the disjoint-set

operations to compute the connected components of a graph. Once CONNECTED-

COMPONENTS has preprocessed the graph, the procedure SAME-COMPONENT

answers queries about whether two vertices are in the same connected component. 1

(In pseudocode, we denote the set of vertices of a graph G by G:V and the set of

edges by G:E.)

1

When the edges of the graph are static—not changing over time—we can compute the connected

components faster by using depth-ﬁrst search (Exercise 22.3-12). Sometimes, however, the edges

are added dynamically and we need to maintain the connected components as each edge is added. In

this case, the implementation given here can be more efﬁcient than running a new depth-ﬁrst search

for each new edge.

21.1 Disjoint-set operations 563

a b

c d

e f

g

h

i

j

Edge processed

initial sets

(b,d)

(e,g)

(a,c)

(h,i)

(a,b)

(e, f )

(b,c)

{a,b,c,d}

{a,b,c,d}

{a,c}

{a,c}

{a}

{a}

{a}

{a,b,c,d}

{b,d}

{b,d}

{b,d}

{b,d}

{b}

{c}

{c}

{c} {d}

{e, f,g}

{e, f,g}

{e,g}

{e,g}

{e,g}

{e,g}

{e}

{e} {f}

{f}

{f}

{f}

{f}

{f}

{g}

{g}

{h,i}

{h,i}

{h,i}

{h,i}

{h}

{h}

{h}

{h} {i}

{i}

{i}

{i}

{j}

{j}

{j}

{j}

{j}

{j}

{j}

{j}

Collection of disjoint sets

(a)

(b)

Figure 21.1 (a) A graph with four connected components: fa; b; c; dg, fe; f; gg, fh; ig, and fj g.

(b) The collection of disjoint sets after processing each edge.

CONNECTED-COMPONENTS.G/

1 for each vertex 2 G:V

2 MAKE-SET./

3 for each edge .u; / 2 G:E

4 if FIND-SET.u/ ¤ FIND-SET./

5 UNION.u; /

SAME-COMPONENT.u; /

1 if FIND-SET.u/ == FIND-SET./

2 return TRUE

3 else return FALSE

The procedure CONNECTED-COMPONENTS initially places each vertex in its

own set. Then, for each edge .u; /, it unites the sets containing u and . By

Exercise 21.1-2, after processing all the edges, two vertices are in the same con-

nected component if and only if the corresponding objects are in the same set.

Thus, CONNECTED-COMPONENTS computes sets in such a way that the proce-

dure SAME-COMPONENT can determine whether two vertices are in the same con-

564 Chapter 21 Data Structures for Disjoint Sets

nected component. Figure 21.1(b) illustrates how CONNECTED-COMPONENTS

computes the disjoint sets.

In an actual implementation of this connected-components algorithm, the repre-

sentations of the graph and the disjoint-set data structure would need to reference

each other. That is, an object representing a vertex would contain a pointer to

the corresponding disjoint-set object, and vice versa. These programming details

depend on the implementation language, and we do not address them further here.

Exercises

21.1-1

Suppose that CONNECTED-COMPONENTS is run on the undirected graph G D

.V; E/, where V D fa; b; c; d; e; f; g; h; i; j; kg and the edges of E are pro-

cessed in the order .d; i/; .f; k/; .g; i/; .b; g/; .a; h/; .i; j /; .d; k/; .b; j /; .d; f /;

.g; j /; .a; e/. List the vertices in each connected component after each iteration of

lines 3–5.

21.1-2

Show that after all edges are processed by CONNECTED-COMPONENTS, two ver-

tices are in the same connected component if and only if they are in the same set.

21.1-3

During the execution of CONNECTED-COMPONENTS on an undirected graph G D

.V; E/ with k connected components, how many times is FIND-SET called? How

many times is UNION called? Express your answers in terms of jV j, jEj, and k.

21.2 Linked-list representation of disjoint sets

Figure 21.2(a) shows a simple way to implement a disjoint-set data structure: each

set is represented by its own linked list. The object for each set has attributes head,

pointing to the ﬁrst object in the list, and tail, pointing to the last object. Each

object in the list contains a set member, a pointer to the next object in the list, and

a pointer back to the set object. Within each linked list, the objects may appear in

any order. The representative is the set member in the ﬁrst object in the list.

With this linked-list representation, both MAKE-SET and FIND-SET are easy,

requiring O.1/ time. To carry out MAKE-SET.x/, we create a new linked list

whose only object is x. For FIND-SET.x/, we just follow the pointer from x back

to its set object and then return the member in the object that head points to. For

example, in Figure 21.2(a), the call FIND-SET.g/ would return f .

21.2 Linked-list representation of disjoint sets 565

f g d c h e b

(a)

(b)

head

tail

S

1

c h e

head

tail

S

2

b f g d

head

tail

S

1

Figure 21.2 (a) Linked-list representations of two sets. Set S1 contains members d, f , and g, with

representative f , and set S2 contains members b, c, e, and h, with representative c. Each object in

the list contains a set member, a pointer to the next object in the list, and a pointer back to the set

object. Each set object has pointers head and tail to the ﬁrst and last objects, respectively. (b) The

result of UNION.g; e/, which appends the linked list containing e to the linked list containing g. The

representative of the resulting set is f . The set object for e’s list, S2, is destroyed.

A simple implementation of union

The simplest implementation of the UNION operation using the linked-list set rep-

resentation takes signiﬁcantly more time than MAKE-SET or FIND-SET. As Fig-

ure 21.2(b) shows, we perform UNION.x; y/ by appending y’s list onto the end

of x’s list. The representative of x’s list becomes the representative of the resulting

set. We use the tail pointer for x’s list to quickly ﬁnd where to append y’s list. Be-

cause all members of y’s list join x’s list, we can destroy the set object for y’s list.

Unfortunately, we must update the pointer to the set object for each object origi-

nally on y’s list, which takes time linear in the length of y’s list. In Figure 21.2, for

example, the operation UNION.g; e/ causes pointers to be updated in the objects

for b, c, e, and h.

In fact, we can easily construct a sequence of m operations on n objects that

requires ‚.n 2 / time. Suppose that we have objects x 1 ; x 2 ; : : : ; x n . We execute

the sequence of n MAKE-SET operations followed by n 1 UNION operations

shown in Figure 21.3, so that m D 2n 1. We spend ‚.n/ time performing the n

MAKE-SET operations. Because the ith UNION operation updates i objects, the

total number of objects updated by all n 1 UNION operations is

566 Chapter 21 Data Structures for Disjoint Sets

Operation Number of objects updated

MAKE-SET.x1/ 1

MAKE-SET.x2/ 1

:

:

:

:

:

:

MAKE-SET.xn/ 1

UNION.x2; x1/ 1

UNION.x3; x2/ 2

UNION.x4; x3/ 3

:

:

:

:

:

:

UNION.xn; xn1/ n 1

Figure 21.3 A sequence of 2n 1 operations on n objects that takes ‚.n

2

/ time, or ‚.n/ time

per operation on average, using the linked-list set representation and the simple implementation of

UNION.

n1 X

iD1

i D ‚.n

2

/ :

The total number of operations is 2n1, and so each operation on average requires

‚.n/ time. That is, the amortized time of an operation is ‚.n/.

A weighted-union heuristic

In the worst case, the above implementation of the UNION procedure requires an

average of ‚.n/ time per call because we may be appending a longer list onto

a shorter list; we must update the pointer to the set object for each member of

the longer list. Suppose instead that each list also includes the length of the list

(which we can easily maintain) and that we always append the shorter list onto the

longer, breaking ties arbitrarily. With this simple weighted-union heuristic, a sin-

gle UNION operation can still take .n/ time if both sets have .n/ members. As

the following theorem shows, however, a sequence of m MAKE-SET, UNION, and

FIND-SET operations, n of which are MAKE-SET operations, takes O.m C n lg n/

time.

Theorem 21.1

Using the linked-list representation of disjoint sets and the weighted-union heuris-

tic, a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which

are MAKE-SET operations, takes O.m C n lg n/ time.

21.2 Linked-list representation of disjoint sets 567

Proof Because each UNION operation unites two disjoint sets, we perform at

most n1 UNION operations over all. We now bound the total time taken by these

UNION operations. We start by determining, for each object, an upper bound on the

number of times the object’s pointer back to its set object is updated. Consider a

particular object x. We know that each time x’s pointer was updated, x must have

started in the smaller set. The ﬁrst time x’s pointer was updated, therefore, the

resulting set must have had at least 2 members. Similarly, the next time x’s pointer

was updated, the resulting set must have had at least 4 members. Continuing on,

we observe that for any k n, after x’s pointer has been updated dlg ke times,

the resulting set must have at least k members. Since the largest set has at most n

members, each object’s pointer is updated at most dlg ne times over all the UNION

operations. Thus the total time spent updating object pointers over all UNION

operations is O.n lg n/. We must also account for updating the tail pointers and

the list lengths, which take only ‚.1/ time per UNION operation. The total time

spent in all UNION operations is thus O.n lg n/.

The time for the entire sequence of m operations follows easily. Each MAKE-

SET and FIND-SET operation takes O.1/ time, and there are O.m/ of them. The

total time for the entire sequence is thus O.m C n lg n/.

Exercises

21.2-1

Write pseudocode for MAKE-SET, FIND-SET, and UNION using the linked-list

representation and the weighted-union heuristic. Make sure to specify the attributes

that you assume for set objects and list objects.

21.2-2

Show the data structure that results and the answers returned by the FIND-SET

operations in the following program. Use the linked-list representation with the

weighted-union heuristic.

1 for i D 1 to 16

2 MAKE-SET.x i /

3 for i D 1 to 15 by 2

4 UNION.x i ; x iC1 /

5 for i D 1 to 13 by 4

6 UNION.x i ; x iC2 /

7 UNION.x 1 ; x 5 /

8 UNION.x 11 ; x 13 /

9 UNION.x 1 ; x 10 /

10 FIND-SET.x 2 /

11 FIND-SET.x 9 /

568 Chapter 21 Data Structures for Disjoint Sets

Assume that if the sets containing x i and x j have the same size, then the operation

UNION.x i ; x j / appends x j ’s list onto x i ’s list.

21.2-3

Adapt the aggregate proof of Theorem 21.1 to obtain amortized time bounds

of O.1/ for MAKE-SET and FIND-SET and O.lg n/ for UNION using the linked-

list representation and the weighted-union heuristic.

21.2-4

Give a tight asymptotic bound on the running time of the sequence of operations in

Figure 21.3 assuming the linked-list representation and the weighted-union heuris-

tic.

21.2-5

Professor Gompers suspects that it might be possible to keep just one pointer in

each set object, rather than two (head and tail), while keeping the number of point-

ers in each list element at two. Show that the professor’s suspicion is well founded

by describing how to represent each set by a linked list such that each operation

has the same running time as the operations described in this section. Describe

also how the operations work. Your scheme should allow for the weighted-union

heuristic, with the same effect as described in this section. (Hint: Use the tail of a

linked list as its set’s representative.)

21.2-6

Suggest a simple change to the UNION procedure for the linked-list representation

that removes the need to keep the tail pointer to the last object in each list. Whether

or not the weighted-union heuristic is used, your change should not change the

asymptotic running time of the UNION procedure. (Hint: Rather than appending

one list to another, splice them together.)

21.3 Disjoint-set forests

In a faster implementation of disjoint sets, we represent sets by rooted trees, with

each node containing one member and each tree representing one set. In a disjoint-

set forest, illustrated in Figure 21.4(a), each member points only to its parent. The

root of each tree contains the representative and is its own parent. As we shall

see, although the straightforward algorithms that use this representation are no

faster than ones that use the linked-list representation, by introducing two heuris-

tics—“union by rank” and “path compression”—we can achieve an asymptotically

optimal disjoint-set data structure.

21.3 Disjoint-set forests 569

c

h e

b

f

d

g

(a)

f

c

h e

b

d

g

(b)

Figure 21.4 A disjoint-set forest. (a) Two trees representing the two sets of Figure 21.2. The

tree on the left represents the set fb; c; e; hg, with c as the representative, and the tree on the right

represents the set fd; f; gg, with f as the representative. (b) The result of UNION.e; g/.

We perform the three disjoint-set operations as follows. A MAKE-SET operation

simply creates a tree with just one node. We perform a FIND-SET operation by

following parent pointers until we ﬁnd the root of the tree. The nodes visited on

this simple path toward the root constitute the ﬁnd path. A UNION operation,

shown in Figure 21.4(b), causes the root of one tree to point to the root of the other.

Heuristics to improve the running time

So far, we have not improved on the linked-list implementation. A sequence of

n 1 UNION operations may create a tree that is just a linear chain of n nodes. By

using two heuristics, however, we can achieve a running time that is almost linear

in the total number of operations m.

The ﬁrst heuristic, union by rank, is similar to the weighted-union heuristic we

used with the linked-list representation. The obvious approach would be to make

the root of the tree with fewer nodes point to the root of the tree with more nodes.

Rather than explicitly keeping track of the size of the subtree rooted at each node,

we shall use an approach that eases the analysis. For each node, we maintain a

rank, which is an upper bound on the height of the node. In union by rank, we

make the root with smaller rank point to the root with larger rank during a UNION

operation.

The second heuristic, path compression, is also quite simple and highly effec-

tive. As shown in Figure 21.5, we use it during FIND-SET operations to make each

node on the ﬁnd path point directly to the root. Path compression does not change

any ranks.

570 Chapter 21 Data Structures for Disjoint Sets

a

b

c

d

e

f

a b c d e

f

(a) (b)

Figure 21.5 Path compression during the operation FIND-SET. Arrows and self-loops at roots are

omitted. (a) A tree representing a set prior to executing FIND-SET.a/. Triangles represent subtrees

whose roots are the nodes shown. Each node has a pointer to its parent. (b) The same set after

executing FIND-SET.a/. Each node on the ﬁnd path now points directly to the root.

Pseudocode for disjoint-set forests

To implement a disjoint-set forest with the union-by-rank heuristic, we must keep

track of ranks. With each node x, we maintain the integer value x:rank, which is

an upper bound on the height of x (the number of edges in the longest simple path

between x and a descendant leaf). When MAKE-SET creates a singleton set, the

single node in the corresponding tree has an initial rank of 0. Each FIND-SET oper-

ation leaves all ranks unchanged. The UNION operation has two cases, depending

on whether the roots of the trees have equal rank. If the roots have unequal rank,

we make the root with higher rank the parent of the root with lower rank, but the

ranks themselves remain unchanged. If, instead, the roots have equal ranks, we

arbitrarily choose one of the roots as the parent and increment its rank.

Let us put this method into pseudocode. We designate the parent of node x

by x:p. The LINK procedure, a subroutine called by UNION, takes pointers to two

roots as inputs.

21.3 Disjoint-set forests 571

MAKE-SET.x/

1 x:p D x

2 x:rank D 0

UNION.x; y/

1 LINK.FIND-SET.x/; FIND-SET.y//

LINK.x; y/

1 if x:rank > y:rank

2 y:p D x

3 else x:p D y

4 if x:rank == y:rank

5 y:rank D y:rank C 1

The FIND-SET procedure with path compression is quite simple:

FIND-SET.x/

1 if x ¤ x:p

2 x:p D FIND-SET.x:p/

3 return x:p

The FIND-SET procedure is a two-pass method: as it recurses, it makes one pass

up the ﬁnd path to ﬁnd the root, and as the recursion unwinds, it makes a second

pass back down the ﬁnd path to update each node to point directly to the root. Each

call of FIND-SET.x/ returns x:p in line 3. If x is the root, then FIND-SET skips

line 2 and instead returns x:p, which is x; this is the case in which the recursion

bottoms out. Otherwise, line 2 executes, and the recursive call with parameter x:p

returns a pointer to the root. Line 2 updates node x to point directly to the root,

and line 3 returns this pointer.

Effect of the heuristics on the running time

Separately, either union by rank or path compression improves the running time of

the operations on disjoint-set forests, and the improvement is even greater when

we use the two heuristics together. Alone, union by rank yields a running time

of O.m lg n/ (see Exercise 21.4-4), and this bound is tight (see Exercise 21.3-3).

Although we shall not prove it here, for a sequence of n MAKE-SET opera-

tions (and hence at most n 1 UNION operations) and f FIND-SET opera-

tions, the path-compression heuristic alone gives a worst-case running time of

‚.n C f .1 C log

2Cf=n

n//.

572 Chapter 21 Data Structures for Disjoint Sets

When we use both union by rank and path compression, the worst-case running

time is O.m ˛.n//, where ˛.n/ is a very slowly growing function, which we de-

ﬁne in Section 21.4. In any conceivable application of a disjoint-set data structure,

˛.n/ 4; thus, we can view the running time as linear in m in all practical situa-

tions. Strictly speaking, however, it is superlinear. In Section 21.4, we prove this

upper bound.

Exercises

21.3-1

Redo Exercise 21.2-2 using a disjoint-set forest with union by rank and path com-

pression.

21.3-2

Write a nonrecursive version of FIND-SET with path compression.

21.3-3

Give a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which

are MAKE-SET operations, that takes .m lg n/ time when we use union by rank

only.

21.3-4

Suppose that we wish to add the operation PRINT-SET.x/, which is given a node x

and prints all the members of x’s set, in any order. Show how we can add just

a single attribute to each node in a disjoint-set forest so that PRINT-SET.x/ takes

time linear in the number of members of x’s set and the asymptotic running times

of the other operations are unchanged. Assume that we can print each member of

the set in O.1/ time.

21.3-5 ?

Show that any sequence of m MAKE-SET, FIND-SET, and LINK operations, where

all the LINK operations appear before any of the FIND-SET operations, takes only

O.m/ time if we use both path compression and union by rank. What happens in

the same situation if we use only the path-compression heuristic?

21.4 Analysis of union by rank with path compression 573

? 21.4 Analysis of union by rank with path compression

As noted in Section 21.3, the combined union-by-rank and path-compression heu-

ristic runs in time O.m ˛.n// for m disjoint-set operations on n elements. In this

section, we shall examine the function ˛ to see just how slowly it grows. Then we

prove this running time using the potential method of amortized analysis.

A very quickly growing function and its very slowly growing inverse

For integers k 0 and j 1, we deﬁne the function A k .j / as

A k .j / D

(

j C 1 if k D 0 ;

A

.j C1/

k1

.j / if k 1 ;

where the expression A

.j C1/

k1

.j / uses the functional-iteration notation given in Sec-

tion 3.2. Speciﬁcally, A

.0/

k1

.j / D j and A

.i/

k1

.j / D A k1 .A

.i1/

k1

.j // for i 1.

We will refer to the parameter k as the level of the function A.

The function A k .j / strictly increases with both j and k. To see just how quickly

this function grows, we ﬁrst obtain closed-form expressions for A 1 .j / and A 2 .j /.

Lemma 21.2

For any integer j 1, we have A 1 .j / D 2j C 1.

Proof We ﬁrst use induction on i to show that A

.i/

0

.j / D j Ci. For the base case,

we have A

.0/

0

.j / D j D j C 0. For the inductive step, assume that A

.i1/

0

.j / D

j C .i 1/. Then A

.i/

0 .j / D A 0 .A

.i1/

0 .j // D .j C .i 1// C 1 D j C i. Finally,

we note that A 1 .j / D A

.j C1/

0

.j / D j C .j C 1/ D 2j C 1.

Lemma 21.3

For any integer j 1, we have A 2 .j / D 2 j C1 .j C 1/ 1.

Proof We ﬁrst use induction on i to show that A

.i/

1

.j / D 2 i .j C 1/ 1. For

the base case, we have A

.0/

1

.j / D j D 2 0 .j C 1/ 1. For the inductive step,

assume that A

.i1/

1 .j / D 2 i1 .j C 1/ 1. Then A

.i/

1 .j / D A 1 .A

.i1/

1 .j // D

A 1 .2 i1 .j C 1/ 1/ D 2.2 i1 .j C1/1/C1 D 2 i .j C1/2C1 D 2 i .j C1/1.

Finally, we note that A 2 .j / D A

.j C1/

1

.j / D 2 j C1 .j C 1/ 1.

Now we can see how quickly A k .j / grows by simply examining A k .1/ for levels

k D 0; 1; 2; 3; 4. From the deﬁnition of A 0 .k/ and the above lemmas, we have

A 0 .1/ D 1 C 1 D 2, A 1 .1/ D 2 1 C 1 D 3, and A 2 .1/ D 2 1C1 .1 C 1/ 1 D 7.

574 Chapter 21 Data Structures for Disjoint Sets

We also have

A 3 .1/ D A

.2/

2

.1/

D A 2 .A 2 .1//

D A 2 .7/

D 2

8

8 1

D 2

11

1

D 2047

and

A 4 .1/ D A

.2/

3

.1/

D A 3 .A 3 .1//

D A 3 .2047/

D A

.2048/

2

.2047/

A 2 .2047/

D 2

2048

2048 1

> 2

2048

D .2

4

/

512

D 16

512

10

80

;

which is the estimated number of atoms in the observable universe. (The symbol

“  
” denotes the “much-greater-than” relation.)

We deﬁne the inverse of the function A k .n/, for integer n 0, by

˛.n/ D min fk W A k .1/ ng :

In words, ˛.n/ is the lowest level k for which A k .1/ is at least n. From the above

values of A k .1/, we see that

˛.n/ D

˚ 0 for 0 n 2 ;

1 for n D 3 ;

2 for 4 n 7 ;

3 for 8 n 2047 ;

4 for 2048 n A 4 .1/ :

It is only for values of n so large that the term “astronomical” understates them

(greater than A 4 .1/, a huge number) that ˛.n/ > 4, and so ˛.n/ 4 for all

practical purposes.

21.4 Analysis of union by rank with path compression 575

Properties of ranks

In the remainder of this section, we prove an O.m˛.n// bound on the running time

of the disjoint-set operations with union by rank and path compression. In order to

prove this bound, we ﬁrst prove some simple properties of ranks.

Lemma 21.4

For all nodes x, we have x:rank x:p:rank, with strict inequality if x ¤ x:p.

The value of x:rank is initially 0 and increases through time until x ¤ x:p; from

then on, x:rank does not change. The value of x:p:rank monotonically increases

over time.

Proof The proof is a straightforward induction on the number of operations, us-

ing the implementations of MAKE-SET, UNION, and FIND-SET that appear in

Section 21.3. We leave it as Exercise 21.4-1.

Corollary 21.5

As we follow the simple path from any node toward a root, the node ranks strictly

increase.

Lemma 21.6

Every node has rank at most n 1.

Proof Each node’s rank starts at 0, and it increases only upon LINK operations.

Because there are at most n 1 UNION operations, there are also at most n 1

LINK operations. Because each LINK operation either leaves all ranks alone or

increases some node’s rank by 1, all ranks are at most n 1.

Lemma 21.6 provides a weak bound on ranks. In fact, every node has rank at

most blg nc (see Exercise 21.4-2). The looser bound of Lemma 21.6 will sufﬁce

for our purposes, however.

Proving the time bound

We shall use the potential method of amortized analysis (see Section 17.3) to prove

the O.m ˛.n// time bound. In performing the amortized analysis, we will ﬁnd it

convenient to assume that we invoke the LINK operation rather than the UNION

operation. That is, since the parameters of the LINK procedure are pointers to two

roots, we act as though we perform the appropriate FIND-SET operations sepa-

rately. The following lemma shows that even if we count the extra FIND-SET op-

erations induced by UNION calls, the asymptotic running time remains unchanged.

576 Chapter 21 Data Structures for Disjoint Sets

Lemma 21.7

Suppose we convert a sequence S 0

of m 0

MAKE-SET, UNION, and FIND-SET op-

erations into a sequence S of m MAKE-SET, LINK, and FIND-SET operations by

turning each UNION into two FIND-SET operations followed by a LINK. Then, if

sequence S runs in O.m ˛.n// time, sequence S 0

runs in O.m 0 ˛.n// time.

Proof Since each UNION operation in sequence S 0

is converted into three opera-

tions in S, we have m 0 m 3m 0

. Since m D O.m 0 /, an O.m ˛.n// time bound

for the converted sequence S implies an O.m 0 ˛.n// time bound for the original

sequence S 0

.

In the remainder of this section, we shall assume that the initial sequence of m 0

MAKE-SET, UNION, and FIND-SET operations has been converted to a sequence

of m MAKE-SET, LINK, and FIND-SET operations. We now prove an O.m ˛.n//

time bound for the converted sequence and appeal to Lemma 21.7 to prove the

O.m 0 ˛.n// running time of the original sequence of m 0

operations.

Potential function

The potential function we use assigns a potential q .x/ to each node x in the

disjoint-set forest after q operations. We sum the node potentials for the poten-

tial of the entire forest: ˆ q D

P

x

q .x/, where ˆ q denotes the potential of the

forest after q operations. The forest is empty prior to the ﬁrst operation, and we

arbitrarily set ˆ 0 D 0. No potential ˆ q will ever be negative.

The value of q .x/ depends on whether x is a tree root after the qth operation.

If it is, or if x:rank D 0, then q .x/ D ˛.n/ x:rank.

Now suppose that after the qth operation, x is not a root and that x:rank 1.

We need to deﬁne two auxiliary functions on x before we can deﬁne q .x/. First

we deﬁne

level.x/ D max fk W x:p:rank A k .x:rank/g :

That is, level.x/ is the greatest level k for which A k , applied to x’s rank, is no

greater than x’s parent’s rank.

We claim that

0 level.x/ < ˛.n/ ; (21.1)

which we see as follows. We have

x:p:rank x:rank C 1 (by Lemma 21.4)

D A 0 .x:rank/ (by deﬁnition of A 0 .j /) ,

which implies that level.x/ 0, and we have

21.4 Analysis of union by rank with path compression 577

A ˛.n/ .x:rank/ A ˛.n/ .1/ (because A k .j / is strictly increasing)

n (by the deﬁnition of ˛.n/)

> x:p:rank (by Lemma 21.6) ,

which implies that level.x/ < ˛.n/. Note that because x:p:rank monotonically

increases over time, so does level.x/.

The second auxiliary function applies when x:rank 1:

iter.x/ D max

˚

i W x:p:rank A

.i/

level.x/

.x:rank/

:

That is, iter.x/ is the largest number of times we can iteratively apply A level.x/ ,

applied initially to x’s rank, before we get a value greater than x’s parent’s rank.

We claim that when x:rank 1, we have

1 iter.x/ x:rank ; (21.2)

which we see as follows. We have

x:p:rank A level.x/ .x:rank/ (by deﬁnition of level.x/)

D A

.1/

level.x/

.x:rank/ (by deﬁnition of functional iteration) ,

which implies that iter.x/ 1, and we have

A

.x: rankC1/

level.x/

.x:rank/ D A level.x/C1 .x:rank/ (by deﬁnition of A k .j /)

> x:p:rank (by deﬁnition of level.x/) ,

which implies that iter.x/ x:rank. Note that because x:p:rank monotonically

increases over time, in order for iter.x/ to decrease, level.x/ must increase. As long

as level.x/ remains unchanged, iter.x/ must either increase or remain unchanged.

With these auxiliary functions in place, we are ready to deﬁne the potential of

node x after q operations:

q .x/ D

(

˛.n/ x:rank if x is a root or x:rank D 0 ;

.˛.n/ level.x//x:rank iter.x/ if x is not a root and x:rank 1 :

We next investigate some useful properties of node potentials.

Lemma 21.8

For every node x, and for all operation counts q, we have

0 q .x/ ˛.n/ x:rank :

578 Chapter 21 Data Structures for Disjoint Sets

Proof If x is a root or x:rank D 0, then q .x/ D ˛.n/x:rank by deﬁnition. Now

suppose that x is not a root and that x:rank 1. We obtain a lower bound on q .x/

by maximizing level.x/ and iter.x/. By the bound (21.1), level.x/ ˛.n/1, and

by the bound (21.2), iter.x/ x:rank. Thus,

q .x/ D .˛.n/ level.x// x:rank iter.x/

.˛.n/ .˛.n/ 1// x:rank x:rank

D x:rank x:rank

D 0 :

Similarly, we obtain an upper bound on q .x/ by minimizing level.x/ and iter.x/.

By the bound (21.1), level.x/ 0, and by the bound (21.2), iter.x/ 1. Thus,

q .x/ .˛.n/ 0/ x:rank 1

D ˛.n/ x:rank 1

< ˛.n/ x:rank :

Corollary 21.9

If node x is not a root and x:rank > 0, then q .x/ < ˛.n/ x:rank.

Potential changes and amortized costs of operations

We are now ready to examine how the disjoint-set operations affect node potentials.

With an understanding of the change in potential due to each operation, we can

determine each operation’s amortized cost.

Lemma 21.10

Let x be a node that is not a root, and suppose that the qth operation is either a

LINK or FIND-SET. Then after the qth operation, q .x/ q1 .x/. Moreover, if

x:rank 1 and either level.x/ or iter.x/ changes due to the qth operation, then

q .x/ q1 .x/ 1. That is, x’s potential cannot increase, and if it has positive

rank and either level.x/ or iter.x/ changes, then x’s potential drops by at least 1.

Proof Because x is not a root, the qth operation does not change x:rank, and

because n does not change after the initial n MAKE-SET operations, ˛.n/ remains

unchanged as well. Hence, these components of the formula for x’s potential re-

main the same after the qth operation. If x:rank D 0, then q .x/ D q1 .x/ D 0.

Now assume that x:rank 1.

Recall that level.x/ monotonically increases over time. If the qth operation

leaves level.x/ unchanged, then iter.x/ either increases or remains unchanged.

If both level.x/ and iter.x/ are unchanged, then q .x/ D q1 .x/. If level.x/

21.4 Analysis of union by rank with path compression 579

is unchanged and iter.x/ increases, then it increases by at least 1, and so

q .x/ q1 .x/ 1.

Finally, if the qth operation increases level.x/, it increases by at least 1, so that

the value of the term .˛.n/ level.x// x:rank drops by at least x:rank. Be-

cause level.x/ increased, the value of iter.x/ might drop, but according to the

bound (21.2), the drop is by at most x:rank 1. Thus, the increase in poten-

tial due to the change in iter.x/ is less than the decrease in potential due to the

change in level.x/, and we conclude that q .x/ q1 .x/ 1.

Our ﬁnal three lemmas show that the amortized cost of each MAKE-SET, LINK,

and FIND-SET operation is O.˛.n//. Recall from equation (17.2) that the amor-

tized cost of each operation is its actual cost plus the increase in potential due to

the operation.

Lemma 21.11

The amortized cost of each MAKE-SET operation is O.1/.

Proof Suppose that the qth operation is MAKE-SET.x/. This operation creates

node x with rank 0, so that q .x/ D 0. No other ranks or potentials change, and

so ˆ q D ˆ q1 . Noting that the actual cost of the MAKE-SET operation is O.1/

completes the proof.

Lemma 21.12

The amortized cost of each LINK operation is O.˛.n//.

Proof Suppose that the qth operation is LINK.x; y/. The actual cost of the LINK

operation is O.1/. Without loss of generality, suppose that the LINK makes y the

parent of x.

To determine the change in potential due to the LINK, we note that the only

nodes whose potentials may change are x, y, and the children of y just prior to the

operation. We shall show that the only node whose potential can increase due to

the LINK is y, and that its increase is at most ˛.n/:

By Lemma 21.10, any node that is y’s child just before the LINK cannot have

its potential increase due to the LINK.

From the deﬁnition of q .x/, we see that, since x was a root just before the qth

operation, q1 .x/ D ˛.n/x:rank. If x:rank D 0, then q .x/ D q1 .x/ D 0.

Otherwise,

q .x/ < ˛.n/ x:rank (by Corollary 21.9)

D q1 .x/ ;

and so x’s potential decreases.

580 Chapter 21 Data Structures for Disjoint Sets

Because y is a root prior to the LINK, q1 .y/ D ˛.n/ y:rank. The LINK

operation leaves y as a root, and it either leaves y’s rank alone or it increases y’s

rank by 1. Therefore, either q .y/ D q1 .y/ or q .y/ D q1 .y/ C ˛.n/.

The increase in potential due to the LINK operation, therefore, is at most ˛.n/.

The amortized cost of the LINK operation is O.1/ C ˛.n/ D O.˛.n//.

Lemma 21.13

The amortized cost of each FIND-SET operation is O.˛.n//.

Proof Suppose that the qth operation is a FIND-SET and that the ﬁnd path con-

tains s nodes. The actual cost of the FIND-SET operation is O.s/. We shall

show that no node’s potential increases due to the FIND-SET and that at least

max.0; s .˛.n/ C 2// nodes on the ﬁnd path have their potential decrease by

at least 1.

To see that no node’s potential increases, we ﬁrst appeal to Lemma 21.10 for all

nodes other than the root. If x is the root, then its potential is ˛.n/ x:rank, which

does not change.

Now we show that at least max.0; s .˛.n/ C 2// nodes have their potential

decrease by at least 1. Let x be a node on the ﬁnd path such that x:rank > 0

and x is followed somewhere on the ﬁnd path by another node y that is not a root,

where level.y/ D level.x/ just before the FIND-SET operation. (Node y need not

immediately follow x on the ﬁnd path.) All but at most ˛.n/ C 2 nodes on the ﬁnd

path satisfy these constraints on x. Those that do not satisfy them are the ﬁrst node

on the ﬁnd path (if it has rank 0), the last node on the path (i.e., the root), and the

last node w on the path for which level.w/ D k, for each k D 0; 1; 2; : : : ; ˛.n/1.

Let us ﬁx such a node x, and we shall show that x’s potential decreases by at

least 1. Let k D level.x/ D level.y/. Just prior to the path compression caused by

the FIND-SET, we have

x:p:rank A

.iter.x//

k

.x:rank/ (by deﬁnition of iter.x/) ,

y:p:rank A k .y:rank/ (by deﬁnition of level.y/) ,

y:rank x:p:rank (by Corollary 21.5 and because

y follows x on the ﬁnd path) .

Putting these inequalities together and letting i be the value of iter.x/ before path

compression, we have

y:p:rank A k .y:rank/

A k .x:p:rank/ (because A k .j / is strictly increasing)

A k .A

.iter.x//

k

.x:rank//

D A

.iC1/

k

.x:rank/ :

21.4 Analysis of union by rank with path compression 581

Because path compression will make x and y have the same parent, we know

that after path compression, x:p:rank D y:p:rank and that the path compression

does not decrease y:p:rank. Since x:rank does not change, after path compression

we have that x:p:rank A

.iC1/

k

.x:rank/. Thus, path compression will cause ei-

ther iter.x/ to increase (to at least i C 1) or level.x/ to increase (which occurs if

iter.x/ increases to at least x:rank C 1). In either case, by Lemma 21.10, we have

q .x/ q1 .x/ 1. Hence, x’s potential decreases by at least 1.

The amortized cost of the FIND-SET operation is the actual cost plus the change

in potential. The actual cost is O.s/, and we have shown that the total potential

decreases by at least max.0; s .˛.n/ C 2//. The amortized cost, therefore, is at

most O.s/ .s .˛.n/ C 2// D O.s/ s C O.˛.n// D O.˛.n//, since we can

scale up the units of potential to dominate the constant hidden in O.s/.

Putting the preceding lemmas together yields the following theorem.

Theorem 21.14

A sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are

MAKE-SET operations, can be performed on a disjoint-set forest with union by

rank and path compression in worst-case time O.m ˛.n//.

Proof Immediate from Lemmas 21.7, 21.11, 21.12, and 21.13.

Exercises

21.4-1

Prove Lemma 21.4.

21.4-2

Prove that every node has rank at most blg nc.

21.4-3

In light of Exercise 21.4-2, how many bits are necessary to store x:rank for each

node x?

21.4-4

Using Exercise 21.4-2, give a simple proof that operations on a disjoint-set forest

with union by rank but without path compression run in O.m lg n/ time.

21.4-5

Professor Dante reasons that because node ranks increase strictly along a simple

path to the root, node levels must monotonically increase along the path. In other

582 Chapter 21 Data Structures for Disjoint Sets

words, if x:rank > 0 and x:p is not a root, then level.x/ level.x:p/. Is the

professor correct?

21.4-6 ?

Consider the function ˛ 0 .n/ D min fk W A k .1/ lg.n C 1/g. Show that ˛ 0 .n/ 3

for all practical values of n and, using Exercise 21.4-2, show how to modify the

potential-function argument to prove that we can perform a sequence of m MAKE-

SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, on

a disjoint-set forest with union by rank and path compression in worst-case time

O.m ˛ 0 .n//.

Problems

21-1 Off-line minimum

The off-line minimum problem asks us to maintain a dynamic set T of elements

from the domain f1; 2; : : : ; ng under the operations INSERT and EXTRACT-MIN.

We are given a sequence S of n INSERT and m EXTRACT-MIN calls, where each

key in f1; 2; : : : ; ng is inserted exactly once. We wish to determine which key

is returned by each EXTRACT-MIN call. Speciﬁcally, we wish to ﬁll in an array

extractedŒ1 : : m, where for i D 1; 2; : : : ; m, extractedŒi is the key returned by

the ith EXTRACT-MIN call. The problem is “off-line” in the sense that we are

allowed to process the entire sequence S before determining any of the returned

keys.

a. In the following instance of the off-line minimum problem, each operation

INSERT.i/ is represented by the value of i and each EXTRACT-MIN is rep-

resented by the letter E:

4; 8; E; 3; E; 9; 2; 6; E; E; E; 1; 7; E; 5 :

Fill in the correct values in the extracted array.

To develop an algorithm for this problem, we break the sequence S into homoge-

neous subsequences. That is, we represent S by

I 1 ; E; I 2 ; E; I 3 ; : : : ; I m ; E; I mC1 ;

where each E represents a single EXTRACT-MIN call and each I j represents a (pos-

sibly empty) sequence of INSERT calls. For each subsequence I j , we initially place

the keys inserted by these operations into a set K j , which is empty if I j is empty.

We then do the following:

Problems for Chapter 21 583

OFF-LINE-MINIMUM.m; n/

1 for i D 1 to n

2 determine j such that i 2 K j

3 if j ¤ m C 1

4 extractedŒj D i

5 let l be the smallest value greater than j

for which set K l exists

6 K l D K j [ K l , destroying K j

7 return extracted

b. Argue that the array extracted returned by OFF-LINE-MINIMUM is correct.

c. Describe how to implement OFF-LINE-MINIMUM efﬁciently with a disjoint-

set data structure. Give a tight bound on the worst-case running time of your

implementation.

21-2 Depth determination

In the depth-determination problem, we maintain a forest F D fT i g of rooted

trees under three operations:

MAKE-TREE./ creates a tree whose only node is .

FIND-DEPTH./ returns the depth of node within its tree.

GRAFT.r; / makes node r, which is assumed to be the root of a tree, become the

child of node , which is assumed to be in a different tree than r but may or may

not itself be a root.

a. Suppose that we use a tree representation similar to a disjoint-set forest: :p

is the parent of node , except that :p D if is a root. Suppose further

that we implement GRAFT.r; / by setting r:p D and FIND-DEPTH./ by

following the ﬁnd path up to the root, returning a count of all nodes other than

encountered. Show that the worst-case running time of a sequence of m MAKE-

TREE, FIND-DEPTH, and GRAFT operations is ‚.m 2 /.

By using the union-by-rank and path-compression heuristics, we can reduce the

worst-case running time. We use the disjoint-set forest S D fS i g, where each

set S i (which is itself a tree) corresponds to a tree T i in the forest F . The tree

structure within a set S i , however, does not necessarily correspond to that of T i . In

fact, the implementation of S i does not record the exact parent-child relationships

but nevertheless allows us to determine any node’s depth in T i .

The key idea is to maintain in each node a “pseudodistance” :d, which is

deﬁned so that the sum of the pseudodistances along the simple path from to the

584 Chapter 21 Data Structures for Disjoint Sets

root of its set S i equals the depth of in T i . That is, if the simple path from to its

root in S i is 0 ; 1 ; : : : ; k , where 0 D and k is S i ’s root, then the depth of

in T i is

P k

j D0

j :d.

b. Give an implementation of MAKE-TREE.

c. Show how to modify FIND-SET to implement FIND-DEPTH. Your implemen-

tation should perform path compression, and its running time should be linear

in the length of the ﬁnd path. Make sure that your implementation updates

pseudodistances correctly.

d. Show how to implement GRAFT.r; /, which combines the sets containing r

and , by modifying the UNION and LINK procedures. Make sure that your

implementation updates pseudodistances correctly. Note that the root of a set S i

is not necessarily the root of the corresponding tree T i .

e. Give a tight bound on the worst-case running time of a sequence of m MAKE-

TREE, FIND-DEPTH, and GRAFT operations, n of which are MAKE-TREE op-

erations.

21-3 Tarjan’s off-line least-common-ancestors algorithm

The least common ancestor of two nodes u and in a rooted tree T is the node w

that is an ancestor of both u and and that has the greatest depth in T . In the

off-line least-common-ancestors problem, we are given a rooted tree T and an

arbitrary set P D ffu; gg of unordered pairs of nodes in T , and we wish to deter-

mine the least common ancestor of each pair in P .

To solve the off-line least-common-ancestors problem, the following procedure

performs a tree walk of T with the initial call LCA.T:root/. We assume that each

node is colored WHITE prior to the walk.

LCA.u/

1 MAKE-SET.u/

2 FIND-SET.u/:ancestor D u

3 for each child of u in T

4 LCA./

5 UNION.u; /

6 FIND-SET.u/:ancestor D u

7 u:color D BLACK

8 for each node such that fu; g 2 P

9 if :color == BLACK

10 print “The least common ancestor of”

u “and” “is” FIND-SET./:ancestor

Notes for Chapter 21 585

a. Argue that line 10 executes exactly once for each pair fu; g 2 P .

b. Argue that at the time of the call LCA.u/, the number of sets in the disjoint-set

data structure equals the depth of u in T .

c. Prove that LCA correctly prints the least common ancestor of u and for each

pair fu; g 2 P .

d. Analyze the running time of LCA, assuming that we use the implementation of

the disjoint-set data structure in Section 21.3.

Chapter notes

Many of the important results for disjoint-set data structures are due at least in part

to R. E. Tarjan. Using aggregate analysis, Tarjan [328, 330] gave the ﬁrst tight

upper bound in terms of the very slowly growing inverse y˛.m; n/ of Ackermann’s

function. (The function A k .j / given in Section 21.4 is similar to Ackermann’s

function, and the function ˛.n/ is similar to the inverse. Both ˛.n/ and y˛.m; n/

are at most 4 for all conceivable values of m and n.) An O.m lg

n/ upper bound

was proven earlier by Hopcroft and Ullman [5, 179]. The treatment in Section 21.4

is adapted from a later analysis by Tarjan [332], which is in turn based on an anal-

ysis by Kozen [220]. Harfst and Reingold [161] give a potential-based version of

Tarjan’s earlier bound.

Tarjan and van Leeuwen [333] discuss variants on the path-compression heuris-

tic, including “one-pass methods,” which sometimes offer better constant factors

in their performance than do two-pass methods. As with Tarjan’s earlier analyses

of the basic path-compression heuristic, the analyses by Tarjan and van Leeuwen

are aggregate. Harfst and Reingold [161] later showed how to make a small change

to the potential function to adapt their path-compression analysis to these one-pass

variants. Gabow and Tarjan [121] show that in certain applications, the disjoint-set

operations can be made to run in O.m/ time.

Tarjan [329] showed that a lower bound of .m y˛.m; n// time is required for

operations on any disjoint-set data structure satisfying certain technical conditions.

This lower bound was later generalized by Fredman and Saks [113], who showed

that in the worst case, .m y˛.m; n// .lg n/-bit words of memory must be accessed.

VI Graph Algorithms

Introduction

Graph problems pervade computer science, and algorithms for working with them

are fundamental to the ﬁeld. Hundreds of interesting computational problems are

couched in terms of graphs. In this part, we touch on a few of the more signiﬁcant

ones.

Chapter 22 shows how we can represent a graph in a computer and then discusses

algorithms based on searching a graph using either breadth-ﬁrst search or depth-

ﬁrst search. The chapter gives two applications of depth-ﬁrst search: topologically

sorting a directed acyclic graph and decomposing a directed graph into its strongly

connected components.

Chapter 23 describes how to compute a minimum-weight spanning tree of a

graph: the least-weight way of connecting all of the vertices together when each

edge has an associated weight. The algorithms for computing minimum spanning

trees serve as good examples of greedy algorithms (see Chapter 16).

Chapters 24 and 25 consider how to compute shortest paths between vertices

when each edge has an associated length or “weight.” Chapter 24 shows how to

ﬁnd shortest paths from a given source vertex to all other vertices, and Chapter 25

examines methods to compute shortest paths between every pair of vertices.

Finally, Chapter 26 shows how to compute a maximum ﬂow of material in a ﬂow

network, which is a directed graph having a speciﬁed source vertex of material, a

speciﬁed sink vertex, and speciﬁed capacities for the amount of material that can

traverse each directed edge. This general problem arises in many forms, and a

good algorithm for computing maximum ﬂows can help solve a variety of related

problems efﬁciently.

588 Part VI Graph Algorithms

When we characterize the running time of a graph algorithm on a given graph

G D .V; E/, we usually measure the size of the input in terms of the number of

vertices jV j and the number of edges jEj of the graph. That is, we describe the

size of the input with two parameters, not just one. We adopt a common notational

convention for these parameters. Inside asymptotic notation (such as O-notation

or ‚-notation), and only inside such notation, the symbol V denotes jV j and

the symbol E denotes jEj. For example, we might say, “the algorithm runs in

time O.VE/,” meaning that the algorithm runs in time O.jV j jEj/. This conven-

tion makes the running-time formulas easier to read, without risk of ambiguity.

Another convention we adopt appears in pseudocode. We denote the vertex set

of a graph G by G:V and its edge set by G:E. That is, the pseudocode views vertex

and edge sets as attributes of a graph.

22 Elementary Graph Algorithms

This chapter presents methods for representing a graph and for searching a graph.

Searching a graph means systematically following the edges of the graph so as to

visit the vertices of the graph. A graph-searching algorithm can discover much

about the structure of a graph. Many algorithms begin by searching their input

graph to obtain this structural information. Several other graph algorithms elabo-

rate on basic graph searching. Techniques for searching a graph lie at the heart of

the ﬁeld of graph algorithms.

Section 22.1 discusses the two most common computational representations of

graphs: as adjacency lists and as adjacency matrices. Section 22.2 presents a sim-

ple graph-searching algorithm called breadth-ﬁrst search and shows how to cre-

ate a breadth-ﬁrst tree. Section 22.3 presents depth-ﬁrst search and proves some

standard results about the order in which depth-ﬁrst search visits vertices. Sec-

tion 22.4 provides our ﬁrst real application of depth-ﬁrst search: topologically sort-

ing a directed acyclic graph. A second application of depth-ﬁrst search, ﬁnding the

strongly connected components of a directed graph, is the topic of Section 22.5.

22.1 Representations of graphs

We can choose between two standard ways to represent a graph G D .V; E/:

as a collection of adjacency lists or as an adjacency matrix. Either way applies

to both directed and undirected graphs. Because the adjacency-list representation

provides a compact way to represent sparse graphs—those for which jEj is much

less than jV j

2

—it is usually the method of choice. Most of the graph algorithms

presented in this book assume that an input graph is represented in adjacency-

list form. We may prefer an adjacency-matrix representation, however, when the

graph is dense—jEj is close to jV j

2

—or when we need to be able to tell quickly

if there is an edge connecting two given vertices. For example, two of the all-pairs

590 Chapter 22 Elementary Graph Algorithms

1 2

3

4 5

1

2

3

4

5

2 5

1

2

2

4 1 2

5 3

4

4 5 3

1 0 0 1

0 1 1 1

1 0 1 0

1 1 0 1

1 0 1 0

0

1

0

0

1

1 2 3 4 5

1

2

3

4

5

(a) (b) (c)

Figure 22.1 Two representations of an undirected graph. (a) An undirected graph G with 5 vertices

and 7 edges. (b) An adjacency-list representation of G. (c) The adjacency-matrix representation

of G.

1 2

5 4

1

2

3

4

5

2 4

5

6

2

4

6

5

1 0 1 0

0 0 0 1

0 0 0 1

1 0 0 0

0 0 1 0

0

0

0

0

0

1 2 3 4 5

1

2

3

4

5

(a) (b) (c)

3

6 6

6

6 0 0 0 0 0 1

0

0

1

0

0

Figure 22.2 Two representations of a directed graph. (a) A directed graph G with 6 vertices and 8

edges. (b) An adjacency-list representation of G. (c) The adjacency-matrix representation of G.

shortest-paths algorithms presented in Chapter 25 assume that their input graphs

are represented by adjacency matrices.

The adjacency-list representation of a graph G D .V; E/ consists of an ar-

ray Adj of jV j lists, one for each vertex in V . For each u 2 V , the adjacency list

AdjŒu contains all the vertices such that there is an edge .u; / 2 E. That is,

AdjŒu consists of all the vertices adjacent to u in G. (Alternatively, it may contain

pointers to these vertices.) Since the adjacency lists represent the edges of a graph,

in pseudocode we treat the array Adj as an attribute of the graph, just as we treat

the edge set E. In pseudocode, therefore, we will see notation such as G:AdjŒu.

Figure 22.1(b) is an adjacency-list representation of the undirected graph in Fig-

ure 22.1(a). Similarly, Figure 22.2(b) is an adjacency-list representation of the

directed graph in Figure 22.2(a).

If G is a directed graph, the sum of the lengths of all the adjacency lists is jEj,

since an edge of the form .u; / is represented by having appear in AdjŒu. If G is

22.1 Representations of graphs 591

an undirected graph, the sum of the lengths of all the adjacency lists is 2 jEj, since

if .u; / is an undirected edge, then u appears in ’s adjacency list and vice versa.

For both directed and undirected graphs, the adjacency-list representation has the

desirable property that the amount of memory it requires is ‚.V C E/.

We can readily adapt adjacency lists to represent weighted graphs, that is, graphs

for which each edge has an associated weight, typically given by a weight function

w W E ! R . For example, let G D .V; E/ be a weighted graph with weight

function w. We simply store the weight w.u; / of the edge .u; / 2 E with

vertex in u’s adjacency list. The adjacency-list representation is quite robust in

that we can modify it to support many other graph variants.

A potential disadvantage of the adjacency-list representation is that it provides

no quicker way to determine whether a given edge .u; / is present in the graph

than to search for in the adjacency list AdjŒu. An adjacency-matrix representa-

tion of the graph remedies this disadvantage, but at the cost of using asymptotically

more memory. (See Exercise 22.1-8 for suggestions of variations on adjacency lists

that permit faster edge lookup.)

For the adjacency-matrix representation of a graph G D .V; E/, we assume

that the vertices are numbered 1; 2; : : : ; jV j in some arbitrary manner. Then the

adjacency-matrix representation of a graph G consists of a jV j jV j matrix

A D .a ij / such that

a ij D

(

1 if .i; j / 2 E ;

0 otherwise :

Figures 22.1(c) and 22.2(c) are the adjacency matrices of the undirected and di-

rected graphs in Figures 22.1(a) and 22.2(a), respectively. The adjacency matrix of

a graph requires ‚.V 2 / memory, independent of the number of edges in the graph.

Observe the symmetry along the main diagonal of the adjacency matrix in Fig-

ure 22.1(c). Since in an undirected graph, .u; / and .; u/ represent the same

edge, the adjacency matrix A of an undirected graph is its own transpose: A D A T .

In some applications, it pays to store only the entries on and above the diagonal of

the adjacency matrix, thereby cutting the memory needed to store the graph almost

in half.

Like the adjacency-list representation of a graph, an adjacency matrix can repre-

sent a weighted graph. For example, if G D .V; E/ is a weighted graph with edge-

weight function w, we can simply store the weight w.u; / of the edge .u; / 2 E

as the entry in row u and column of the adjacency matrix. If an edge does not

exist, we can store a NIL value as its corresponding matrix entry, though for many

problems it is convenient to use a value such as 0 or 1.

Although the adjacency-list representation is asymptotically at least as space-

efﬁcient as the adjacency-matrix representation, adjacency matrices are simpler,

and so we may prefer them when graphs are reasonably small. Moreover, adja-

592 Chapter 22 Elementary Graph Algorithms

cency matrices carry a further advantage for unweighted graphs: they require only

one bit per entry.

Representing attributes

Most algorithms that operate on graphs need to maintain attributes for vertices

and/or edges. We indicate these attributes using our usual notation, such as :d

for an attribute d of a vertex . When we indicate edges as pairs of vertices, we

use the same style of notation. For example, if edges have an attribute f , then we

denote this attribute for edge .u; / by .u; /:f . For the purpose of presenting and

understanding algorithms, our attribute notation sufﬁces.

Implementing vertex and edge attributes in real programs can be another story

entirely. There is no one best way to store and access vertex and edge attributes.

For a given situation, your decision will likely depend on the programming lan-

guage you are using, the algorithm you are implementing, and how the rest of your

program uses the graph. If you represent a graph using adjacency lists, one design

represents vertex attributes in additional arrays, such as an array dŒ1 : : jV j that

parallels the Adj array. If the vertices adjacent to u are in AdjŒu, then what we call

the attribute u:d would actually be stored in the array entry dŒu. Many other ways

of implementing attributes are possible. For example, in an object-oriented pro-

gramming language, vertex attributes might be represented as instance variables

within a subclass of a Vertex class.

Exercises

22.1-1

Given an adjacency-list representation of a directed graph, how long does it take

to compute the out-degree of every vertex? How long does it take to compute the

in-degrees?

22.1-2

Give an adjacency-list representation for a complete binary tree on 7 vertices. Give

an equivalent adjacency-matrix representation. Assume that vertices are numbered

from 1 to 7 as in a binary heap.

22.1-3

The transpose of a directed graph G D .V; E/ is the graph G T D .V; E T /, where

E T D f.; u/ 2 V V W .u; / 2 Eg. Thus, G T is G with all its edges reversed.

Describe efﬁcient algorithms for computing G T from G, for both the adjacency-

list and adjacency-matrix representations of G. Analyze the running times of your

algorithms.

22.1 Representations of graphs 593

22.1-4

Given an adjacency-list representation of a multigraph G D .V; E/, describe an

O.V C E/-time algorithm to compute the adjacency-list representation of the

“equivalent” undirected graph G 0 D .V; E 0 /, where E 0

consists of the edges in E

with all multiple edges between two vertices replaced by a single edge and with all

self-loops removed.

22.1-5

The square of a directed graph G D .V; E/ is the graph G 2 D .V; E 2 / such that

.u; / 2 E 2

if and only G contains a path with at most two edges between u and .

Describe efﬁcient algorithms for computing G 2

from G for both the adjacency-

list and adjacency-matrix representations of G. Analyze the running times of your

algorithms.

22.1-6

Most graph algorithms that take an adjacency-matrix representation as input re-

quire time .V 2 /, but there are some exceptions. Show how to determine whether

a directed graph G contains a universal sink—a vertex with in-degree jV j 1 and

out-degree 0—in time O.V /, given an adjacency matrix for G.

22.1-7

The incidence matrix of a directed graph G D .V; E/ with no self-loops is a

jV j jEj matrix B D .b ij / such that

b ij D

1 if edge j leaves vertex i ;

1 if edge j enters vertex i ;

0 otherwise :

Describe what the entries of the matrix product BB T represent, where B T is the

transpose of B.

22.1-8

Suppose that instead of a linked list, each array entry AdjŒu is a hash table contain-

ing the vertices for which .u; / 2 E. If all edge lookups are equally likely, what

is the expected time to determine whether an edge is in the graph? What disadvan-

tages does this scheme have? Suggest an alternate data structure for each edge list

that solves these problems. Does your alternative have disadvantages compared to

the hash table?

594 Chapter 22 Elementary Graph Algorithms

22.2 Breadth-ﬁrst search

Breadth-ﬁrst search is one of the simplest algorithms for searching a graph and

the archetype for many important graph algorithms. Prim’s minimum-spanning-

tree algorithm (Section 23.2) and Dijkstra’s single-source shortest-paths algorithm

(Section 24.3) use ideas similar to those in breadth-ﬁrst search.

Given a graph G D .V; E/ and a distinguished source vertex s, breadth-ﬁrst

search systematically explores the edges of G to “discover” every vertex that is

reachable from s. It computes the distance (smallest number of edges) from s

to each reachable vertex. It also produces a “breadth-ﬁrst tree” with root s that

contains all reachable vertices. For any vertex reachable from s, the simple path

in the breadth-ﬁrst tree from s to corresponds to a “shortest path” from s to

in G, that is, a path containing the smallest number of edges. The algorithm works

on both directed and undirected graphs.

Breadth-ﬁrst search is so named because it expands the frontier between discov-

ered and undiscovered vertices uniformly across the breadth of the frontier. That

is, the algorithm discovers all vertices at distance k from s before discovering any

vertices at distance k C 1.

To keep track of progress, breadth-ﬁrst search colors each vertex white, gray, or

black. All vertices start out white and may later become gray and then black. A

vertex is discovered the ﬁrst time it is encountered during the search, at which time

it becomes nonwhite. Gray and black vertices, therefore, have been discovered, but

breadth-ﬁrst search distinguishes between them to ensure that the search proceeds

in a breadth-ﬁrst manner. 1 If .u; / 2 E and vertex u is black, then vertex

is either gray or black; that is, all vertices adjacent to black vertices have been

discovered. Gray vertices may have some adjacent white vertices; they represent

the frontier between discovered and undiscovered vertices.

Breadth-ﬁrst search constructs a breadth-ﬁrst tree, initially containing only its

root, which is the source vertex s. Whenever the search discovers a white vertex

in the course of scanning the adjacency list of an already discovered vertex u, the

vertex and the edge .u; / are added to the tree. We say that u is the predecessor

or parent of in the breadth-ﬁrst tree. Since a vertex is discovered at most once, it

has at most one parent. Ancestor and descendant relationships in the breadth-ﬁrst

tree are deﬁned relative to the root s as usual: if u is on the simple path in the tree

from the root s to vertex , then u is an ancestor of and is a descendant of u.

1

We distinguish between gray and black vertices to help us understand how breadth-ﬁrst search op-

erates. In fact, as Exercise 22.2-3 shows, we would get the same result even if we did not distinguish

between gray and black vertices.

22.2 Breadth-ﬁrst search 595

The breadth-ﬁrst-search procedure BFS below assumes that the input graph

G D .V; E/ is represented using adjacency lists. It attaches several additional

attributes to each vertex in the graph. We store the color of each vertex u 2 V

in the attribute u:color and the predecessor of u in the attribute u:. If u has no

predecessor (for example, if u D s or u has not been discovered), then u: D NIL.

The attribute u:d holds the distance from the source s to vertex u computed by the

algorithm. The algorithm also uses a ﬁrst-in, ﬁrst-out queue Q (see Section 10.1)

to manage the set of gray vertices.

BFS.G; s/

1 for each vertex u 2 G:V fsg

2 u:color D WHITE

3 u:d D 1

4 u: D NIL

5 s:color D GRAY

6 s:d D 0

7 s: D NIL

8 Q D ;

9 ENQUEUE.Q; s/

10 while Q ¤ ;

11 u D DEQUEUE.Q/

12 for each 2 G:AdjŒu

13 if :color == WHITE

14 :color D GRAY

15 :d D u:d C 1

16 : D u

17 ENQUEUE.Q; /

18 u:color D BLACK

Figure 22.3 illustrates the progress of BFS on a sample graph.

The procedure BFS works as follows. With the exception of the source vertex s,

lines 1–4 paint every vertex white, set u:d to be inﬁnity for each vertex u, and set

the parent of every vertex to be NIL. Line 5 paints s gray, since we consider it to be

discovered as the procedure begins. Line 6 initializes s:d to 0, and line 7 sets the

predecessor of the source to be NIL. Lines 8–9 initialize Q to the queue containing

just the vertex s.

The while loop of lines 10–18 iterates as long as there remain gray vertices,

which are discovered vertices that have not yet had their adjacency lists fully ex-

amined. This while loop maintains the following invariant:

At the test in line 10, the queue Q consists of the set of gray vertices.

596 Chapter 22 Elementary Graph Algorithms

r s t u

v w x y

0 ∞ ∞ ∞

∞ ∞ ∞ ∞

s

0

Q (a)

t u

v w x y

0 1 ∞ ∞

∞ ∞ ∞ 1

w

1

Q (b) r

1

t u

v w x y

0 1 2 ∞

∞ 2 ∞ 1

Q (c) r

1

t u

v w x y

0 1 ∞

∞

Q (d)

(e) (f)

(g) (h)

Q (i)

r s

r s r s

t

2

x

2

2

2 1 2

t

2

x

2

v

2

t u

v w x y

0 1

∞

Q

r s

2

2 1 2

x

2

v

2

u

3

3

t u

v w x y

0 1

3

Q

r s

2

2 1 2

v

2

u

3

3

y

3

t u

v w x y

0 1

3

Q

r s

2

2 1

u

3

3

y

3 2

t u

v w x y

0 1

3

Q

r s

2

2 1

3

y

3 2

t u

v w x y

0 1

r s

2

2 1

3

2 3

;

Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they

are produced by BFS. The value of u:d appears within each vertex u. The queue Q is shown at the

beginning of each iteration of the while loop of lines 10–18. Vertex distances appear below vertices

in the queue.

Although we won’t use this loop invariant to prove correctness, it is easy to see

that it holds prior to the ﬁrst iteration and that each iteration of the loop maintains

the invariant. Prior to the ﬁrst iteration, the only gray vertex, and the only vertex

in Q, is the source vertex s. Line 11 determines the gray vertex u at the head of

the queue Q and removes it from Q. The for loop of lines 12–17 considers each

vertex in the adjacency list of u. If is white, then it has not yet been discovered,

and the procedure discovers it by executing lines 14–17. The procedure paints

vertex gray, sets its distance :d to u:dC1, records u as its parent :, and places

it at the tail of the queue Q. Once the procedure has examined all the vertices on u’s

22.2 Breadth-ﬁrst search 597

adjacency list, it blackens u in line 18. The loop invariant is maintained because

whenever a vertex is painted gray (in line 14) it is also enqueued (in line 17), and

whenever a vertex is dequeued (in line 11) it is also painted black (in line 18).

The results of breadth-ﬁrst search may depend upon the order in which the neigh-

bors of a given vertex are visited in line 12: the breadth-ﬁrst tree may vary, but the

distances d computed by the algorithm will not. (See Exercise 22.2-5.)

Analysis

Before proving the various properties of breadth-ﬁrst search, we take on the some-

what easier job of analyzing its running time on an input graph G D .V; E/. We

use aggregate analysis, as we saw in Section 17.1. After initialization, breadth-ﬁrst

search never whitens a vertex, and thus the test in line 13 ensures that each vertex

is enqueued at most once, and hence dequeued at most once. The operations of

enqueuing and dequeuing take O.1/ time, and so the total time devoted to queue

operations is O.V /. Because the procedure scans the adjacency list of each vertex

only when the vertex is dequeued, it scans each adjacency list at most once. Since

the sum of the lengths of all the adjacency lists is ‚.E/, the total time spent in

scanning adjacency lists is O.E/. The overhead for initialization is O.V /, and

thus the total running time of the BFS procedure is O.V C E/. Thus, breadth-ﬁrst

search runs in time linear in the size of the adjacency-list representation of G.

Shortest paths

At the beginning of this section, we claimed that breadth-ﬁrst search ﬁnds the dis-

tance to each reachable vertex in a graph G D .V; E/ from a given source vertex

s 2 V . Deﬁne the shortest-path distance ı.s; / from s to as the minimum num-

ber of edges in any path from vertex s to vertex ; if there is no path from s to ,

then ı.s; / D 1. We call a path of length ı.s; / from s to a shortest path 2

from s to . Before showing that breadth-ﬁrst search correctly computes shortest-

path distances, we investigate an important property of shortest-path distances.

2

In Chapters 24 and 25, we shall generalize our study of shortest paths to weighted graphs, in which

every edge has a real-valued weight and the weight of a path is the sum of the weights of its con-

stituent edges. The graphs considered in the present chapter are unweighted or, equivalently, all

edges have unit weight.

598 Chapter 22 Elementary Graph Algorithms

Lemma 22.1

Let G D .V; E/ be a directed or undirected graph, and let s 2 V be an arbitrary

vertex. Then, for any edge .u; / 2 E,

ı.s; / ı.s; u/ C 1 :

Proof If u is reachable from s, then so is . In this case, the shortest path from s

to cannot be longer than the shortest path from s to u followed by the edge .u; /,

and thus the inequality holds. If u is not reachable from s, then ı.s; u/ D 1, and

the inequality holds.

We want to show that BFS properly computes :d D ı.s; / for each ver-

tex 2 V . We ﬁrst show that :d bounds ı.s; / from above.

Lemma 22.2

Let G D .V; E/ be a directed or undirected graph, and suppose that BFS is run

on G from a given source vertex s 2 V . Then upon termination, for each ver-

tex 2 V , the value :d computed by BFS satisﬁes :d ı.s; /.

Proof We use induction on the number of ENQUEUE operations. Our inductive

hypothesis is that :d ı.s; / for all 2 V .

The basis of the induction is the situation immediately after enqueuing s in line 9

of BFS. The inductive hypothesis holds here, because s:d D 0 D ı.s; s/ and

:d D 1 ı.s; / for all 2 V fsg.

For the inductive step, consider a white vertex that is discovered during the

search from a vertex u. The inductive hypothesis implies that u:d ı.s; u/. From

the assignment performed by line 15 and from Lemma 22.1, we obtain

:d D u:d C 1

ı.s; u/ C 1

ı.s; / :

Vertex is then enqueued, and it is never enqueued again because it is also grayed

and the then clause of lines 14–17 is executed only for white vertices. Thus, the

value of :d never changes again, and the inductive hypothesis is maintained.

To prove that :d D ı.s; /, we must ﬁrst show more precisely how the queue Q

operates during the course of BFS. The next lemma shows that at all times, the

queue holds at most two distinct d values.

22.2 Breadth-ﬁrst search 599

Lemma 22.3

Suppose that during the execution of BFS on a graph G D .V; E/, the queue Q

contains the vertices h 1 ; 2 ; : : : ; r i, where 1 is the head of Q and r is the tail.

Then, r :d 1 :d C 1 and i :d iC1 :d for i D 1; 2; : : : ; r 1.

Proof The proof is by induction on the number of queue operations. Initially,

when the queue contains only s, the lemma certainly holds.

For the inductive step, we must prove that the lemma holds after both dequeuing

and enqueuing a vertex. If the head 1 of the queue is dequeued, 2 becomes the

new head. (If the queue becomes empty, then the lemma holds vacuously.) By the

inductive hypothesis, 1 :d 2 :d. But then we have r :d 1 :d C 1 2 :d C 1,

and the remaining inequalities are unaffected. Thus, the lemma follows with 2 as

the head.

In order to understand what happens upon enqueuing a vertex, we need to ex-

amine the code more closely. When we enqueue a vertex in line 17 of BFS, it

becomes rC1 . At that time, we have already removed vertex u, whose adjacency

list is currently being scanned, from the queue Q, and by the inductive hypothesis,

the new head 1 has 1 :d u:d. Thus, rC1 :d D :d D u:dC1 1 :dC1. From

the inductive hypothesis, we also have r :d u:d C 1, and so r :d u:d C 1 D

:d D rC1 :d, and the remaining inequalities are unaffected. Thus, the lemma

follows when is enqueued.

The following corollary shows that the d values at the time that vertices are

enqueued are monotonically increasing over time.

Corollary 22.4

Suppose that vertices i and j are enqueued during the execution of BFS, and

that i is enqueued before j . Then i :d j :d at the time that j is enqueued.

Proof Immediate from Lemma 22.3 and the property that each vertex receives a

ﬁnite d value at most once during the course of BFS.

We can now prove that breadth-ﬁrst search correctly ﬁnds shortest-path dis-

tances.

Theorem 22.5 (Correctness of breadth-ﬁrst search)

Let G D .V; E/ be a directed or undirected graph, and suppose that BFS is run

on G from a given source vertex s 2 V . Then, during its execution, BFS discovers

every vertex 2 V that is reachable from the source s, and upon termination,

:d D ı.s; / for all 2 V . Moreover, for any vertex ¤ s that is reachable

600 Chapter 22 Elementary Graph Algorithms

from s, one of the shortest paths from s to is a shortest path from s to :

followed by the edge .:; /.

Proof Assume, for the purpose of contradiction, that some vertex receives a d

value not equal to its shortest-path distance. Let be the vertex with min-

imum ı.s; / that receives such an incorrect d value; clearly ¤ s. By

Lemma 22.2, :d ı.s; /, and thus we have that :d > ı.s; /. Vertex must be

reachable from s, for if it is not, then ı.s; / D 1 :d. Let u be the vertex im-

mediately preceding on a shortest path from s to , so that ı.s; / D ı.s; u/ C 1.

Because ı.s; u/ < ı.s; /, and because of how we chose , we have u:d D ı.s; u/.

Putting these properties together, we have

:d > ı.s; / D ı.s; u/ C 1 D u:d C 1 : (22.1)

Now consider the time when BFS chooses to dequeue vertex u from Q in

line 11. At this time, vertex is either white, gray, or black. We shall show

that in each of these cases, we derive a contradiction to inequality (22.1). If is

white, then line 15 sets :d D u:d C 1, contradicting inequality (22.1). If is

black, then it was already removed from the queue and, by Corollary 22.4, we have

:d u:d, again contradicting inequality (22.1). If is gray, then it was painted

gray upon dequeuing some vertex w, which was removed from Q earlier than u

and for which :d D w:d C 1. By Corollary 22.4, however, w:d u:d, and so we

have :d D w:d C 1 u:d C 1, once again contradicting inequality (22.1).

Thus we conclude that :d D ı.s; / for all 2 V . All vertices reachable

from s must be discovered, for otherwise they would have 1 D :d > ı.s; /. To

conclude the proof of the theorem, observe that if : D u, then :d D u:d C 1.

Thus, we can obtain a shortest path from s to by taking a shortest path from s

to : and then traversing the edge .:; /.

Breadth-ﬁrst trees

The procedure BFS builds a breadth-ﬁrst tree as it searches the graph, as Fig-

ure 22.3 illustrates. The tree corresponds to the attributes. More formally, for

a graph G D .V; E/ with source s, we deﬁne the predecessor subgraph of G as

G D .V ; E /, where

V D f 2 V W : ¤ NILg [ fsg

and

E D f.:; / W 2 V fsgg :

The predecessor subgraph G is a breadth-ﬁrst tree if V consists of the vertices

reachable from s and, for all 2 V , the subgraph G contains a unique simple

22.2 Breadth-ﬁrst search 601

path from s to that is also a shortest path from s to in G. A breadth-ﬁrst tree

is in fact a tree, since it is connected and jE j D jV j 1 (see Theorem B.2). We

call the edges in E tree edges.

The following lemma shows that the predecessor subgraph produced by the BFS

procedure is a breadth-ﬁrst tree.

Lemma 22.6

When applied to a directed or undirected graph G D .V; E/, procedure BFS con-

structs so that the predecessor subgraph G D .V ; E / is a breadth-ﬁrst tree.

Proof Line 16 of BFS sets : D u if and only if .u; / 2 E and ı.s; / < 1—

that is, if is reachable from s—and thus V consists of the vertices in V reachable

from s. Since G forms a tree, by Theorem B.2, it contains a unique simple path

from s to each vertex in V . By applying Theorem 22.5 inductively, we conclude

that every such path is a shortest path in G.

The following procedure prints out the vertices on a shortest path from s to ,

assuming that BFS has already computed a breadth-ﬁrst tree:

PRINT-PATH.G; s; /

1 if == s

2 print s

3 elseif : == NIL

4 print “no path from” s “to” “exists”

5 else PRINT-PATH.G; s; :/

6 print

This procedure runs in time linear in the number of vertices in the path printed,

since each recursive call is for a path one vertex shorter.

Exercises

22.2-1

Show the d and values that result from running breadth-ﬁrst search on the di-

rected graph of Figure 22.2(a), using vertex 3 as the source.

22.2-2

Show the d and values that result from running breadth-ﬁrst search on the undi-

rected graph of Figure 22.3, using vertex u as the source.

602 Chapter 22 Elementary Graph Algorithms

22.2-3

Show that using a single bit to store each vertex color sufﬁces by arguing that the

BFS procedure would produce the same result if lines 5 and 14 were removed.

22.2-4

What is the running time of BFS if we represent its input graph by an adjacency

matrix and modify the algorithm to handle this form of input?

22.2-5

Argue that in a breadth-ﬁrst search, the value u:d assigned to a vertex u is inde-

pendent of the order in which the vertices appear in each adjacency list. Using

Figure 22.3 as an example, show that the breadth-ﬁrst tree computed by BFS can

depend on the ordering within adjacency lists.

22.2-6

Give an example of a directed graph G D .V; E/, a source vertex s 2 V , and a

set of tree edges E E such that for each vertex 2 V , the unique simple path

in the graph .V; E / from s to is a shortest path in G, yet the set of edges E

cannot be produced by running BFS on G, no matter how the vertices are ordered

in each adjacency list.

22.2-7

There are two types of professional wrestlers: “babyfaces” (“good guys”) and

“heels” (“bad guys”). Between any pair of professional wrestlers, there may or

may not be a rivalry. Suppose we have n professional wrestlers and we have a list

of r pairs of wrestlers for which there are rivalries. Give an O.n C r/-time algo-

rithm that determines whether it is possible to designate some of the wrestlers as

babyfaces and the remainder as heels such that each rivalry is between a babyface

and a heel. If it is possible to perform such a designation, your algorithm should

produce it.

22.2-8 ?

The diameter of a tree T D .V; E/ is deﬁned as max u;2V ı.u; /, that is, the

largest of all shortest-path distances in the tree. Give an efﬁcient algorithm to

compute the diameter of a tree, and analyze the running time of your algorithm.

22.2-9

Let G D .V; E/ be a connected, undirected graph. Give an O.V C E/-time algo-

rithm to compute a path in G that traverses each edge in E exactly once in each

direction. Describe how you can ﬁnd your way out of a maze if you are given a

large supply of pennies.

22.3 Depth-ﬁrst search 603

22.3 Depth-ﬁrst search

The strategy followed by depth-ﬁrst search is, as its name implies, to search

“deeper” in the graph whenever possible. Depth-ﬁrst search explores edges out

of the most recently discovered vertex that still has unexplored edges leaving it.

Once all of ’s edges have been explored, the search “backtracks” to explore edges

leaving the vertex from which was discovered. This process continues until we

have discovered all the vertices that are reachable from the original source vertex.

If any undiscovered vertices remain, then depth-ﬁrst search selects one of them as

a new source, and it repeats the search from that source. The algorithm repeats this

entire process until it has discovered every vertex. 3

As in breadth-ﬁrst search, whenever depth-ﬁrst search discovers a vertex dur-

ing a scan of the adjacency list of an already discovered vertex u, it records this

event by setting ’s predecessor attribute : to u. Unlike breadth-ﬁrst search,

whose predecessor subgraph forms a tree, the predecessor subgraph produced by

a depth-ﬁrst search may be composed of several trees, because the search may

repeat from multiple sources. Therefore, we deﬁne the predecessor subgraph of

a depth-ﬁrst search slightly differently from that of a breadth-ﬁrst search: we let

G D .V; E /, where

E D f.:; / W 2 V and : ¤ NILg :

The predecessor subgraph of a depth-ﬁrst search forms a depth-ﬁrst forest com-

prising several depth-ﬁrst trees. The edges in E are tree edges.

As in breadth-ﬁrst search, depth-ﬁrst search colors vertices during the search to

indicate their state. Each vertex is initially white, is grayed when it is discovered

in the search, and is blackened when it is ﬁnished, that is, when its adjacency list

has been examined completely. This technique guarantees that each vertex ends up

in exactly one depth-ﬁrst tree, so that these trees are disjoint.

Besides creating a depth-ﬁrst forest, depth-ﬁrst search also timestamps each ver-

tex. Each vertex has two timestamps: the ﬁrst timestamp :d records when

is ﬁrst discovered (and grayed), and the second timestamp :f records when the

search ﬁnishes examining ’s adjacency list (and blackens ). These timestamps

3

It may seem arbitrary that breadth-ﬁrst search is limited to only one source whereas depth-ﬁrst

search may search from multiple sources. Although conceptually, breadth-ﬁrst search could proceed

from multiple sources and depth-ﬁrst search could be limited to one source, our approach reﬂects how

the results of these searches are typically used. Breadth-ﬁrst search usually serves to ﬁnd shortest-

path distances (and the associated predecessor subgraph) from a given source. Depth-ﬁrst search is

often a subroutine in another algorithm, as we shall see later in this chapter.

604 Chapter 22 Elementary Graph Algorithms

provide important information about the structure of the graph and are generally

helpful in reasoning about the behavior of depth-ﬁrst search.

The procedure DFS below records when it discovers vertex u in the attribute u:d

and when it ﬁnishes vertex u in the attribute u:f . These timestamps are integers

between 1 and 2 jV j, since there is one discovery event and one ﬁnishing event for

each of the jV j vertices. For every vertex u,

u:d < u:f : (22.2)

Vertex u is WHITE before time u:d, GRAY between time u:d and time u:f , and

BLACK thereafter.

The following pseudocode is the basic depth-ﬁrst-search algorithm. The input

graph G may be undirected or directed. The variable time is a global variable that

we use for timestamping.

DFS.G/

1 for each vertex u 2 G:V

2 u:color D WHITE

3 u: D NIL

4 time D 0

5 for each vertex u 2 G:V

6 if u:color == WHITE

7 DFS-VISIT.G; u/

DFS-VISIT.G; u/

1 time D time C 1 // white vertex u has just been discovered

2 u:d D time

3 u:color D GRAY

4 for each 2 G:AdjŒu // explore edge .u; /

5 if :color == WHITE

6 : D u

7 DFS-VISIT.G; /

8 u:color D BLACK // blacken u; it is ﬁnished

9 time D time C 1

10 u:f D time

Figure 22.4 illustrates the progress of DFS on the graph shown in Figure 22.2.

Procedure DFS works as follows. Lines 1–3 paint all vertices white and ini-

tialize their attributes to NIL. Line 4 resets the global time counter. Lines 5–7

check each vertex in V in turn and, when a white vertex is found, visit it using

DFS-VISIT. Every time DFS-VISIT.G; u/ is called in line 7, vertex u becomes

22.3 Depth-ﬁrst search 605

u v w

x y z

1/ 1/ 2/ 1/ 2/

3/

1/ 2/

3/ 4/

1/ 2/

3/ 4/

B

1/ 2/

3/

B

4/5

1/ 2/

B

4/5 3/6

1/

B

4/5 3/6

2/7

1/

B

4/5 3/6

2/7

F B

4/5 3/6

2/7

F

1/8

B

4/5 3/6

2/7

F

1/8 9/

B

4/5 3/6

2/7

F

1/8 9/

C

B

4/5 3/6

2/7

F

1/8 9/

C B

4/5 3/6

2/7

F

1/8 9/

C

B

B

4/5 3/6

2/7

F

1/8 9/

C

B

10/11

B

4/5 3/6

2/7

F

1/8

C

B

10/11

9/12

u v w

x y z

u v w

x y z

u v w

x y z

u v w

x y z

u v w

x y z

u v w

x y z

u v w

x y z

u v w

x y z

u v w

x y z

u v w

x y z

u v w

x y z

u v w

x y z

u v w

x y z

u v w

x y z

u v w

x y z

(m) (n) (o) (p)

(i) (j) (k) (l)

(e) (f) (g) (h)

(a) (b) (c) (d)

10/ 10/

Figure 22.4 The progress of the depth-ﬁrst-search algorithm DFS on a directed graph. As edges

are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed

(otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or

forward edges. Timestamps within vertices indicate discovery time/ﬁnishing times.

the root of a new tree in the depth-ﬁrst forest. When DFS returns, every vertex u

has been assigned a discovery time u:d and a ﬁnishing time u:f .

In each call DFS-VISIT.G; u/, vertex u is initially white. Line 1 increments

the global variable time, line 2 records the new value of time as the discovery

time u:d, and line 3 paints u gray. Lines 4–7 examine each vertex adjacent to u

and recursively visit if it is white. As each vertex 2 AdjŒu is considered in

line 4, we say that edge .u; / is explored by the depth-ﬁrst search. Finally, after

every edge leaving u has been explored, lines 8–10 paint u black, increment time,

and record the ﬁnishing time in u:f .

Note that the results of depth-ﬁrst search may depend upon the order in which

line 5 of DFS examines the vertices and upon the order in which line 4 of DFS-

VISIT visits the neighbors of a vertex. These different visitation orders tend not

606 Chapter 22 Elementary Graph Algorithms

to cause problems in practice, as we can usually use any depth-ﬁrst search result

effectively, with essentially equivalent results.

What is the running time of DFS? The loops on lines 1–3 and lines 5–7 of DFS

take time ‚.V /, exclusive of the time to execute the calls to DFS-VISIT. As we did

for breadth-ﬁrst search, we use aggregate analysis. The procedure DFS-VISIT is

called exactly once for each vertex 2 V , since the vertex u on which DFS-VISIT

is invoked must be white and the ﬁrst thing DFS-VISIT does is paint vertex u gray.

During an execution of DFS-VISIT.G; /, the loop on lines 4–7 executes jAdjŒj

times. Since

X

2V

jAdjŒj D ‚.E/ ;

the total cost of executing lines 4–7 of DFS-VISIT is ‚.E/. The running time of

DFS is therefore ‚.V C E/.

Properties of depth-ﬁrst search

Depth-ﬁrst search yields valuable information about the structure of a graph. Per-

haps the most basic property of depth-ﬁrst search is that the predecessor sub-

graph G does indeed form a forest of trees, since the structure of the depth-

ﬁrst trees exactly mirrors the structure of recursive calls of DFS-VISIT. That is,

u D : if and only if DFS-VISIT.G; / was called during a search of u’s ad-

jacency list. Additionally, vertex is a descendant of vertex u in the depth-ﬁrst

forest if and only if is discovered during the time in which u is gray.

Another important property of depth-ﬁrst search is that discovery and ﬁnishing

times have parenthesis structure. If we represent the discovery of vertex u with

a left parenthesis “.u” and represent its ﬁnishing by a right parenthesis “u/”, then

the history of discoveries and ﬁnishings makes a well-formed expression in the

sense that the parentheses are properly nested. For example, the depth-ﬁrst search

of Figure 22.5(a) corresponds to the parenthesization shown in Figure 22.5(b). The

following theorem provides another way to characterize the parenthesis structure.

Theorem 22.7 (Parenthesis theorem)

In any depth-ﬁrst search of a (directed or undirected) graph G D .V; E/, for any

two vertices u and , exactly one of the following three conditions holds:

the intervals Œu:d; u:f and Œ:d; :f are entirely disjoint, and neither u nor

is a descendant of the other in the depth-ﬁrst forest,

the interval Œu:d; u:f is contained entirely within the interval Œ:d; :f , and u

is a descendant of in a depth-ﬁrst tree, or

the interval Œ:d; :f is contained entirely within the interval Œu:d; u:f , and

is a descendant of u in a depth-ﬁrst tree.

22.3 Depth-ﬁrst search 607

3/6 2/9 1/10 11/16

14/15 12/13 7/8 4/5

y z s t

u v w x

B

C

F

C

C

C

B

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

s t

z

y w

x

v u

s

z

y w

x

t

v u

C

F

B

C

C

B

C

(a)

(b)

(c)

(s (z (y (x x) y) (w w) z) s) (t (v v) (u u) t)

Figure 22.5 Properties of depth-ﬁrst search. (a) The result of a depth-ﬁrst search of a directed

graph. Vertices are timestamped and edge types are indicated as in Figure 22.4. (b) Intervals for

the discovery time and ﬁnishing time of each vertex correspond to the parenthesization shown. Each

rectangle spans the interval given by the discovery and ﬁnishing times of the corresponding vertex.

Only tree edges are shown. If two intervals overlap, then one is nested within the other, and the

vertex corresponding to the smaller interval is a descendant of the vertex corresponding to the larger.

(c) The graph of part (a) redrawn with all tree and forward edges going down within a depth-ﬁrst tree

and all back edges going up from a descendant to an ancestor.

608 Chapter 22 Elementary Graph Algorithms

Proof We begin with the case in which u:d < :d. We consider two subcases,

according to whether :d < u:f or not. The ﬁrst subcase occurs when :d < u:f ,

so was discovered while u was still gray, which implies that is a descendant

of u. Moreover, since was discovered more recently than u, all of its outgo-

ing edges are explored, and is ﬁnished, before the search returns to and ﬁn-

ishes u. In this case, therefore, the interval Œ:d; :f is entirely contained within

the interval Œu:d; u:f . In the other subcase, u:f < :d, and by inequality (22.2),

u:d < u:f < :d < :f ; thus the intervals Œu:d; u:f and Œ:d; :f are disjoint.

Because the intervals are disjoint, neither vertex was discovered while the other

was gray, and so neither vertex is a descendant of the other.

The case in which :d < u:d is similar, with the roles of u and reversed in the

above argument.

Corollary 22.8 (Nesting of descendants’ intervals)

Vertex is a proper descendant of vertex u in the depth-ﬁrst forest for a (directed

or undirected) graph G if and only if u:d < :d < :f < u:f .

Proof Immediate from Theorem 22.7.

The next theorem gives another important characterization of when one vertex

is a descendant of another in the depth-ﬁrst forest.

Theorem 22.9 (White-path theorem)

In a depth-ﬁrst forest of a (directed or undirected) graph G D .V; E/, vertex is

a descendant of vertex u if and only if at the time u:d that the search discovers u,

there is a path from u to consisting entirely of white vertices.

Proof ): If D u, then the path from u to contains just vertex u, which is still

white when we set the value of u:d. Now, suppose that is a proper descendant

of u in the depth-ﬁrst forest. By Corollary 22.8, u:d < :d, and so is white at

time u:d. Since can be any descendant of u, all vertices on the unique simple

path from u to in the depth-ﬁrst forest are white at time u:d.

(: Suppose that there is a path of white vertices from u to at time u:d, but

does not become a descendant of u in the depth-ﬁrst tree. Without loss of general-

ity, assume that every vertex other than along the path becomes a descendant of u.

(Otherwise, let be the closest vertex to u along the path that doesn’t become a de-

scendant of u.) Let w be the predecessor of in the path, so that w is a descendant

of u (w and u may in fact be the same vertex). By Corollary 22.8, w:f u:f . Be-

cause must be discovered after u is discovered, but before w is ﬁnished, we have

u:d < :d < w:f u:f . Theorem 22.7 then implies that the interval Œ:d; :f

22.3 Depth-ﬁrst search 609

is contained entirely within the interval Œu:d; u:f . By Corollary 22.8, must after

all be a descendant of u.

Classiﬁcation of edges

Another interesting property of depth-ﬁrst search is that the search can be used

to classify the edges of the input graph G D .V; E/. The type of each edge can

provide important information about a graph. For example, in the next section, we

shall see that a directed graph is acyclic if and only if a depth-ﬁrst search yields no

“back” edges (Lemma 22.11).

We can deﬁne four edge types in terms of the depth-ﬁrst forest G produced by

a depth-ﬁrst search on G:

1. Tree edges are edges in the depth-ﬁrst forest G . Edge .u; / is a tree edge if

was ﬁrst discovered by exploring edge .u; /.

2. Back edges are those edges .u; / connecting a vertex u to an ancestor in a

depth-ﬁrst tree. We consider self-loops, which may occur in directed graphs, to

be back edges.

3. Forward edges are those nontree edges .u; / connecting a vertex u to a de-

scendant in a depth-ﬁrst tree.

4. Cross edges are all other edges. They can go between vertices in the same

depth-ﬁrst tree, as long as one vertex is not an ancestor of the other, or they can

go between vertices in different depth-ﬁrst trees.

In Figures 22.4 and 22.5, edge labels indicate edge types. Figure 22.5(c) also shows

how to redraw the graph of Figure 22.5(a) so that all tree and forward edges head

downward in a depth-ﬁrst tree and all back edges go up. We can redraw any graph

in this fashion.

The DFS algorithm has enough information to classify some edges as it encoun-

ters them. The key idea is that when we ﬁrst explore an edge .u; /, the color of

vertex tells us something about the edge:

1. WHITE indicates a tree edge,

2. GRAY indicates a back edge, and

3. BLACK indicates a forward or cross edge.

The ﬁrst case is immediate from the speciﬁcation of the algorithm. For the sec-

ond case, observe that the gray vertices always form a linear chain of descendants

corresponding to the stack of active DFS-VISIT invocations; the number of gray

vertices is one more than the depth in the depth-ﬁrst forest of the vertex most re-

cently discovered. Exploration always proceeds from the deepest gray vertex, so

610 Chapter 22 Elementary Graph Algorithms

an edge that reaches another gray vertex has reached an ancestor. The third case

handles the remaining possibility; Exercise 22.3-5 asks you to show that such an

edge .u; / is a forward edge if u:d < :d and a cross edge if u:d > :d.

An undirected graph may entail some ambiguity in how we classify edges,

since .u; / and .; u/ are really the same edge. In such a case, we classify the

edge as the ﬁrst type in the classiﬁcation list that applies. Equivalently (see Ex-

ercise 22.3-6), we classify the edge according to whichever of .u; / or .; u/ the

search encounters ﬁrst.

We now show that forward and cross edges never occur in a depth-ﬁrst search of

an undirected graph.

Theorem 22.10

In a depth-ﬁrst search of an undirected graph G, every edge of G is either a tree

edge or a back edge.

Proof Let .u; / be an arbitrary edge of G, and suppose without loss of generality

that u:d < :d. Then the search must discover and ﬁnish before it ﬁnishes u

(while u is gray), since is on u’s adjacency list. If the ﬁrst time that the search

explores edge .u; /, it is in the direction from u to , then is undiscovered

(white) until that time, for otherwise the search would have explored this edge

already in the direction from to u. Thus, .u; / becomes a tree edge. If the

search explores .u; / ﬁrst in the direction from to u, then .u; / is a back edge,

since u is still gray at the time the edge is ﬁrst explored.

We shall see several applications of these theorems in the following sections.

Exercises

22.3-1

Make a 3-by-3 chart with row and column labels WHITE, GRAY, and BLACK. In

each cell .i; j /, indicate whether, at any point during a depth-ﬁrst search of a di-

rected graph, there can be an edge from a vertex of color i to a vertex of color j .

For each possible edge, indicate what edge types it can be. Make a second such

chart for depth-ﬁrst search of an undirected graph.

22.3-2

Show how depth-ﬁrst search works on the graph of Figure 22.6. Assume that the

for loop of lines 5–7 of the DFS procedure considers the vertices in alphabetical

order, and assume that each adjacency list is ordered alphabetically. Show the

discovery and ﬁnishing times for each vertex, and show the classiﬁcation of each

edge.

22.3 Depth-ﬁrst search 611

q r

s t u

v w x y

z

Figure 22.6 A directed graph for use in Exercises 22.3-2 and 22.5-2.

22.3-3

Show the parenthesis structure of the depth-ﬁrst search of Figure 22.4.

22.3-4

Show that using a single bit to store each vertex color sufﬁces by arguing that

the DFS procedure would produce the same result if line 3 of DFS-VISIT was

removed.

22.3-5

Show that edge .u; / is

a. a tree edge or forward edge if and only if u:d < :d < :f < u:f ,

b. a back edge if and only if :d u:d < u:f :f , and

c. a cross edge if and only if :d < :f < u:d < u:f .

22.3-6

Show that in an undirected graph, classifying an edge .u; / as a tree edge or a back

edge according to whether .u; / or .; u/ is encountered ﬁrst during the depth-ﬁrst

search is equivalent to classifying it according to the ordering of the four types in

the classiﬁcation scheme.

22.3-7

Rewrite the procedure DFS, using a stack to eliminate recursion.

22.3-8

Give a counterexample to the conjecture that if a directed graph G contains a path

from u to , and if u:d < :d in a depth-ﬁrst search of G, then is a descendant

of u in the depth-ﬁrst forest produced.

612 Chapter 22 Elementary Graph Algorithms

22.3-9

Give a counterexample to the conjecture that if a directed graph G contains a path

from u to , then any depth-ﬁrst search must result in :d u:f .

22.3-10

Modify the pseudocode for depth-ﬁrst search so that it prints out every edge in the

directed graph G, together with its type. Show what modiﬁcations, if any, you need

to make if G is undirected.

22.3-11

Explain how a vertex u of a directed graph can end up in a depth-ﬁrst tree contain-

ing only u, even though u has both incoming and outgoing edges in G.

22.3-12

Show that we can use a depth-ﬁrst search of an undirected graph G to identify the

connected components of G, and that the depth-ﬁrst forest contains as many trees

as G has connected components. More precisely, show how to modify depth-ﬁrst

search so that it assigns to each vertex an integer label :cc between 1 and k,

where k is the number of connected components of G, such that u:cc D :cc if

and only if u and are in the same connected component.

22.3-13 ?

A directed graph G D .V; E/ is singly connected if u implies that G contains

at most one simple path from u to for all vertices u; 2 V . Give an efﬁcient

algorithm to determine whether or not a directed graph is singly connected.

22.4 Topological sort

This section shows how we can use depth-ﬁrst search to perform a topological sort

of a directed acyclic graph, or a “dag” as it is sometimes called. A topological sort

of a dag G D .V; E/ is a linear ordering of all its vertices such that if G contains an

edge .u; /, then u appears before in the ordering. (If the graph contains a cycle,

then no linear ordering is possible.) We can view a topological sort of a graph as

an ordering of its vertices along a horizontal line so that all directed edges go from

left to right. Topological sorting is thus different from the usual kind of “sorting”

studied in Part II.

Many applications use directed acyclic graphs to indicate precedences among

events. Figure 22.7 gives an example that arises when Professor Bumstead gets

dressed in the morning. The professor must don certain garments before others

(e.g., socks before shoes). Other items may be put on in any order (e.g., socks and

22.4 Topological sort 613

11/16

12/15

6/7

1/8

2/5

3/4

17/18

13/14

9/10

17/18 11/16 12/15 13/14 9/10 1/8 6/7 2/5 3/4

(a)

(b)

undershorts

pants

belt

shirt

tie

jacket

socks

shoes

watch

socks undershorts pants shoes watch shirt belt tie jacket

Figure 22.7 (a) Professor Bumstead topologically sorts his clothing when getting dressed. Each

directed edge .u; / means that garment u must be put on before garment . The discovery and

ﬁnishing times from a depth-ﬁrst search are shown next to each vertex. (b) The same graph shown

topologically sorted, with its vertices arranged from left to right in order of decreasing ﬁnishing time.

All directed edges go from left to right.

pants). A directed edge .u; / in the dag of Figure 22.7(a) indicates that garment u

must be donned before garment . A topological sort of this dag therefore gives an

order for getting dressed. Figure 22.7(b) shows the topologically sorted dag as an

ordering of vertices along a horizontal line such that all directed edges go from left

to right.

The following simple algorithm topologically sorts a dag:

TOPOLOGICAL-SORT.G/

1 call DFS.G/ to compute ﬁnishing times :f for each vertex

2 as each vertex is ﬁnished, insert it onto the front of a linked list

3 return the linked list of vertices

Figure 22.7(b) shows how the topologically sorted vertices appear in reverse order

of their ﬁnishing times.

We can perform a topological sort in time ‚.V C E/, since depth-ﬁrst search

takes ‚.V C E/ time and it takes O.1/ time to insert each of the jV j vertices onto

the front of the linked list.

We prove the correctness of this algorithm using the following key lemma char-

acterizing directed acyclic graphs.

614 Chapter 22 Elementary Graph Algorithms

Lemma 22.11

A directed graph G is acyclic if and only if a depth-ﬁrst search of G yields no back

edges.

Proof ): Suppose that a depth-ﬁrst search produces a back edge .u; /. Then

vertex is an ancestor of vertex u in the depth-ﬁrst forest. Thus, G contains a path

from to u, and the back edge .u; / completes a cycle.

(: Suppose that G contains a cycle c. We show that a depth-ﬁrst search of G

yields a back edge. Let be the ﬁrst vertex to be discovered in c, and let .u; / be

the preceding edge in c. At time :d, the vertices of c form a path of white vertices

from to u. By the white-path theorem, vertex u becomes a descendant of in the

depth-ﬁrst forest. Therefore, .u; / is a back edge.

Theorem 22.12

TOPOLOGICAL-SORT produces a topological sort of the directed acyclic graph

provided as its input.

Proof Suppose that DFS is run on a given dag G D .V; E/ to determine ﬁn-

ishing times for its vertices. It sufﬁces to show that for any pair of distinct ver-

tices u; 2 V , if G contains an edge from u to , then :f < u:f . Consider any

edge .u; / explored by DFS.G/. When this edge is explored, cannot be gray,

since then would be an ancestor of u and .u; / would be a back edge, contra-

dicting Lemma 22.11. Therefore, must be either white or black. If is white,

it becomes a descendant of u, and so :f < u:f . If is black, it has already been

ﬁnished, so that :f has already been set. Because we are still exploring from u, we

have yet to assign a timestamp to u:f , and so once we do, we will have :f < u:f

as well. Thus, for any edge .u; / in the dag, we have :f < u:f , proving the

theorem.

Exercises

22.4-1

Show the ordering of vertices produced by TOPOLOGICAL-SORT when it is run on

the dag of Figure 22.8, under the assumption of Exercise 22.3-2.

22.4-2

Give a linear-time algorithm that takes as input a directed acyclic graph G D

.V; E/ and two vertices s and t, and returns the number of simple paths from s

to t in G. For example, the directed acyclic graph of Figure 22.8 contains exactly

four simple paths from vertex p to vertex : po, pory, posry, and psry.

(Your algorithm needs only to count the simple paths, not list them.)

22.5 Strongly connected components 615

z y x

w v u t

s r q

p o n m

Figure 22.8 A dag for topological sorting.

22.4-3

Give an algorithm that determines whether or not a given undirected graph G D

.V; E/ contains a cycle. Your algorithm should run in O.V / time, independent

of jEj.

22.4-4

Prove or disprove: If a directed graph G contains cycles, then TOPOLOGICAL-

SORT.G/ produces a vertex ordering that minimizes the number of “bad” edges

that are inconsistent with the ordering produced.

22.4-5

Another way to perform topological sorting on a directed acyclic graph G D

.V; E/ is to repeatedly ﬁnd a vertex of in-degree 0, output it, and remove it and

all of its outgoing edges from the graph. Explain how to implement this idea so

that it runs in time O.V C E/. What happens to this algorithm if G has cycles?

22.5 Strongly connected components

We now consider a classic application of depth-ﬁrst search: decomposing a di-

rected graph into its strongly connected components. This section shows how to do

so using two depth-ﬁrst searches. Many algorithms that work with directed graphs

begin with such a decomposition. After decomposing the graph into strongly con-

nected components, such algorithms run separately on each one and then combine

the solutions according to the structure of connections among components.

Recall from Appendix B that a strongly connected component of a directed

graph G D .V; E/ is a maximal set of vertices C V such that for every pair

of vertices u and in C , we have both u and u; that is, vertices u and

are reachable from each other. Figure 22.9 shows an example.

616 Chapter 22 Elementary Graph Algorithms

13/14 11/16

12/15 3/4

1/10

2/7

8/9

5/6

a b c d

e f g h

a b c d

e f g h

abe

cd

fg h

(c)

(b)

(a)

Figure 22.9 (a) A directed graph G. Each shaded region is a strongly connected component of G.

Each vertex is labeled with its discovery and ﬁnishing times in a depth-ﬁrst search, and tree edges

are shaded. (b) The graph G

T

, the transpose of G, with the depth-ﬁrst forest computed in line 3

of STRONGLY-CONNECTED-COMPONENTS shown and tree edges shaded. Each strongly connected

component corresponds to one depth-ﬁrst tree. Vertices b, c, g, and h, which are heavily shaded, are

the roots of the depth-ﬁrst trees produced by the depth-ﬁrst search of G

T

. (c) The acyclic component

graph G

SCC

obtained by contracting all edges within each strongly connected component of G so

that only a single vertex remains in each component.

Our algorithm for ﬁnding strongly connected components of a graph G D

.V; E/ uses the transpose of G, which we deﬁned in Exercise 22.1-3 to be the

graph G T D .V; E T /, where E T D f.u; / W .; u/ 2 Eg. That is, E T consists of

the edges of G with their directions reversed. Given an adjacency-list representa-

tion of G, the time to create G T is O.V C E/. It is interesting to observe that G

and G T have exactly the same strongly connected components: u and are reach-

able from each other in G if and only if they are reachable from each other in G T .

Figure 22.9(b) shows the transpose of the graph in Figure 22.9(a), with the strongly

connected components shaded.

22.5 Strongly connected components 617

The following linear-time (i.e., ‚.V CE/-time) algorithm computes the strongly

connected components of a directed graph G D .V; E/ using two depth-ﬁrst

searches, one on G and one on G T .

STRONGLY-CONNECTED-COMPONENTS.G/

1 call DFS.G/ to compute ﬁnishing times u:f for each vertex u

2 compute G T

3 call DFS.G T /, but in the main loop of DFS, consider the vertices

in order of decreasing u:f (as computed in line 1)

4 output the vertices of each tree in the depth-ﬁrst forest formed in line 3 as a

separate strongly connected component

The idea behind this algorithm comes from a key property of the component

graph G SCC D .V SCC ; E SCC /, which we deﬁne as follows. Suppose that G

has strongly connected components C 1 ; C 2 ; : : : ; C k . The vertex set V SCC is

f 1 ; 2 ; : : : ; k g, and it contains a vertex i for each strongly connected compo-

nent C i of G. There is an edge . i ; j / 2 E SCC if G contains a directed edge .x; y/

for some x 2 C i and some y 2 C j . Looked at another way, by contracting all

edges whose incident vertices are within the same strongly connected component

of G, the resulting graph is G SCC . Figure 22.9(c) shows the component graph of

the graph in Figure 22.9(a).

The key property is that the component graph is a dag, which the following

lemma implies.

Lemma 22.13

Let C and C 0

be distinct strongly connected components in directed graph G D

.V; E/, let u; 2 C , let u 0 ; 0 2 C 0

, and suppose that G contains a path u u 0

.

Then G cannot also contain a path 0 .

Proof If G contains a path 0 , then it contains paths u u 0 0

and

0 u. Thus, u and 0

are reachable from each other, thereby contradicting

the assumption that C and C 0

are distinct strongly connected components.

We shall see that by considering vertices in the second depth-ﬁrst search in de-

creasing order of the ﬁnishing times that were computed in the ﬁrst depth-ﬁrst

search, we are, in essence, visiting the vertices of the component graph (each of

which corresponds to a strongly connected component of G) in topologically sorted

order.

Because the STRONGLY-CONNECTED-COMPONENTS procedure performs two

depth-ﬁrst searches, there is the potential for ambiguity when we discuss u:d

or u:f . In this section, these values always refer to the discovery and ﬁnishing

times as computed by the ﬁrst call of DFS, in line 1.

618 Chapter 22 Elementary Graph Algorithms

We extend the notation for discovery and ﬁnishing times to sets of vertices.

If U V , then we deﬁne d.U / D min u2U fu:dg and f .U / D max u2U fu:f g.

That is, d.U / and f .U / are the earliest discovery time and latest ﬁnishing time,

respectively, of any vertex in U .

The following lemma and its corollary give a key property relating strongly con-

nected components and ﬁnishing times in the ﬁrst depth-ﬁrst search.

Lemma 22.14

Let C and C 0

be distinct strongly connected components in directed graph G D

.V; E/. Suppose that there is an edge .u; / 2 E, where u 2 C and 2 C 0

. Then

f .C / > f .C 0 /.

Proof We consider two cases, depending on which strongly connected compo-

nent, C or C 0

, had the ﬁrst discovered vertex during the depth-ﬁrst search.

If d.C / < d.C 0 /, let x be the ﬁrst vertex discovered in C . At time x:d, all ver-

tices in C and C 0

are white. At that time, G contains a path from x to each vertex

in C consisting only of white vertices. Because .u; / 2 E, for any vertex w 2 C 0

,

there is also a path in G at time x:d from x to w consisting only of white vertices:

x u ! w. By the white-path theorem, all vertices in C and C 0

become

descendants of x in the depth-ﬁrst tree. By Corollary 22.8, x has the latest ﬁnishing

time of any of its descendants, and so x:f D f .C / > f .C 0 /.

If instead we have d.C / > d.C 0 /, let y be the ﬁrst vertex discovered in C 0

.

At time y:d, all vertices in C 0

are white and G contains a path from y to each

vertex in C 0

consisting only of white vertices. By the white-path theorem, all ver-

tices in C 0

become descendants of y in the depth-ﬁrst tree, and by Corollary 22.8,

y:f D f .C 0 /. At time y:d, all vertices in C are white. Since there is an edge .u; /

from C to C 0

, Lemma 22.13 implies that there cannot be a path from C 0

to C .

Hence, no vertex in C is reachable from y. At time y:f , therefore, all vertices in C

are still white. Thus, for any vertex w 2 C , we have w:f > y:f , which implies

that f .C / > f .C 0 /.

The following corollary tells us that each edge in G T that goes between different

strongly connected components goes from a component with an earlier ﬁnishing

time (in the ﬁrst depth-ﬁrst search) to a component with a later ﬁnishing time.

Corollary 22.15

Let C and C 0

be distinct strongly connected components in directed graph G D

.V; E/. Suppose that there is an edge .u; / 2 E T , where u 2 C and 2 C 0

. Then

f .C / < f .C 0 /.

22.5 Strongly connected components 619

Proof Since .u; / 2 E T , we have .; u/ 2 E. Because the strongly con-

nected components of G and G T are the same, Lemma 22.14 implies that

f .C / < f .C 0 /.

Corollary 22.15 provides the key to understanding why the strongly connected

components algorithm works. Let us examine what happens when we perform the

second depth-ﬁrst search, which is on G T . We start with the strongly connected

component C whose ﬁnishing time f .C / is maximum. The search starts from

some vertex x 2 C , and it visits all vertices in C . By Corollary 22.15, G T contains

no edges from C to any other strongly connected component, and so the search

from x will not visit vertices in any other component. Thus, the tree rooted at x

contains exactly the vertices of C . Having completed visiting all vertices in C ,

the search in line 3 selects as a root a vertex from some other strongly connected

component C 0

whose ﬁnishing time f .C 0 / is maximum over all components other

than C . Again, the search will visit all vertices in C 0

, but by Corollary 22.15,

the only edges in G T from C 0

to any other component must be to C , which we

have already visited. In general, when the depth-ﬁrst search of G T in line 3 visits

any strongly connected component, any edges out of that component must be to

components that the search already visited. Each depth-ﬁrst tree, therefore, will be

exactly one strongly connected component. The following theorem formalizes this

argument.

Theorem 22.16

The STRONGLY-CONNECTED-COMPONENTS procedure correctly computes the

strongly connected components of the directed graph G provided as its input.

Proof We argue by induction on the number of depth-ﬁrst trees found in the

depth-ﬁrst search of G T in line 3 that the vertices of each tree form a strongly

connected component. The inductive hypothesis is that the ﬁrst k trees produced

in line 3 are strongly connected components. The basis for the induction, when

k D 0, is trivial.

In the inductive step, we assume that each of the ﬁrst k depth-ﬁrst trees produced

in line 3 is a strongly connected component, and we consider the .k C 1/st tree

produced. Let the root of this tree be vertex u, and let u be in strongly connected

component C . Because of how we choose roots in the depth-ﬁrst search in line 3,

u:f D f .C / > f .C 0 / for any strongly connected component C 0

other than C

that has yet to be visited. By the inductive hypothesis, at the time that the search

visits u, all other vertices of C are white. By the white-path theorem, therefore, all

other vertices of C are descendants of u in its depth-ﬁrst tree. Moreover, by the

inductive hypothesis and by Corollary 22.15, any edges in G T that leave C must be

to strongly connected components that have already been visited. Thus, no vertex

620 Chapter 22 Elementary Graph Algorithms

in any strongly connected component other than C will be a descendant of u during

the depth-ﬁrst search of G T . Thus, the vertices of the depth-ﬁrst tree in G T that is

rooted at u form exactly one strongly connected component, which completes the

inductive step and the proof.

Here is another way to look at how the second depth-ﬁrst search operates. Con-

sider the component graph .G T / SCC of G T . If we map each strongly connected

component visited in the second depth-ﬁrst search to a vertex of .G T / SCC , the sec-

ond depth-ﬁrst search visits vertices of .G T / SCC in the reverse of a topologically

sorted order. If we reverse the edges of .G T / SCC , we get the graph ..G T / SCC / T .

Because ..G T / SCC / T D G SCC (see Exercise 22.5-4), the second depth-ﬁrst search

visits the vertices of G SCC in topologically sorted order.

Exercises

22.5-1

How can the number of strongly connected components of a graph change if a new

edge is added?

22.5-2

Show how the procedure STRONGLY-CONNECTED-COMPONENTS works on the

graph of Figure 22.6. Speciﬁcally, show the ﬁnishing times computed in line 1 and

the forest produced in line 3. Assume that the loop of lines 5–7 of DFS considers

vertices in alphabetical order and that the adjacency lists are in alphabetical order.

22.5-3

Professor Bacon claims that the algorithm for strongly connected components

would be simpler if it used the original (instead of the transpose) graph in the

second depth-ﬁrst search and scanned the vertices in order of increasing ﬁnishing

times. Does this simpler algorithm always produce correct results?

22.5-4

Prove that for any directed graph G, we have ..G T / SCC / T D G SCC . That is, the

transpose of the component graph of G T is the same as the component graph of G.

22.5-5

Give an O.V C E/-time algorithm to compute the component graph of a directed

graph G D .V; E/. Make sure that there is at most one edge between two vertices

in the component graph your algorithm produces.

Problems for Chapter 22 621

22.5-6

Given a directed graph G D .V; E/, explain how to create another graph G 0 D

.V; E 0 / such that (a) G 0

has the same strongly connected components as G, (b) G 0

has the same component graph as G, and (c) E 0

is as small as possible. Describe a

fast algorithm to compute G 0

.

22.5-7

A directed graph G D .V; E/ is semiconnected if, for all pairs of vertices u; 2 V ,

we have u or u. Give an efﬁcient algorithm to determine whether

or not G is semiconnected. Prove that your algorithm is correct, and analyze its

running time.

Problems

22-1 Classifying edges by breadth-ﬁrst search

A depth-ﬁrst forest classiﬁes the edges of a graph into tree, back, forward, and

cross edges. A breadth-ﬁrst tree can also be used to classify the edges reachable

from the source of the search into the same four categories.

a. Prove that in a breadth-ﬁrst search of an undirected graph, the following prop-

erties hold:

1. There are no back edges and no forward edges.

2. For each tree edge .u; /, we have :d D u:d C 1.

3. For each cross edge .u; /, we have :d D u:d or :d D u:d C 1.

b. Prove that in a breadth-ﬁrst search of a directed graph, the following properties

hold:

1. There are no forward edges.

2. For each tree edge .u; /, we have :d D u:d C 1.

3. For each cross edge .u; /, we have :d u:d C 1.

4. For each back edge .u; /, we have 0 :d u:d.

22-2 Articulation points, bridges, and biconnected components

Let G D .V; E/ be a connected, undirected graph. An articulation point of G is

a vertex whose removal disconnects G. A bridge of G is an edge whose removal

disconnects G. A biconnected component of G is a maximal set of edges such

that any two edges in the set lie on a common simple cycle. Figure 22.10 illustrates

622 Chapter 22 Elementary Graph Algorithms

1

2

3

4

5

6

Figure 22.10 The articulation points, bridges, and biconnected components of a connected, undi-

rected graph for use in Problem 22-2. The articulation points are the heavily shaded vertices, the

bridges are the heavily shaded edges, and the biconnected components are the edges in the shaded

regions, with a bcc numbering shown.

these deﬁnitions. We can determine articulation points, bridges, and biconnected

components using depth-ﬁrst search. Let G D .V; E / be a depth-ﬁrst tree of G.

a. Prove that the root of G is an articulation point of G if and only if it has at

least two children in G .

b. Let be a nonroot vertex of G . Prove that is an articulation point of G if and

only if has a child s such that there is no back edge from s or any descendant

of s to a proper ancestor of .

c. Let

:low D min

(

:d ;

w:d W .u; w/ is a back edge for some descendant u of :

Show how to compute :low for all vertices 2 V in O.E/ time.

d. Show how to compute all articulation points in O.E/ time.

e. Prove that an edge of G is a bridge if and only if it does not lie on any simple

cycle of G.

f. Show how to compute all the bridges of G in O.E/ time.

g. Prove that the biconnected components of G partition the nonbridge edges of G.

h. Give an O.E/-time algorithm to label each edge e of G with a positive in-

teger e:bcc such that e:bcc D e 0 :bcc if and only if e and e 0

are in the same

biconnected component.

Notes for Chapter 22 623

22-3 Euler tour

An Euler tour of a strongly connected, directed graph G D .V; E/ is a cycle that

traverses each edge of G exactly once, although it may visit a vertex more than

once.

a. Show that G has an Euler tour if and only if in-degree./ D out-degree./ for

each vertex 2 V .

b. Describe an O.E/-time algorithm to ﬁnd an Euler tour of G if one exists. (Hint:

Merge edge-disjoint cycles.)

22-4 Reachability

Let G D .V; E/ be a directed graph in which each vertex u 2 V is labeled with

a unique integer L.u/ from the set f1; 2; : : : ; jV jg. For each vertex u 2 V , let

R.u/ D f 2 V W u g be the set of vertices that are reachable from u. Deﬁne

min.u/ to be the vertex in R.u/ whose label is minimum, i.e., min.u/ is the vertex

such that L./ D min fL.w/ W w 2 R.u/g. Give an O.V CE/-time algorithm that

computes min.u/ for all vertices u 2 V .

Chapter notes

Even [103] and Tarjan [330] are excellent references for graph algorithms.

Breadth-ﬁrst search was discovered by Moore [260] in the context of ﬁnding

paths through mazes. Lee [226] independently discovered the same algorithm in

the context of routing wires on circuit boards.

Hopcroft and Tarjan [178] advocated the use of the adjacency-list representation

over the adjacency-matrix representation for sparse graphs and were the ﬁrst to

recognize the algorithmic importance of depth-ﬁrst search. Depth-ﬁrst search has

been widely used since the late 1950s, especially in artiﬁcial intelligence programs.

Tarjan [327] gave a linear-time algorithm for ﬁnding strongly connected compo-

nents. The algorithm for strongly connected components in Section 22.5 is adapted

from Aho, Hopcroft, and Ullman [6], who credit it to S. R. Kosaraju (unpublished)

and M. Sharir [314]. Gabow [119] also developed an algorithm for strongly con-

nected components that is based on contracting cycles and uses two stacks to make

it run in linear time. Knuth [209] was the ﬁrst to give a linear-time algorithm for

topological sorting.

23 Minimum Spanning Trees

Electronic circuit designs often need to make the pins of several components elec-

trically equivalent by wiring them together. To interconnect a set of n pins, we can

use an arrangement of n 1 wires, each connecting two pins. Of all such arrange-

ments, the one that uses the least amount of wire is usually the most desirable.

We can model this wiring problem with a connected, undirected graph G D

.V; E/, where V is the set of pins, E is the set of possible interconnections between

pairs of pins, and for each edge .u; / 2 E, we have a weight w.u; / specifying

the cost (amount of wire needed) to connect u and . We then wish to ﬁnd an

acyclic subset T E that connects all of the vertices and whose total weight

w.T / D

X

.u;/2T

w.u; /

is minimized. Since T is acyclic and connects all of the vertices, it must form a tree,

which we call a spanning tree since it “spans” the graph G. We call the problem of

determining the tree T the minimum-spanning-tree problem. 1 Figure 23.1 shows

an example of a connected graph and a minimum spanning tree.

In this chapter, we shall examine two algorithms for solving the minimum-

spanning-tree problem: Kruskal’s algorithm and Prim’s algorithm. We can easily

make each of them run in time O.E lg V / using ordinary binary heaps. By using

Fibonacci heaps, Prim’s algorithm runs in time O.E C V lg V /, which improves

over the binary-heap implementation if jV j is much smaller than jEj.

The two algorithms are greedy algorithms, as described in Chapter 16. Each

step of a greedy algorithm must make one of several possible choices. The greedy

strategy advocates making the choice that is the best at the moment. Such a strat-

egy does not generally guarantee that it will always ﬁnd globally optimal solutions

1

The phrase “minimum spanning tree” is a shortened form of the phrase “minimum-weight spanning

tree.” We are not, for example, minimizing the number of edges in T , since all spanning trees have

exactly jV j 1 edges by Theorem B.2.

23.1 Growing a minimum spanning tree 625

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

2

7 6

Figure 23.1 A minimum spanning tree for a connected graph. The weights on edges are shown,

and the edges in a minimum spanning tree are shaded. The total weight of the tree shown is 37. This

minimum spanning tree is not unique: removing the edge .b; c/ and replacing it with the edge .a; h/

yields another spanning tree with weight 37.

to problems. For the minimum-spanning-tree problem, however, we can prove that

certain greedy strategies do yield a spanning tree with minimum weight. Although

you can read this chapter independently of Chapter 16, the greedy methods pre-

sented here are a classic application of the theoretical notions introduced there.

Section 23.1 introduces a “generic” minimum-spanning-tree method that grows

a spanning tree by adding one edge at a time. Section 23.2 gives two algorithms

that implement the generic method. The ﬁrst algorithm, due to Kruskal, is similar

to the connected-components algorithm from Section 21.1. The second, due to

Prim, resembles Dijkstra’s shortest-paths algorithm (Section 24.3).

Because a tree is a type of graph, in order to be precise we must deﬁne a tree in

terms of not just its edges, but its vertices as well. Although this chapter focuses

on trees in terms of their edges, we shall operate with the understanding that the

vertices of a tree T are those that some edge of T is incident on.

23.1 Growing a minimum spanning tree

Assume that we have a connected, undirected graph G D .V; E/ with a weight

function w W E ! R , and we wish to ﬁnd a minimum spanning tree for G. The

two algorithms we consider in this chapter use a greedy approach to the problem,

although they differ in how they apply this approach.

This greedy strategy is captured by the following generic method, which grows

the minimum spanning tree one edge at a time. The generic method manages a set

of edges A, maintaining the following loop invariant:

Prior to each iteration, A is a subset of some minimum spanning tree.

At each step, we determine an edge .u; / that we can add to A without violating

this invariant, in the sense that A[f.u; /g is also a subset of a minimum spanning

626 Chapter 23 Minimum Spanning Trees

tree. We call such an edge a safe edge for A, since we can add it safely to A while

maintaining the invariant.

GENERIC-MST.G; w/

1 A D ;

2 while A does not form a spanning tree

3 ﬁnd an edge .u; / that is safe for A

4 A D A [ f.u; /g

5 return A

We use the loop invariant as follows:

Initialization: After line 1, the set A trivially satisﬁes the loop invariant.

Maintenance: The loop in lines 2–4 maintains the invariant by adding only safe

edges.

Termination: All edges added to A are in a minimum spanning tree, and so the

set A returned in line 5 must be a minimum spanning tree.

The tricky part is, of course, ﬁnding a safe edge in line 3. One must exist, since

when line 3 is executed, the invariant dictates that there is a spanning tree T such

that A T . Within the while loop body, A must be a proper subset of T , and

therefore there must be an edge .u; / 2 T such that .u; / 62 A and .u; / is safe

for A.

In the remainder of this section, we provide a rule (Theorem 23.1) for recogniz-

ing safe edges. The next section describes two algorithms that use this rule to ﬁnd

safe edges efﬁciently.

We ﬁrst need some deﬁnitions. A cut .S; V S/ of an undirected graph G D

.V; E/ is a partition of V . Figure 23.2 illustrates this notion. We say that an edge

.u; / 2 E crosses the cut .S; V S/ if one of its endpoints is in S and the other

is in V S. We say that a cut respects a set A of edges if no edge in A crosses the

cut. An edge is a light edge crossing a cut if its weight is the minimum of any edge

crossing the cut. Note that there can be more than one light edge crossing a cut in

the case of ties. More generally, we say that an edge is a light edge satisfying a

given property if its weight is the minimum of any edge satisfying the property.

Our rule for recognizing safe edges is given by the following theorem.

Theorem 23.1

Let G D .V; E/ be a connected, undirected graph with a real-valued weight func-

tion w deﬁned on E. Let A be a subset of E that is included in some minimum

spanning tree for G, let .S; V S/ be any cut of G that respects A, and let .u; /

be a light edge crossing .S; V S/. Then, edge .u; / is safe for A.

23.1 Growing a minimum spanning tree 627

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14

4

2 1

2

7 6

a

b

d

e

h

i

g

c

f

8

11

8

7

14

10

4

6

7

4

9

2

1

2

S

(a) (b)

V – S

S

V – S

S

V – S

Figure 23.2 Two ways of viewing a cut .S; V S/ of the graph from Figure 23.1. (a) Black

vertices are in the set S, and white vertices are in V S. The edges crossing the cut are those

connecting white vertices with black vertices. The edge .d; c/ is the unique light edge crossing the

cut. A subset A of the edges is shaded; note that the cut .S; V S/ respects A, since no edge of A

crosses the cut. (b) The same graph with the vertices in the set S on the left and the vertices in the

set V S on the right. An edge crosses the cut if it connects a vertex on the left with a vertex on the

right.

Proof Let T be a minimum spanning tree that includes A, and assume that T

does not contain the light edge .u; /, since if it does, we are done. We shall

construct another minimum spanning tree T 0

that includes A [ f.u; /g by using a

cut-and-paste technique, thereby showing that .u; / is a safe edge for A.

The edge .u; / forms a cycle with the edges on the simple path p from u

to in T , as Figure 23.3 illustrates. Since u and are on opposite sides of the

cut .S; V S/, at least one edge in T lies on the simple path p and also crosses

the cut. Let .x; y/ be any such edge. The edge .x; y/ is not in A, because the cut

respects A. Since .x; y/ is on the unique simple path from u to in T , remov-

ing .x; y/ breaks T into two components. Adding .u; / reconnects them to form

a new spanning tree T 0 D T f.x; y/g [ f.u; /g.

We next show that T 0

is a minimum spanning tree. Since .u; / is a light edge

crossing .S; V S/ and .x; y/ also crosses this cut, w.u; / w.x; y/. Therefore,

w.T

0

/ D w.T / w.x; y/ C w.u; /

w.T / :

628 Chapter 23 Minimum Spanning Trees

y

v

u

x

p

Figure 23.3 The proof of Theorem 23.1. Black vertices are in S, and white vertices are in V S.

The edges in the minimum spanning tree T are shown, but the edges in the graph G are not. The

edges in A are shaded, and .u; / is a light edge crossing the cut .S; V S/. The edge .x; y/ is

an edge on the unique simple path p from u to in T . To form a minimum spanning tree T

0

that

contains .u; /, remove the edge .x; y/ from T and add the edge .u; /.

But T is a minimum spanning tree, so that w.T / w.T 0 /; thus, T 0

must be a

minimum spanning tree also.

It remains to show that .u; / is actually a safe edge for A. We have A T 0

,

since A T and .x; y/ 62 A; thus, A [ f.u; /g T 0

. Consequently, since T 0

is a

minimum spanning tree, .u; / is safe for A.

Theorem 23.1 gives us a better understanding of the workings of the GENERIC-

MST method on a connected graph G D .V; E/. As the method proceeds, the

set A is always acyclic; otherwise, a minimum spanning tree including A would

contain a cycle, which is a contradiction. At any point in the execution, the graph

G A D .V; A/ is a forest, and each of the connected components of G A is a tree.

(Some of the trees may contain just one vertex, as is the case, for example, when

the method begins: A is empty and the forest contains jV j trees, one for each

vertex.) Moreover, any safe edge .u; / for A connects distinct components of G A ,

since A [ f.u; /g must be acyclic.

The while loop in lines 2–4 of GENERIC-MST executes jV j 1 times because

it ﬁnds one of the jV j 1 edges of a minimum spanning tree in each iteration.

Initially, when A D ;, there are jV j trees in G A , and each iteration reduces that

number by 1. When the forest contains only a single tree, the method terminates.

The two algorithms in Section 23.2 use the following corollary to Theorem 23.1.

23.1 Growing a minimum spanning tree 629

Corollary 23.2

Let G D .V; E/ be a connected, undirected graph with a real-valued weight func-

tion w deﬁned on E. Let A be a subset of E that is included in some minimum

spanning tree for G, and let C D .V C ; E C / be a connected component (tree) in the

forest G A D .V; A/. If .u; / is a light edge connecting C to some other component

in G A , then .u; / is safe for A.

Proof The cut .V C ; V V C / respects A, and .u; / is a light edge for this cut.

Therefore, .u; / is safe for A.

Exercises

23.1-1

Let .u; / be a minimum-weight edge in a connected graph G. Show that .u; /

belongs to some minimum spanning tree of G.

23.1-2

Professor Sabatier conjectures the following converse of Theorem 23.1. Let G D

.V; E/ be a connected, undirected graph with a real-valued weight function w de-

ﬁned on E. Let A be a subset of E that is included in some minimum spanning

tree for G, let .S; V S/ be any cut of G that respects A, and let .u; / be a safe

edge for A crossing .S; V S/. Then, .u; / is a light edge for the cut. Show that

the professor’s conjecture is incorrect by giving a counterexample.

23.1-3

Show that if an edge .u; / is contained in some minimum spanning tree, then it is

a light edge crossing some cut of the graph.

23.1-4

Give a simple example of a connected graph such that the set of edges f.u; / W

there exists a cut .S; V S/ such that .u; / is a light edge crossing .S; V S/g

does not form a minimum spanning tree.

23.1-5

Let e be a maximum-weight edge on some cycle of connected graph G D .V; E/.

Prove that there is a minimum spanning tree of G 0 D .V; E feg/ that is also a

minimum spanning tree of G. That is, there is a minimum spanning tree of G that

does not include e.

630 Chapter 23 Minimum Spanning Trees

23.1-6

Show that a graph has a unique minimum spanning tree if, for every cut of the

graph, there is a unique light edge crossing the cut. Show that the converse is not

true by giving a counterexample.

23.1-7

Argue that if all edge weights of a graph are positive, then any subset of edges that

connects all vertices and has minimum total weight must be a tree. Give an example

to show that the same conclusion does not follow if we allow some weights to be

nonpositive.

23.1-8

Let T be a minimum spanning tree of a graph G, and let L be the sorted list of the

edge weights of T . Show that for any other minimum spanning tree T 0

of G, the

list L is also the sorted list of edge weights of T 0

.

23.1-9

Let T be a minimum spanning tree of a graph G D .V; E/, and let V 0

be a subset

of V . Let T 0

be the subgraph of T induced by V 0

, and let G 0

be the subgraph of G

induced by V 0

. Show that if T 0

is connected, then T 0

is a minimum spanning tree

of G 0

.

23.1-10

Given a graph G and a minimum spanning tree T , suppose that we decrease the

weight of one of the edges in T . Show that T is still a minimum spanning tree

for G. More formally, let T be a minimum spanning tree for G with edge weights

given by weight function w. Choose one edge .x; y/ 2 T and a positive number k,

and deﬁne the weight function w 0

by

w

0

.u; / D

(

w.u; / if .u; / ¤ .x; y/ ;

w.x; y/ k if .u; / D .x; y/ :

Show that T is a minimum spanning tree for G with edge weights given by w 0

.

23.1-11 ?

Given a graph G and a minimum spanning tree T , suppose that we decrease the

weight of one of the edges not in T . Give an algorithm for ﬁnding the minimum

spanning tree in the modiﬁed graph.

23.2 The algorithms of Kruskal and Prim 631

23.2 The algorithms of Kruskal and Prim

The two minimum-spanning-tree algorithms described in this section elaborate on

the generic method. They each use a speciﬁc rule to determine a safe edge in line 3

of GENERIC-MST. In Kruskal’s algorithm, the set A is a forest whose vertices are

all those of the given graph. The safe edge added to A is always a least-weight

edge in the graph that connects two distinct components. In Prim’s algorithm, the

set A forms a single tree. The safe edge added to A is always a least-weight edge

connecting the tree to a vertex not in the tree.

Kruskal’s algorithm

Kruskal’s algorithm ﬁnds a safe edge to add to the growing forest by ﬁnding, of all

the edges that connect any two trees in the forest, an edge .u; / of least weight.

Let C 1 and C 2 denote the two trees that are connected by .u; /. Since .u; / must

be a light edge connecting C 1 to some other tree, Corollary 23.2 implies that .u; /

is a safe edge for C 1 . Kruskal’s algorithm qualiﬁes as a greedy algorithm because

at each step it adds to the forest an edge of least possible weight.

Our implementation of Kruskal’s algorithm is like the algorithm to compute

connected components from Section 21.1. It uses a disjoint-set data structure to

maintain several disjoint sets of elements. Each set contains the vertices in one tree

of the current forest. The operation FIND-SET.u/ returns a representative element

from the set that contains u. Thus, we can determine whether two vertices u and

belong to the same tree by testing whether FIND-SET.u/ equals FIND-SET./. To

combine trees, Kruskal’s algorithm calls the UNION procedure.

MST-KRUSKAL.G; w/

1 A D ;

2 for each vertex 2 G:V

3 MAKE-SET./

4 sort the edges of G:E into nondecreasing order by weight w

5 for each edge .u; / 2 G:E, taken in nondecreasing order by weight

6 if FIND-SET.u/ ¤ FIND-SET./

7 A D A [ f.u; /g

8 UNION.u; /

9 return A

Figure 23.4 shows how Kruskal’s algorithm works. Lines 1–3 initialize the set A

to the empty set and create jV j trees, one containing each vertex. The for loop in

lines 5–8 examines edges in order of weight, from lowest to highest. The loop

632 Chapter 23 Minimum Spanning Trees

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

2

7 6

(a) (b)

(c) (d)

(e)

(g)

(f)

(h)

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

7 6

2

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

7 6

2

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

7 6

2

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

7 6

2

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

7 6

2

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

7 6

2

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

7 6

2

Figure 23.4 The execution of Kruskal’s algorithm on the graph from Figure 23.1. Shaded edges

belong to the forest A being grown. The algorithm considers each edge in sorted order by weight.

An arrow points to the edge under consideration at each step of the algorithm. If the edge joins two

distinct trees in the forest, it is added to the forest, thereby merging the two trees.

checks, for each edge .u; /, whether the endpoints u and belong to the same

tree. If they do, then the edge .u; / cannot be added to the forest without creating

a cycle, and the edge is discarded. Otherwise, the two vertices belong to different

trees. In this case, line 7 adds the edge .u; / to A, and line 8 merges the vertices

in the two trees.

23.2 The algorithms of Kruskal and Prim 633

(i) (j)

(k) (l)

(n) (m)

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

7 6

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

7 6

2 2

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

7 6

2

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

7 6

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

7 6

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

7 6

2

2

2

Figure 23.4, continued Further steps in the execution of Kruskal’s algorithm.

The running time of Kruskal’s algorithm for a graph G D .V; E/ depends

on how we implement the disjoint-set data structure. We assume that we use

the disjoint-set-forest implementation of Section 21.3 with the union-by-rank and

path-compression heuristics, since it is the asymptotically fastest implementation

known. Initializing the set A in line 1 takes O.1/ time, and the time to sort the

edges in line 4 is O.E lg E/. (We will account for the cost of the jV j MAKE-SET

operations in the for loop of lines 2–3 in a moment.) The for loop of lines 5–8

performs O.E/ FIND-SET and UNION operations on the disjoint-set forest. Along

with the jV j MAKE-SET operations, these take a total of O..V C E/ ˛.V // time,

where ˛ is the very slowly growing function deﬁned in Section 21.4. Because we

assume that G is connected, we have jEj jV j 1, and so the disjoint-set opera-

tions take O.E˛.V // time. Moreover, since ˛.jV j/ D O.lg V / D O.lg E/, the to-

tal running time of Kruskal’s algorithm is O.E lg E/. Observing that jEj < jV j

2

,

we have lg jEj D O.lg V /, and so we can restate the running time of Kruskal’s

algorithm as O.E lg V /.

634 Chapter 23 Minimum Spanning Trees

Prim’s algorithm

Like Kruskal’s algorithm, Prim’s algorithm is a special case of the generic min-

imum-spanning-tree method from Section 23.1. Prim’s algorithm operates much

like Dijkstra’s algorithm for ﬁnding shortest paths in a graph, which we shall see in

Section 24.3. Prim’s algorithm has the property that the edges in the set A always

form a single tree. As Figure 23.5 shows, the tree starts from an arbitrary root

vertex r and grows until the tree spans all the vertices in V . Each step adds to the

tree A a light edge that connects A to an isolated vertex—one on which no edge

of A is incident. By Corollary 23.2, this rule adds only edges that are safe for A;

therefore, when the algorithm terminates, the edges in A form a minimum spanning

tree. This strategy qualiﬁes as greedy since at each step it adds to the tree an edge

that contributes the minimum amount possible to the tree’s weight.

In order to implement Prim’s algorithm efﬁciently, we need a fast way to select

a new edge to add to the tree formed by the edges in A. In the pseudocode below,

the connected graph G and the root r of the minimum spanning tree to be grown

are inputs to the algorithm. During execution of the algorithm, all vertices that

are not in the tree reside in a min-priority queue Q based on a key attribute. For

each vertex , the attribute :key is the minimum weight of any edge connecting

to a vertex in the tree; by convention, :key D 1 if there is no such edge. The

attribute : names the parent of in the tree. The algorithm implicitly maintains

the set A from GENERIC-MST as

A D f.; :/ W 2 V frg Qg :

When the algorithm terminates, the min-priority queue Q is empty; the minimum

spanning tree A for G is thus

A D f.; :/ W 2 V frgg :

MST-PRIM.G; w; r/

1 for each u 2 G:V

2 u:key D 1

3 u: D NIL

4 r:key D 0

5 Q D G:V

6 while Q ¤ ;

7 u D EXTRACT-MIN.Q/

8 for each 2 G:AdjŒu

9 if 2 Q and w.u; / < :key

10 : D u

11 :key D w.u; /

23.2 The algorithms of Kruskal and Prim 635

(a) (b)

(c) (d)

(e) (f)

(g) (h)

(i)

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

2

7 6

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

2

7 6

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

2

7 6

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

7 6

2

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

7 6

2

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

7 6

2

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

7 6

2

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

7 6

2

b

a

h

c

g

i

d

f

e

4

8

11

8 7

9

10

14 4

2 1

7 6

2

Figure 23.5 The execution of Prim’s algorithm on the graph from Figure 23.1. The root vertex

is a. Shaded edges are in the tree being grown, and black vertices are in the tree. At each step of

the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut

is added to the tree. In the second step, for example, the algorithm has a choice of adding either

edge .b; c/ or edge .a; h/ to the tree since both are light edges crossing the cut.

636 Chapter 23 Minimum Spanning Trees

Figure 23.5 shows how Prim’s algorithm works. Lines 1–5 set the key of each

vertex to 1 (except for the root r, whose key is set to 0 so that it will be the

ﬁrst vertex processed), set the parent of each vertex to NIL, and initialize the min-

priority queue Q to contain all the vertices. The algorithm maintains the following

three-part loop invariant:

Prior to each iteration of the while loop of lines 6–11,

1. A D f.; :/ W 2 V frg Qg.

2. The vertices already placed into the minimum spanning tree are those in

V Q.

3. For all vertices 2 Q, if : ¤ NIL, then :key < 1 and :key is

the weight of a light edge .; :/ connecting to some vertex already

placed into the minimum spanning tree.

Line 7 identiﬁes a vertex u 2 Q incident on a light edge that crosses the cut

.V Q; Q/ (with the exception of the ﬁrst iteration, in which u D r due to line 4).

Removing u from the set Q adds it to the set V Q of vertices in the tree, thus

adding .u; u:/ to A. The for loop of lines 8–11 updates the key and attributes

of every vertex adjacent to u but not in the tree, thereby maintaining the third

part of the loop invariant.

The running time of Prim’s algorithm depends on how we implement the min-

priority queue Q. If we implement Q as a binary min-heap (see Chapter 6), we

can use the BUILD-MIN-HEAP procedure to perform lines 1–5 in O.V / time. The

body of the while loop executes jV j times, and since each EXTRACT-MIN opera-

tion takes O.lg V / time, the total time for all calls to EXTRACT-MIN is O.V lg V /.

The for loop in lines 8–11 executes O.E/ times altogether, since the sum of the

lengths of all adjacency lists is 2 jEj. Within the for loop, we can implement the

test for membership in Q in line 9 in constant time by keeping a bit for each vertex

that tells whether or not it is in Q, and updating the bit when the vertex is removed

from Q. The assignment in line 11 involves an implicit DECREASE-KEY opera-

tion on the min-heap, which a binary min-heap supports in O.lg V / time. Thus,

the total time for Prim’s algorithm is O.V lg V C E lg V / D O.E lg V /, which is

asymptotically the same as for our implementation of Kruskal’s algorithm.

We can improve the asymptotic running time of Prim’s algorithm by using Fi-

bonacci heaps. Chapter 19 shows that if a Fibonacci heap holds jV j elements, an

EXTRACT-MIN operation takes O.lg V / amortized time and a DECREASE-KEY

operation (to implement line 11) takes O.1/ amortized time. Therefore, if we use a

Fibonacci heap to implement the min-priority queue Q, the running time of Prim’s

algorithm improves to O.E C V lg V /.

23.2 The algorithms of Kruskal and Prim 637

Exercises

23.2-1

Kruskal’s algorithm can return different spanning trees for the same input graph G,

depending on how it breaks ties when the edges are sorted into order. Show that

for each minimum spanning tree T of G, there is a way to sort the edges of G in

Kruskal’s algorithm so that the algorithm returns T .

23.2-2

Suppose that we represent the graph G D .V; E/ as an adjacency matrix. Give a

simple implementation of Prim’s algorithm for this case that runs in O.V 2 / time.

23.2-3

For a sparse graph G D .V; E/, where jEj D ‚.V /, is the implementation of

Prim’s algorithm with a Fibonacci heap asymptotically faster than the binary-heap

implementation? What about for a dense graph, where jEj D ‚.V 2 /? How

must the sizes jEj and jV j be related for the Fibonacci-heap implementation to

be asymptotically faster than the binary-heap implementation?

23.2-4

Suppose that all edge weights in a graph are integers in the range from 1 to jV j.

How fast can you make Kruskal’s algorithm run? What if the edge weights are

integers in the range from 1 to W for some constant W ?

23.2-5

Suppose that all edge weights in a graph are integers in the range from 1 to jV j.

How fast can you make Prim’s algorithm run? What if the edge weights are integers

in the range from 1 to W for some constant W ?

23.2-6 ?

Suppose that the edge weights in a graph are uniformly distributed over the half-

open interval Œ0; 1/. Which algorithm, Kruskal’s or Prim’s, can you make run

faster?

23.2-7 ?

Suppose that a graph G has a minimum spanning tree already computed. How

quickly can we update the minimum spanning tree if we add a new vertex and

incident edges to G?

23.2-8

Professor Borden proposes a new divide-and-conquer algorithm for computing

minimum spanning trees, which goes as follows. Given a graph G D .V; E/,

partition the set V of vertices into two sets V 1 and V 2 such that jV 1 j and jV 2 j differ

638 Chapter 23 Minimum Spanning Trees

by at most 1. Let E 1 be the set of edges that are incident only on vertices in V 1 , and

let E 2 be the set of edges that are incident only on vertices in V 2 . Recursively solve

a minimum-spanning-tree problem on each of the two subgraphs G 1 D .V 1 ; E 1 /

and G 2 D .V 2 ; E 2 /. Finally, select the minimum-weight edge in E that crosses the

cut .V 1 ; V 2 /, and use this edge to unite the resulting two minimum spanning trees

into a single spanning tree.

Either argue that the algorithm correctly computes a minimum spanning tree

of G, or provide an example for which the algorithm fails.

Problems

23-1 Second-best minimum spanning tree

Let G D .V; E/ be an undirected, connected graph whose weight function is

w W E ! R , and suppose that jEj jV j and all edge weights are distinct.

We deﬁne a second-best minimum spanning tree as follows. Let T be the set

of all spanning trees of G, and let T 0

be a minimum spanning tree of G. Then

a second-best minimum spanning tree is a spanning tree T such that w.T / D

min T 00 2T fT 0 g fw.T 00 /g.

a. Show that the minimum spanning tree is unique, but that the second-best mini-

mum spanning tree need not be unique.

b. Let T be the minimum spanning tree of G. Prove that G contains edges

.u; / 2 T and .x; y/ 62 T such that T f.u; /g [ f.x; y/g is a second-best

minimum spanning tree of G.

c. Let T be a spanning tree of G and, for any two vertices u; 2 V , let maxŒu;

denote an edge of maximum weight on the unique simple path between u and

in T . Describe an O.V 2 /-time algorithm that, given T , computes maxŒu; for

all u; 2 V .

d. Give an efﬁcient algorithm to compute the second-best minimum spanning tree

of G.

23-2 Minimum spanning tree in sparse graphs

For a very sparse connected graph G D .V; E/, we can further improve upon the

O.E C V lg V / running time of Prim’s algorithm with Fibonacci heaps by prepro-

cessing G to decrease the number of vertices before running Prim’s algorithm. In

particular, we choose, for each vertex u, the minimum-weight edge .u; / incident

on u, and we put .u; / into the minimum spanning tree under construction. We

Problems for Chapter 23 639

then contract all chosen edges (see Section B.4). Rather than contracting these

edges one at a time, we ﬁrst identify sets of vertices that are united into the same

new vertex. Then we create the graph that would have resulted from contracting

these edges one at a time, but we do so by “renaming” edges according to the sets

into which their endpoints were placed. Several edges from the original graph may

be renamed the same as each other. In such a case, only one edge results, and its

weight is the minimum of the weights of the corresponding original edges.

Initially, we set the minimum spanning tree T being constructed to be empty,

and for each edge .u; / 2 E, we initialize the attributes .u; /:orig D .u; /

and .u; /:c D w.u; /. We use the orig attribute to reference the edge from the

initial graph that is associated with an edge in the contracted graph. The c attribute

holds the weight of an edge, and as edges are contracted, we update it according to

the above scheme for choosing edge weights. The procedure MST-REDUCE takes

inputs G and T , and it returns a contracted graph G 0

with updated attributes orig

0

and c 0

. The procedure also accumulates edges of G into the minimum spanning

tree T .

MST-REDUCE.G; T /

1 for each 2 G:V

2 :mark D FALSE

3 MAKE-SET./

4 for each u 2 G:V

5 if u:mark == FALSE

6 choose 2 G:AdjŒu such that .u; /:c is minimized

7 UNION.u; /

8 T D T [ f.u; /:origg

9 u:mark D :mark D TRUE

10 G 0 :V D fFIND-SET./ W 2 G:Vg

11 G 0 :E D ;

12 for each .x; y/ 2 G:E

13 u D FIND-SET.x/

14 D FIND-SET.y/

15 if .u; / 62 G 0 :E

16 G 0 :E D G 0 :E [ f.u; /g

17 .u; /:orig

0

D .x; y/:orig

18 .u; /:c 0 D .x; y/:c

19 else if .x; y/:c < .u; /:c 0

20 .u; /:orig

0

D .x; y/:orig

21 .u; /:c 0 D .x; y/:c

22 construct adjacency lists G 0 :Adj for G 0

23 return G 0

and T

640 Chapter 23 Minimum Spanning Trees

a. Let T be the set of edges returned by MST-REDUCE, and let A be the minimum

spanning tree of the graph G 0

formed by the call MST-PRIM.G 0 ; c 0 ; r/, where c 0

is the weight attribute on the edges of G 0 :E and r is any vertex in G 0 :V. Prove

that T [ f.x; y/:orig

0

W .x; y/ 2 Ag is a minimum spanning tree of G.

b. Argue that jG 0 :Vj jV j =2.

c. Show how to implement MST-REDUCE so that it runs in O.E/ time. (Hint:

Use simple data structures.)

d. Suppose that we run k phases of MST-REDUCE, using the output G 0

produced

by one phase as the input G to the next phase and accumulating edges in T .

Argue that the overall running time of the k phases is O.kE/.

e. Suppose that after running k phases of MST-REDUCE, as in part (d), we run

Prim’s algorithm by calling MST-PRIM.G 0 ; c 0 ; r/, where G 0

, with weight at-

tribute c 0

, is returned by the last phase and r is any vertex in G 0 :V. Show how

to pick k so that the overall running time is O.E lg lg V /. Argue that your

choice of k minimizes the overall asymptotic running time.

f. For what values of jEj (in terms of jV j) does Prim’s algorithm with preprocess-

ing asymptotically beat Prim’s algorithm without preprocessing?

23-3 Bottleneck spanning tree

A bottleneck spanning tree T of an undirected graph G is a spanning tree of G

whose largest edge weight is minimum over all spanning trees of G. We say that

the value of the bottleneck spanning tree is the weight of the maximum-weight

edge in T .

a. Argue that a minimum spanning tree is a bottleneck spanning tree.

Part (a) shows that ﬁnding a bottleneck spanning tree is no harder than ﬁnding

a minimum spanning tree. In the remaining parts, we will show how to ﬁnd a

bottleneck spanning tree in linear time.

b. Give a linear-time algorithm that given a graph G and an integer b, determines

whether the value of the bottleneck spanning tree is at most b.

c. Use your algorithm for part (b) as a subroutine in a linear-time algorithm for

the bottleneck-spanning-tree problem. (Hint: You may want to use a subroutine

that contracts sets of edges, as in the MST-REDUCE procedure described in

Problem 23-2.)

Notes for Chapter 23 641

23-4 Alternative minimum-spanning-tree algorithms

In this problem, we give pseudocode for three different algorithms. Each one takes

a connected graph and a weight function as input and returns a set of edges T . For

each algorithm, either prove that T is a minimum spanning tree or prove that T is

not a minimum spanning tree. Also describe the most efﬁcient implementation of

each algorithm, whether or not it computes a minimum spanning tree.

a. MAYBE-MST-A.G; w/

1 sort the edges into nonincreasing order of edge weights w

2 T D E

3 for each edge e, taken in nonincreasing order by weight

4 if T feg is a connected graph

5 T D T feg

6 return T

b. MAYBE-MST-B.G; w/

1 T D ;

2 for each edge e, taken in arbitrary order

3 if T [ feg has no cycles

4 T D T [ feg

5 return T

c. MAYBE-MST-C.G; w/

1 T D ;

2 for each edge e, taken in arbitrary order

3 T D T [ feg

4 if T has a cycle c

5 let e 0

be a maximum-weight edge on c

6 T D T fe 0 g

7 return T

Chapter notes

Tarjan [330] surveys the minimum-spanning-tree problem and provides excellent

advanced material. Graham and Hell [151] compiled a history of the minimum-

spanning-tree problem.

Tarjan attributes the ﬁrst minimum-spanning-tree algorithm to a 1926 paper by

O. Bor˙uvka. Bor˙uvka’s algorithm consists of running O.lg V / iterations of the

642 Chapter 23 Minimum Spanning Trees

procedure MST-REDUCE described in Problem 23-2. Kruskal’s algorithm was

reported by Kruskal [222] in 1956. The algorithm commonly known as Prim’s

algorithm was indeed invented by Prim [285], but it was also invented earlier by

V. Jarn´ık in 1930.

The reason underlying why greedy algorithms are effective at ﬁnding minimum

spanning trees is that the set of forests of a graph forms a graphic matroid. (See

Section 16.4.)

When jEj D .V lg V /, Prim’s algorithm, implemented with Fibonacci heaps,

runs in O.E/ time. For sparser graphs, using a combination of the ideas from

Prim’s algorithm, Kruskal’s algorithm, and Bor˙uvka’s algorithm, together with ad-

vanced data structures, Fredman and Tarjan [114] give an algorithm that runs in

O.E lg

V / time. Gabow, Galil, Spencer, and Tarjan [120] improved this algo-

rithm to run in O.E lg lg

V / time. Chazelle [60] gives an algorithm that runs

in O.E y˛.E; V // time, where y˛.E; V / is the functional inverse of Ackermann’s

function. (See the chapter notes for Chapter 21 for a brief discussion of Acker-

mann’s function and its inverse.) Unlike previous minimum-spanning-tree algo-

rithms, Chazelle’s algorithm does not follow the greedy method.

A related problem is spanning-tree veriﬁcation, in which we are given a graph

G D .V; E/ and a tree T E, and we wish to determine whether T is a minimum

spanning tree of G. King [203] gives a linear-time algorithm to verify a spanning

tree, building on earlier work of Koml´os [215] and Dixon, Rauch, and Tarjan [90].

The above algorithms are all deterministic and fall into the comparison-based

model described in Chapter 8. Karger, Klein, and Tarjan [195] give a randomized

minimum-spanning-tree algorithm that runs in O.V C E/ expected time. This

algorithm uses recursion in a manner similar to the linear-time selection algorithm

in Section 9.3: a recursive call on an auxiliary problem identiﬁes a subset of the

edges E 0

that cannot be in any minimum spanning tree. Another recursive call

on E E 0

then ﬁnds the minimum spanning tree. The algorithm also uses ideas

from Bor˙uvka’s algorithm and King’s algorithm for spanning-tree veriﬁcation.

Fredman and Willard [116] showed how to ﬁnd a minimum spanning tree in

O.V CE/ time using a deterministic algorithm that is not comparison based. Their

algorithm assumes that the data are b-bit integers and that the computer memory

consists of addressable b-bit words.

24 Single-Source Shortest Paths

Professor Patrick wishes to ﬁnd the shortest possible route from Phoenix to Indi-

anapolis. Given a road map of the United States on which the distance between

each pair of adjacent intersections is marked, how can she determine this shortest

route?

One possible way would be to enumerate all the routes from Phoenix to Indi-

anapolis, add up the distances on each route, and select the shortest. It is easy to

see, however, that even disallowing routes that contain cycles, Professor Patrick

would have to examine an enormous number of possibilities, most of which are

simply not worth considering. For example, a route from Phoenix to Indianapolis

that passes through Seattle is obviously a poor choice, because Seattle is several

hundred miles out of the way.

In this chapter and in Chapter 25, we show how to solve such problems ef-

ﬁciently. In a shortest-paths problem, we are given a weighted, directed graph

G D .V; E/, with weight function w W E ! R mapping edges to real-valued

weights. The weight w.p/ of path p D h 0 ; 1 ; : : : ; k i is the sum of the weights

of its constituent edges:

w.p/ D

k X

iD1

w. i1 ; i / :

We deﬁne the shortest-path weight ı.u; / from u to by

ı.u; / D

(

minfw.p/ W u

p g if there is a path from u to ;

1 otherwise :

A shortest path from vertex u to vertex is then deﬁned as any path p with weight

w.p/ D ı.u; /.

In the Phoenix-to-Indianapolis example, we can model the road map as a graph:

vertices represent intersections, edges represent road segments between intersec-

tions, and edge weights represent road distances. Our goal is to ﬁnd a shortest path

from a given intersection in Phoenix to a given intersection in Indianapolis.

644 Chapter 24 Single-Source Shortest Paths

Edge weights can represent metrics other than distances, such as time, cost,

penalties, loss, or any other quantity that accumulates linearly along a path and

that we would want to minimize.

The breadth-ﬁrst-search algorithm from Section 22.2 is a shortest-paths algo-

rithm that works on unweighted graphs, that is, graphs in which each edge has unit

weight. Because many of the concepts from breadth-ﬁrst search arise in the study

of shortest paths in weighted graphs, you might want to review Section 22.2 before

proceeding.

Variants

In this chapter, we shall focus on the single-source shortest-paths problem: given

a graph G D .V; E/, we want to ﬁnd a shortest path from a given source vertex

s 2 V to each vertex 2 V . The algorithm for the single-source problem can

solve many other problems, including the following variants.

Single-destination shortest-paths problem: Find a shortest path to a given des-

tination vertex t from each vertex . By reversing the direction of each edge in

the graph, we can reduce this problem to a single-source problem.

Single-pair shortest-path problem: Find a shortest path from u to for given

vertices u and . If we solve the single-source problem with source vertex u,

we solve this problem also. Moreover, all known algorithms for this problem

have the same worst-case asymptotic running time as the best single-source

algorithms.

All-pairs shortest-paths problem: Find a shortest path from u to for every pair

of vertices u and . Although we can solve this problem by running a single-

source algorithm once from each vertex, we usually can solve it faster. Addi-

tionally, its structure is interesting in its own right. Chapter 25 addresses the

all-pairs problem in detail.

Optimal substructure of a shortest path

Shortest-paths algorithms typically rely on the property that a shortest path be-

tween two vertices contains other shortest paths within it. (The Edmonds-Karp

maximum-ﬂow algorithm in Chapter 26 also relies on this property.) Recall

that optimal substructure is one of the key indicators that dynamic programming

(Chapter 15) and the greedy method (Chapter 16) might apply. Dijkstra’s algo-

rithm, which we shall see in Section 24.3, is a greedy algorithm, and the Floyd-

Warshall algorithm, which ﬁnds shortest paths between all pairs of vertices (see

Section 25.2), is a dynamic-programming algorithm. The following lemma states

the optimal-substructure property of shortest paths more precisely.

Chapter 24 Single-Source Shortest Paths 645

Lemma 24.1 (Subpaths of shortest paths are shortest paths)

Given a weighted, directed graph G D .V; E/ with weight function w W E ! R ,

let p D h 0 ; 1 ; : : : ; k i be a shortest path from vertex 0 to vertex k and, for any

i and j such that 0 i j k, let p ij D h i ; iC1 ; : : : ; j i be the subpath of p

from vertex i to vertex j . Then, p ij is a shortest path from i to j .

Proof If we decompose path p into 0

p 0i i

p ij j

p jk k , then we have that

w.p/ D w.p 0i / C w.p ij / C w.p jk /. Now, assume that there is a path p 0

ij

from i

to j with weight w.p 0

ij

/ < w.p ij /. Then, 0

p 0i i

p

0

ij j

p jk k is a path from 0

to k whose weight w.p 0i /Cw.p 0

ij

/Cw.p jk / is less than w.p/, which contradicts

the assumption that p is a shortest path from 0 to k .

Negative-weight edges

Some instances of the single-source shortest-paths problem may include edges

whose weights are negative. If the graph G D .V; E/ contains no negative-

weight cycles reachable from the source s, then for all 2 V , the shortest-path

weight ı.s; / remains well deﬁned, even if it has a negative value. If the graph

contains a negative-weight cycle reachable from s, however, shortest-path weights

are not well deﬁned. No path from s to a vertex on the cycle can be a short-

est path—we can always ﬁnd a path with lower weight by following the proposed

“shortest” path and then traversing the negative-weight cycle. If there is a negative-

weight cycle on some path from s to , we deﬁne ı.s; / D 1.

Figure 24.1 illustrates the effect of negative weights and negative-weight cy-

cles on shortest-path weights. Because there is only one path from s to a (the

path hs; ai), we have ı.s; a/ D w.s; a/ D 3. Similarly, there is only one path

from s to b, and so ı.s; b/ D w.s; a/ C w.a; b/ D 3 C .4/ D 1. There are

inﬁnitely many paths from s to c: hs; ci, hs; c; d; ci, hs; c; d; c; d; ci, and so on.

Because the cycle hc; d; ci has weight 6 C .3/ D 3 > 0, the shortest path from s

to c is hs;ci, with weight ı.s; c/ D w.s; c/ D 5. Similarly, the shortest path from s

to d is hs;c;di, with weight ı.s; d/ D w.s; c/Cw.c; d/ D 11. Analogously, there

are inﬁnitely many paths from s to e: hs; ei, hs; e; f; ei, hs; e; f; e; f; ei, and so

on. Because the cycle he; f; ei has weight 3 C .6/ D 3 < 0, however, there

is no shortest path from s to e. By traversing the negative-weight cycle he; f; ei

arbitrarily many times, we can ﬁnd paths from s to e with arbitrarily large negative

weights, and so ı.s; e/ D 1. Similarly, ı.s; f / D 1. Because g is reachable

from f , we can also ﬁnd paths with arbitrarily large negative weights from s to g,

and so ı.s; g/ D 1. Vertices h, i, and j also form a negative-weight cycle. They

are not reachable from s, however, and so ı.s; h/ D ı.s; i/ D ı.s; j / D 1.

646 Chapter 24 Single-Source Shortest Paths

5

c

11

d 6

–3

– ∞

e

– ∞

f 3

–6

3

a

–1

b

0

s

– ∞

g

–4

5

3

2

8

4

7

∞

h

∞

i

2

∞

j

–8 3

Figure 24.1 Negative edge weights in a directed graph. The shortest-path weight from source s

appears within each vertex. Because vertices e and f form a negative-weight cycle reachable from s,

they have shortest-path weights of 1. Because vertex g is reachable from a vertex whose shortest-

path weight is 1, it, too, has a shortest-path weight of 1. Vertices such as h, i, and j are not

reachable from s, and so their shortest-path weights are 1, even though they lie on a negative-weight

cycle.

Some shortest-paths algorithms, such as Dijkstra’s algorithm, assume that all

edge weights in the input graph are nonnegative, as in the road-map example. Oth-

ers, such as the Bellman-Ford algorithm, allow negative-weight edges in the in-

put graph and produce a correct answer as long as no negative-weight cycles are

reachable from the source. Typically, if there is such a negative-weight cycle, the

algorithm can detect and report its existence.

Cycles

Can a shortest path contain a cycle? As we have just seen, it cannot contain a

negative-weight cycle. Nor can it contain a positive-weight cycle, since remov-

ing the cycle from the path produces a path with the same source and destination

vertices and a lower path weight. That is, if p D h 0 ; 1 ; : : : ; k i is a path and

c D h i ; iC1 ; : : : ; j i is a positive-weight cycle on this path (so that i D j and

w.c/ > 0), then the path p 0 D h 0 ; 1 ; : : : ; i ; j C1 ; j C2 ; : : : ; k i has weight

w.p 0 / D w.p/ w.c/ < w.p/, and so p cannot be a shortest path from 0 to k .

That leaves only 0-weight cycles. We can remove a 0-weight cycle from any

path to produce another path whose weight is the same. Thus, if there is a shortest

path from a source vertex s to a destination vertex that contains a 0-weight cycle,

then there is another shortest path from s to without this cycle. As long as a

shortest path has 0-weight cycles, we can repeatedly remove these cycles from the

path until we have a shortest path that is cycle-free. Therefore, without loss of

generality we can assume that when we are ﬁnding shortest paths, they have no

cycles, i.e., they are simple paths. Since any acyclic path in a graph G D .V; E/

Chapter 24 Single-Source Shortest Paths 647

contains at most jV j distinct vertices, it also contains at most jV j 1 edges. Thus,

we can restrict our attention to shortest paths of at most jV j 1 edges.

Representing shortest paths

We often wish to compute not only shortest-path weights, but the vertices on short-

est paths as well. We represent shortest paths similarly to how we represented

breadth-ﬁrst trees in Section 22.2. Given a graph G D .V; E/, we maintain for

each vertex 2 V a predecessor : that is either another vertex or NIL. The

shortest-paths algorithms in this chapter set the attributes so that the chain of pre-

decessors originating at a vertex runs backwards along a shortest path from s to .

Thus, given a vertex for which : ¤ NIL, the procedure PRINT-PATH.G; s; /

from Section 22.2 will print a shortest path from s to .

In the midst of executing a shortest-paths algorithm, however, the values might

not indicate shortest paths. As in breadth-ﬁrst search, we shall be interested in the

predecessor subgraph G D .V ; E / induced by the values. Here again, we

deﬁne the vertex set V to be the set of vertices of G with non-NIL predecessors,

plus the source s:

V D f 2 V W : ¤ NILg [ fsg :

The directed edge set E is the set of edges induced by the values for vertices

in V :

E D f.:; / 2 E W 2 V fsgg :

We shall prove that the values produced by the algorithms in this chapter have

the property that at termination G is a “shortest-paths tree”—informally, a rooted

tree containing a shortest path from the source s to every vertex that is reachable

from s. A shortest-paths tree is like the breadth-ﬁrst tree from Section 22.2, but it

contains shortest paths from the source deﬁned in terms of edge weights instead of

numbers of edges. To be precise, let G D .V; E/ be a weighted, directed graph

with weight function w W E ! R , and assume that G contains no negative-weight

cycles reachable from the source vertex s 2 V , so that shortest paths are well

deﬁned. A shortest-paths tree rooted at s is a directed subgraph G 0 D .V 0 ; E 0 /,

where V 0 V and E 0 E, such that

1. V 0

is the set of vertices reachable from s in G,

2. G 0

forms a rooted tree with root s, and

3. for all 2 V 0

, the unique simple path from s to in G 0

is a shortest path from s

to in G.

648 Chapter 24 Single-Source Shortest Paths

(a) (b) (c)

0

6

6

7 2 1 2

4

3

5

3

s

t x

y z

3 9

5 11

0

6

6

7 2 1 2

4

3

5

3

s

t x

y z

3 9

5 11

0

6

6

7 2 1 2

4

3

5

3

s

t x

y z

3 9

5 11

Figure 24.2 (a) A weighted, directed graph with shortest-path weights from source s. (b) The

shaded edges form a shortest-paths tree rooted at the source s. (c) Another shortest-paths tree with

the same root.

Shortest paths are not necessarily unique, and neither are shortest-paths trees. For

example, Figure 24.2 shows a weighted, directed graph and two shortest-paths trees

with the same root.

Relaxation

The algorithms in this chapter use the technique of relaxation. For each vertex

2 V , we maintain an attribute :d, which is an upper bound on the weight of

a shortest path from source s to . We call :d a shortest-path estimate. We

initialize the shortest-path estimates and predecessors by the following ‚.V /-time

procedure:

INITIALIZE-SINGLE-SOURCE.G; s/

1 for each vertex 2 G:V

2 :d D 1

3 : D NIL

4 s:d D 0

After initialization, we have : D NIL for all 2 V , s:d D 0, and :d D 1 for

2 V fsg.

The process of relaxing an edge .u; / consists of testing whether we can im-

prove the shortest path to found so far by going through u and, if so, updat-

ing :d and :. A relaxation step 1 may decrease the value of the shortest-path

1

The use of the term is historical. The outcome of a relaxation step can be viewed as a relaxation

of the constraint :d u:d C w.u; /, which, by the triangle inequality (Lemma 24.10), must be

satisﬁed if u:d D ı.s; u/ and :d D ı.s; /. That is, if :d u:d C w.u; /, there is no “pressure”

It may seem strange that the term “relaxation” is used for an operation that tightens an upper bound.

so the constraint is “relaxed.” to satisfy this constraint,

Chapter 24 Single-Source Shortest Paths 649

u v

5 9

2

u v

5 7

2

RELAX(u,v,w)

(a) (b)

u v

5 6

2

u v

5 6

2

RELAX(u,v,w)

Figure 24.3 Relaxing an edge .u; / with weight w.u; / D 2. The shortest-path estimate of each

vertex appears within the vertex. (a) Because :d > u:d C w.u; / prior to relaxation, the value

of :d decreases. (b) Here, :d u:d C w.u; / before relaxing the edge, and so the relaxation step

leaves :d unchanged.

estimate :d and update ’s predecessor attribute :. The following code per-

forms a relaxation step on edge .u; / in O.1/ time:

RELAX.u; ; w/

1 if :d > u:d C w.u; /

2 :d D u:d C w.u; /

3 : D u

Figure 24.3 shows two examples of relaxing an edge, one in which a shortest-path

estimate decreases and one in which no estimate changes.

Each algorithm in this chapter calls INITIALIZE-SINGLE-SOURCE and then re-

peatedly relaxes edges. Moreover, relaxation is the only means by which shortest-

path estimates and predecessors change. The algorithms in this chapter differ in

how many times they relax each edge and the order in which they relax edges. Dijk-

stra’s algorithm and the shortest-paths algorithm for directed acyclic graphs relax

each edge exactly once. The Bellman-Ford algorithm relaxes each edge jV j 1

times.

Properties of shortest paths and relaxation

To prove the algorithms in this chapter correct, we shall appeal to several prop-

erties of shortest paths and relaxation. We state these properties here, and Sec-

tion 24.5 proves them formally. For your reference, each property stated here in-

cludes the appropriate lemma or corollary number from Section 24.5. The latter

ﬁve of these properties, which refer to shortest-path estimates or the predecessor

subgraph, implicitly assume that the graph is initialized with a call to INITIALIZE-

SINGLE-SOURCE.G; s/ and that the only way that shortest-path estimates and the

predecessor subgraph change are by some sequence of relaxation steps.

650 Chapter 24 Single-Source Shortest Paths

Triangle inequality (Lemma 24.10)

For any edge .u; / 2 E, we have ı.s; / ı.s; u/ C w.u; /.

Upper-bound property (Lemma 24.11)

We always have :d ı.s; / for all vertices 2 V , and once :d achieves the

value ı.s; /, it never changes.

No-path property (Corollary 24.12)

If there is no path from s to , then we always have :d D ı.s; / D 1.

Convergence property (Lemma 24.14)

If s u ! is a shortest path in G for some u; 2 V , and if u:d D ı.s; u/ at

any time prior to relaxing edge .u; /, then :d D ı.s; / at all times afterward.

Path-relaxation property (Lemma 24.15)

If p D h 0 ; 1 ; : : : ; k i is a shortest path from s D 0 to k , and we relax the

edges of p in the order . 0 ; 1 /; . 1 ; 2 /; : : : ; . k1 ; k /, then k :d D ı.s; k /.

This property holds regardless of any other relaxation steps that occur, even if

they are intermixed with relaxations of the edges of p.

Predecessor-subgraph property (Lemma 24.17)

Once :d D ı.s; / for all 2 V , the predecessor subgraph is a shortest-paths

tree rooted at s.

Chapter outline

Section 24.1 presents the Bellman-Ford algorithm, which solves the single-source

shortest-paths problem in the general case in which edges can have negative weight.

The Bellman-Ford algorithm is remarkably simple, and it has the further beneﬁt

of detecting whether a negative-weight cycle is reachable from the source. Sec-

tion 24.2 gives a linear-time algorithm for computing shortest paths from a single

source in a directed acyclic graph. Section 24.3 covers Dijkstra’s algorithm, which

has a lower running time than the Bellman-Ford algorithm but requires the edge

weights to be nonnegative. Section 24.4 shows how we can use the Bellman-Ford

algorithm to solve a special case of linear programming. Finally, Section 24.5

proves the properties of shortest paths and relaxation stated above.

We require some conventions for doing arithmetic with inﬁnities. We shall as-

sume that for any real number a ¤ 1, we have a C 1 D 1 C a D 1. Also, to

make our proofs hold in the presence of negative-weight cycles, we shall assume

that for any real number a ¤ 1, we have a C .1/ D .1/ C a D 1.

All algorithms in this chapter assume that the directed graph G is stored in the

adjacency-list representation. Additionally, stored with each edge is its weight, so

that as we traverse each adjacency list, we can determine the edge weights in O.1/

time per edge.

24.1 The Bellman-Ford algorithm 651

24.1 The Bellman-Ford algorithm

The Bellman-Ford algorithm solves the single-source shortest-paths problem in

the general case in which edge weights may be negative. Given a weighted, di-

rected graph G D .V; E/ with source s and weight function w W E ! R , the

Bellman-Ford algorithm returns a boolean value indicating whether or not there is

a negative-weight cycle that is reachable from the source. If there is such a cy-

cle, the algorithm indicates that no solution exists. If there is no such cycle, the

algorithm produces the shortest paths and their weights.

The algorithm relaxes edges, progressively decreasing an estimate :d on the

weight of a shortest path from the source s to each vertex 2 V until it achieves

the actual shortest-path weight ı.s; /. The algorithm returns TRUE if and only if

the graph contains no negative-weight cycles that are reachable from the source.

BELLMAN-FORD.G; w; s/

1 INITIALIZE-SINGLE-SOURCE.G; s/

2 for i D 1 to jG:Vj 1

3 for each edge .u; / 2 G:E

4 RELAX.u; ; w/

5 for each edge .u; / 2 G:E

6 if :d > u:d C w.u; /

7 return FALSE

8 return TRUE

Figure 24.4 shows the execution of the Bellman-Ford algorithm on a graph

with 5 vertices. After initializing the d and values of all vertices in line 1,

the algorithm makes jV j 1 passes over the edges of the graph. Each pass is

one iteration of the for loop of lines 2–4 and consists of relaxing each edge of the

graph once. Figures 24.4(b)–(e) show the state of the algorithm after each of the

four passes over the edges. After making jV j 1 passes, lines 5–8 check for a

negative-weight cycle and return the appropriate boolean value. (We’ll see a little

later why this check works.)

The Bellman-Ford algorithm runs in time O.VE/, since the initialization in

line 1 takes ‚.V / time, each of the jV j 1 passes over the edges in lines 2–4

takes ‚.E/ time, and the for loop of lines 5–7 takes O.E/ time.

To prove the correctness of the Bellman-Ford algorithm, we start by showing that

if there are no negative-weight cycles, the algorithm computes correct shortest-path

weights for all vertices reachable from the source.

652 Chapter 24 Single-Source Shortest Paths

(a) (b) (c)

(d)

0

5

9

7 8

6

7

(e)

t x

s

y z

–4

–3

–2 2

7

4

–2

2

0

5

9

7 8

6

7

t x

s

y z

–4

–3

–2 2

7

4

2

2

0

5

9

7 8

6

7

t x

s

y z

–4

–3

–2 6

7

4

2

2

0

5

9

7 8

6

7

t x

s

y z

–4

–3

–2 6

7

∞

∞

2

0

5

9

7 8

6

7

t x

s

y z

–4

–3

–2 ∞

∞

2

∞

∞

Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d val-

ues appear within the vertices, and shaded edges indicate predecessor values: if edge .u; / is

shaded, then : D u. In this particular example, each pass relaxes the edges in the order

.t; x/; .t; y/; .t; ´/; .x; t/; .y; x/; .y; ´/; .´; x/; .´; s/; .s; t/; .s; y/. (a) The situation just before the

ﬁrst pass over the edges. (b)–(e) The situation after each successive pass over the edges. The d

and values in part (e) are the ﬁnal values. The Bellman-Ford algorithm returns TRUE in this

example.

Lemma 24.2

Let G D .V; E/ be a weighted, directed graph with source s and weight func-

tion w W E ! R , and assume that G contains no negative-weight cycles that are

reachable from s. Then, after the jV j 1 iterations of the for loop of lines 2–4

of BELLMAN-FORD, we have :d D ı.s; / for all vertices that are reachable

from s.

Proof We prove the lemma by appealing to the path-relaxation property. Con-

sider any vertex that is reachable from s, and let p D h 0 ; 1 ; : : : ; k i, where

0 D s and k D , be any shortest path from s to . Because shortest paths are

simple, p has at most jV j 1 edges, and so k jV j 1. Each of the jV j 1 itera-

tions of the for loop of lines 2–4 relaxes all jEj edges. Among the edges relaxed in

the ith iteration, for i D 1; 2; : : : ; k, is . i1 ; i /. By the path-relaxation property,

therefore, :d D k :d D ı.s; k / D ı.s; /.

24.1 The Bellman-Ford algorithm 653

Corollary 24.3

Let G D .V; E/ be a weighted, directed graph with source vertex s and weight

function w W E ! R , and assume that G contains no negative-weight cycles that

are reachable from s. Then, for each vertex 2 V , there is a path from s to if

and only if BELLMAN-FORD terminates with :d < 1 when it is run on G.

Proof The proof is left as Exercise 24.1-2.

Theorem 24.4 (Correctness of the Bellman-Ford algorithm)

Let BELLMAN-FORD be run on a weighted, directed graph G D .V; E/ with

source s and weight function w W E ! R . If G contains no negative-weight cycles

that are reachable from s, then the algorithm returns TRUE, we have :d D ı.s; /

for all vertices 2 V , and the predecessor subgraph G is a shortest-paths tree

rooted at s. If G does contain a negative-weight cycle reachable from s, then the

algorithm returns FALSE.

Proof Suppose that graph G contains no negative-weight cycles that are reach-

able from the source s. We ﬁrst prove the claim that at termination, :d D ı.s; /

for all vertices 2 V . If vertex is reachable from s, then Lemma 24.2 proves this

claim. If is not reachable from s, then the claim follows from the no-path prop-

erty. Thus, the claim is proven. The predecessor-subgraph property, along with the

claim, implies that G is a shortest-paths tree. Now we use the claim to show that

BELLMAN-FORD returns TRUE. At termination, we have for all edges .u; / 2 E,

:d D ı.s; /

ı.s; u/ C w.u; / (by the triangle inequality)

D u:d C w.u; / ;

and so none of the tests in line 6 causes BELLMAN-FORD to return FALSE. There-

fore, it returns TRUE.

Now, suppose that graph G contains a negative-weight cycle that is reachable

from the source s; let this cycle be c D h 0 ; 1 ; : : : ; k i, where 0 D k . Then,

k X

iD1

w. i1 ; i / < 0 : (24.1)

Assume for the purpose of contradiction that the Bellman-Ford algorithm returns

TRUE. Thus, i :d i1 :d C w. i1 ; i / for i D 1; 2; : : : ; k. Summing the

inequalities around cycle c gives us

654 Chapter 24 Single-Source Shortest Paths

k X

iD1

i :d

k X

iD1

. i1 :d C w. i1 ; i //

D

k X

iD1

i1 :d C

k X

iD1

w. i1 ; i / :

Since 0 D k , each vertex in c appears exactly once in each of the summations

P k

iD1

i :d and

P k

iD1

i1 :d, and so

k X

iD1

i :d D

k X

iD1

i1 :d :

Moreover, by Corollary 24.3, i :d is ﬁnite for i D 1; 2; : : : ; k. Thus,

0

k X

iD1

w. i1 ; i / ;

which contradicts inequality (24.1). We conclude that the Bellman-Ford algorithm

returns TRUE if graph G contains no negative-weight cycles reachable from the

source, and FALSE otherwise.

Exercises

24.1-1

Run the Bellman-Ford algorithm on the directed graph of Figure 24.4, using ver-

tex ´ as the source. In each pass, relax edges in the same order as in the ﬁgure, and

show the d and values after each pass. Now, change the weight of edge .´; x/

to 4 and run the algorithm again, using s as the source.

24.1-2

Prove Corollary 24.3.

24.1-3

Given a weighted, directed graph G D .V; E/ with no negative-weight cycles,

let m be the maximum over all vertices 2 V of the minimum number of edges

in a shortest path from the source s to . (Here, the shortest path is by weight, not

the number of edges.) Suggest a simple change to the Bellman-Ford algorithm that

allows it to terminate in m C 1 passes, even if m is not known in advance.

24.1-4

Modify the Bellman-Ford algorithm so that it sets :d to 1 for all vertices for

which there is a negative-weight cycle on some path from the source to .

24.2 Single-source shortest paths in directed acyclic graphs 655

24.1-5 ?

Let G D .V; E/ be a weighted, directed graph with weight function w W E ! R .

Give an O.VE/-time algorithm to ﬁnd, for each vertex 2 V , the value ı ./ D

min u2V fı.u; /g.

24.1-6 ?

Suppose that a weighted, directed graph G D .V; E/ has a negative-weight cycle.

Give an efﬁcient algorithm to list the vertices of one such cycle. Prove that your

algorithm is correct.

24.2 Single-source shortest paths in directed acyclic graphs

By relaxing the edges of a weighted dag (directed acyclic graph) G D .V; E/

according to a topological sort of its vertices, we can compute shortest paths from

a single source in ‚.V C E/ time. Shortest paths are always well deﬁned in a dag,

since even if there are negative-weight edges, no negative-weight cycles can exist.

The algorithm starts by topologically sorting the dag (see Section 22.4) to im-

pose a linear ordering on the vertices. If the dag contains a path from vertex u to

vertex , then u precedes in the topological sort. We make just one pass over the

vertices in the topologically sorted order. As we process each vertex, we relax each

edge that leaves the vertex.

DAG-SHORTEST-PATHS.G; w; s/

1 topologically sort the vertices of G

2 INITIALIZE-SINGLE-SOURCE.G; s/

3 for each vertex u, taken in topologically sorted order

4 for each vertex 2 G:AdjŒu

5 RELAX.u; ; w/

Figure 24.5 shows the execution of this algorithm.

The running time of this algorithm is easy to analyze. As shown in Section 22.4,

the topological sort of line 1 takes ‚.V C E/ time. The call of INITIALIZE-

SINGLE-SOURCE in line 2 takes ‚.V / time. The for loop of lines 3–5 makes one

iteration per vertex. Altogether, the for loop of lines 4–5 relaxes each edge exactly

once. (We have used an aggregate analysis here.) Because each iteration of the

inner for loop takes ‚.1/ time, the total running time is ‚.V C E/, which is linear

in the size of an adjacency-list representation of the graph.

The following theorem shows that the DAG-SHORTEST-PATHS procedure cor-

rectly computes the shortest paths.

656 Chapter 24 Single-Source Shortest Paths

2

∞ ∞ 0

5

1 6

3 4

∞ ∞ ∞

7 –1 –2

2

(a)

x t s r y z

2 5

1 6

3 4

7 –1 –2

2

(c)

x t s r y z

2 5

1 6

3 4

7 –1 –2

2

(e)

x t s r y z

2 5

1 6

3 4

7 –1 –2

2

(g)

x t s r y z

2 5

1 6

3 4

7 –1 –2

2

(b)

x t s r y z

2 5

1 6

3 4

7 –1 –2

2

(d)

x t s r y z

2 5

1 6

3 4

7 –1 –2

2

(f)

x t s r y z

∞ 0 ∞ ∞ 2 6

∞ 0 2 6 5 4

∞ 0 2 6 5 3

∞ 0 2 6 5 3

∞ 0 2 6 6 4

∞ ∞ 0 ∞ ∞ ∞

Figure 24.5 The execution of the algorithm for shortest paths in a directed acyclic graph. The

vertices are topologically sorted from left to right. The source vertex is s. The d values appear

within the vertices, and shaded edges indicate the values. (a) The situation before the ﬁrst iteration

of the for loop of lines 3–5. (b)–(g) The situation after each iteration of the for loop of lines 3–5.

The newly blackened vertex in each iteration was used as u in that iteration. The values shown in

part (g) are the ﬁnal values.

Theorem 24.5

If a weighted, directed graph G D .V; E/ has source vertex s and no cycles, then

at the termination of the DAG-SHORTEST-PATHS procedure, :d D ı.s; / for all

vertices 2 V , and the predecessor subgraph G is a shortest-paths tree.

Proof We ﬁrst show that :d D ı.s; / for all vertices 2 V at termina-

tion. If is not reachable from s, then :d D ı.s; / D 1 by the no-path

property. Now, suppose that is reachable from s, so that there is a short-

est path p D h 0 ; 1 ; : : : ; k i, where 0 D s and k D . Because we pro-

24.2 Single-source shortest paths in directed acyclic graphs 657

cess the vertices in topologically sorted order, we relax the edges on p in the

order . 0 ; 1 /; . 1 ; 2 /; : : : ; . k1 ; k /. The path-relaxation property implies that

i :d D ı.s; i / at termination for i D 0; 1; : : : ; k. Finally, by the predecessor-

subgraph property, G is a shortest-paths tree.

An interesting application of this algorithm arises in determining critical paths

in PERT chart 2 analysis. Edges represent jobs to be performed, and edge weights

represent the times required to perform particular jobs. If edge .u; / enters ver-

tex and edge .; x/ leaves , then job .u; / must be performed before job .; x/.

A path through this dag represents a sequence of jobs that must be performed in a

particular order. A critical path is a longest path through the dag, corresponding

to the longest time to perform any sequence of jobs. Thus, the weight of a critical

path provides a lower bound on the total time to perform all the jobs. We can ﬁnd

a critical path by either

negating the edge weights and running DAG-SHORTEST-PATHS, or

running DAG-SHORTEST-PATHS, with the modiﬁcation that we replace “1”

by “1” in line 2 of INITIALIZE-SINGLE-SOURCE and “>” by “<” in the

RELAX procedure.

Exercises

24.2-1

Run DAG-SHORTEST-PATHS on the directed graph of Figure 24.5, using vertex r

as the source.

24.2-2

Suppose we change line 3 of DAG-SHORTEST-PATHS to read

3 for the ﬁrst jV j 1 vertices, taken in topologically sorted order

Show that the procedure would remain correct.

24.2-3

The PERT chart formulation given above is somewhat unnatural. In a more natu-

ral structure, vertices would represent jobs and edges would represent sequencing

constraints; that is, edge .u; / would indicate that job u must be performed before

job . We would then assign weights to vertices, not edges. Modify the DAG-

SHORTEST-PATHS procedure so that it ﬁnds a longest path in a directed acyclic

graph with weighted vertices in linear time.

2

“PERT” is an acronym for “program evaluation and review technique.”

658 Chapter 24 Single-Source Shortest Paths

24.2-4

Give an efﬁcient algorithm to count the total number of paths in a directed acyclic

graph. Analyze your algorithm.

24.3 Dijkstra’s algorithm

Dijkstra’s algorithm solves the single-source shortest-paths problem on a weighted,

directed graph G D .V; E/ for the case in which all edge weights are nonnegative.

In this section, therefore, we assume that w.u; / 0 for each edge .u; / 2 E. As

we shall see, with a good implementation, the running time of Dijkstra’s algorithm

is lower than that of the Bellman-Ford algorithm.

Dijkstra’s algorithm maintains a set S of vertices whose ﬁnal shortest-path

weights from the source s have already been determined. The algorithm repeat-

edly selects the vertex u 2 V S with the minimum shortest-path estimate, adds u

to S, and relaxes all edges leaving u. In the following implementation, we use a

min-priority queue Q of vertices, keyed by their d values.

DIJKSTRA.G; w; s/

1 INITIALIZE-SINGLE-SOURCE.G; s/

2 S D ;

3 Q D G:V

4 while Q ¤ ;

5 u D EXTRACT-MIN.Q/

6 S D S [ fug

7 for each vertex 2 G:AdjŒu

8 RELAX.u; ; w/

Dijkstra’s algorithm relaxes edges as shown in Figure 24.6. Line 1 initializes

the d and values in the usual way, and line 2 initializes the set S to the empty

set. The algorithm maintains the invariant that Q D V S at the start of each

iteration of the while loop of lines 4–8. Line 3 initializes the min-priority queue Q

to contain all the vertices in V ; since S D ; at that time, the invariant is true after

line 3. Each time through the while loop of lines 4–8, line 5 extracts a vertex u from

Q D V S and line 6 adds it to set S, thereby maintaining the invariant. (The ﬁrst

time through this loop, u D s.) Vertex u, therefore, has the smallest shortest-path

estimate of any vertex in V S. Then, lines 7–8 relax each edge .u; / leaving u,

thus updating the estimate :d and the predecessor : if we can improve the

shortest path to found so far by going through u. Observe that the algorithm

never inserts vertices into Q after line 3 and that each vertex is extracted from Q

24.3 Dijkstra’s algorithm 659

0

∞ ∞

∞ ∞

0

∞

∞

1

2

10

5

(c)

10

5

0

8

5

14

7

0

8

5

13

7

0

8

5

9

7

0

5

9

7

8

6 4 3 2

9

7

s

t x

y z

1

2

10

5

(f)

6 4 3 2

9

7

s

t x

y z

1

2

10

5

(b)

6 4 3 2

9

7

s

t x

y z

1

2

10

5

(e)

6 4 3 2

9

7

s

t x

y z

1

2

10

5

(a)

6 4 3 2

9

7

s

t x

y z

1

2

10

5

(d)

6 4 3 2

9

7

s

t x

y z

Figure 24.6 The execution of Dijkstra’s algorithm. The source s is the leftmost vertex. The

shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values.

Black vertices are in the set S, and white vertices are in the min-priority queue Q D V S. (a) The

situation just before the ﬁrst iteration of the while loop of lines 4–8. The shaded vertex has the mini-

mum d value and is chosen as vertex u in line 5. (b)–(f) The situation after each successive iteration

of the while loop. The shaded vertex in each part is chosen as vertex u in line 5 of the next iteration.

The d values and predecessors shown in part (f) are the ﬁnal values.

and added to S exactly once, so that the while loop of lines 4–8 iterates exactly jV j

times.

Because Dijkstra’s algorithm always chooses the “lightest” or “closest” vertex

in V S to add to set S, we say that it uses a greedy strategy. Chapter 16 explains

greedy strategies in detail, but you need not have read that chapter to understand

Dijkstra’s algorithm. Greedy strategies do not always yield optimal results in gen-

eral, but as the following theorem and its corollary show, Dijkstra’s algorithm does

indeed compute shortest paths. The key is to show that each time it adds a vertex u

to set S, we have u:d D ı.s; u/.

Theorem 24.6 (Correctness of Dijkstra’s algorithm)

Dijkstra’s algorithm, run on a weighted, directed graph G D .V; E/ with non-

negative weight function w and source s, terminates with u:d D ı.s; u/ for all

vertices u 2 V .

660 Chapter 24 Single-Source Shortest Paths

p

1

S

p

2

u

y

s

x

Figure 24.7 The proof of Theorem 24.6. Set S is nonempty just before vertex u is added to it. We

decompose a shortest path p from source s to vertex u into s

p 1 x ! y

p 2 u, where y is the ﬁrst

vertex on the path that is not in S and x 2 S immediately precedes y. Vertices x and y are distinct,

but we may have s D x or y D u. Path p2 may or may not reenter set S.

Proof We use the following loop invariant:

At the start of each iteration of the while loop of lines 4–8, :d D ı.s; /

for each vertex 2 S.

It sufﬁces to show for each vertex u 2 V , we have u:d D ı.s; u/ at the time when u

is added to set S. Once we show that u:d D ı.s; u/, we rely on the upper-bound

property to show that the equality holds at all times thereafter.

Initialization: Initially, S D ;, and so the invariant is trivially true.

Maintenance: We wish to show that in each iteration, u:d D ı.s; u/ for the vertex

added to set S. For the purpose of contradiction, let u be the ﬁrst vertex for

which u:d ¤ ı.s; u/ when it is added to set S. We shall focus our attention

on the situation at the beginning of the iteration of the while loop in which u

is added to S and derive the contradiction that u:d D ı.s; u/ at that time by

examining a shortest path from s to u. We must have u ¤ s because s is the

ﬁrst vertex added to set S and s:d D ı.s; s/ D 0 at that time. Because u ¤ s,

we also have that S ¤ ; just before u is added to S. There must be some

path from s to u, for otherwise u:d D ı.s; u/ D 1 by the no-path property,

which would violate our assumption that u:d ¤ ı.s; u/. Because there is at

least one path, there is a shortest path p from s to u. Prior to adding u to S,

path p connects a vertex in S, namely s, to a vertex in V S, namely u. Let us

consider the ﬁrst vertex y along p such that y 2 V S, and let x 2 S be y’s

predecessor along p. Thus, as Figure 24.7 illustrates, we can decompose path p

into s

p 1 x ! y

p 2 u. (Either of paths p 1 or p 2 may have no edges.)

We claim that y:d D ı.s; y/ when u is added to S. To prove this claim, ob-

serve that x 2 S. Then, because we chose u as the ﬁrst vertex for which

u:d ¤ ı.s; u/ when it is added to S, we had x:d D ı.s; x/ when x was added

24.3 Dijkstra’s algorithm 661

to S. Edge .x; y/ was relaxed at that time, and the claim follows from the

convergence property.

We can now obtain a contradiction to prove that u:d D ı.s; u/. Because y

appears before u on a shortest path from s to u and all edge weights are non-

negative (notably those on path p 2 ), we have ı.s; y/ ı.s; u/, and thus

y:d D ı.s; y/

ı.s; u/ (24.2)

u:d (by the upper-bound property) .

But because both vertices u and y were in V S when u was chosen in line 5,

we have u:d y:d. Thus, the two inequalities in (24.2) are in fact equalities,

giving

y:d D ı.s; y/ D ı.s; u/ D u:d :

Consequently, u:d D ı.s; u/, which contradicts our choice of u. We conclude

that u:d D ı.s; u/ when u is added to S, and that this equality is maintained at

all times thereafter.

Termination: At termination, Q D ; which, along with our earlier invariant that

Q D V S, implies that S D V . Thus, u:d D ı.s; u/ for all vertices u 2 V .

Corollary 24.7

If we run Dijkstra’s algorithm on a weighted, directed graph G D .V; E/ with

nonnegative weight function w and source s, then at termination, the predecessor

subgraph G is a shortest-paths tree rooted at s.

Proof Immediate from Theorem 24.6 and the predecessor-subgraph property.

Analysis

How fast is Dijkstra’s algorithm? It maintains the min-priority queue Q by call-

ing three priority-queue operations: INSERT (implicit in line 3), EXTRACT-MIN

(line 5), and DECREASE-KEY (implicit in RELAX, which is called in line 8). The

algorithm calls both INSERT and EXTRACT-MIN once per vertex. Because each

vertex u 2 V is added to set S exactly once, each edge in the adjacency list AdjŒu

is examined in the for loop of lines 7–8 exactly once during the course of the al-

gorithm. Since the total number of edges in all the adjacency lists is jEj, this for

loop iterates a total of jEj times, and thus the algorithm calls DECREASE-KEY at

most jEj times overall. (Observe once again that we are using aggregate analysis.)

The running time of Dijkstra’s algorithm depends on how we implement the

min-priority queue. Consider ﬁrst the case in which we maintain the min-priority

662 Chapter 24 Single-Source Shortest Paths

queue by taking advantage of the vertices being numbered 1 to jV j. We simply

store :d in the th entry of an array. Each INSERT and DECREASE-KEY operation

takes O.1/ time, and each EXTRACT-MIN operation takes O.V / time (since we

have to search through the entire array), for a total time of O.V 2 C E/ D O.V 2 /.

If the graph is sufﬁciently sparse—in particular, E D o.V 2 = lg V /—we can

improve the algorithm by implementing the min-priority queue with a binary min-

heap. (As discussed in Section 6.5, the implementation should make sure that

vertices and corresponding heap elements maintain handles to each other.) Each

EXTRACT-MIN operation then takes time O.lg V /. As before, there are jV j such

operations. The time to build the binary min-heap is O.V /. Each DECREASE-KEY

operation takes time O.lg V /, and there are still at most jEj such operations. The

total running time is therefore O..V C E/ lg V /, which is O.E lg V / if all vertices

are reachable from the source. This running time improves upon the straightfor-

ward O.V 2 /-time implementation if E D o.V 2 = lg V /.

We can in fact achieve a running time of O.V lg V C E/ by implementing the

min-priority queue with a Fibonacci heap (see Chapter 19). The amortized cost

of each of the jV j EXTRACT-MIN operations is O.lg V /, and each DECREASE-

KEY call, of which there are at most jEj, takes only O.1/ amortized time. His-

torically, the development of Fibonacci heaps was motivated by the observation

that Dijkstra’s algorithm typically makes many more DECREASE-KEY calls than

EXTRACT-MIN calls, so that any method of reducing the amortized time of each

DECREASE-KEY operation to o.lg V / without increasing the amortized time of

EXTRACT-MIN would yield an asymptotically faster implementation than with bi-

nary heaps.

Dijkstra’s algorithm resembles both breadth-ﬁrst search (see Section 22.2) and

Prim’s algorithm for computing minimum spanning trees (see Section 23.2). It is

like breadth-ﬁrst search in that set S corresponds to the set of black vertices in a

breadth-ﬁrst search; just as vertices in S have their ﬁnal shortest-path weights, so

do black vertices in a breadth-ﬁrst search have their correct breadth-ﬁrst distances.

Dijkstra’s algorithm is like Prim’s algorithm in that both algorithms use a min-

priority queue to ﬁnd the “lightest” vertex outside a given set (the set S in Dijkstra’s

algorithm and the tree being grown in Prim’s algorithm), add this vertex into the

set, and adjust the weights of the remaining vertices outside the set accordingly.

Exercises

24.3-1

Run Dijkstra’s algorithm on the directed graph of Figure 24.2, ﬁrst using vertex s

as the source and then using vertex ´ as the source. In the style of Figure 24.6,

show the d and values and the vertices in set S after each iteration of the while

loop.

24.3 Dijkstra’s algorithm 663

24.3-2

Give a simple example of a directed graph with negative-weight edges for which

Dijkstra’s algorithm produces incorrect answers. Why doesn’t the proof of Theo-

rem 24.6 go through when negative-weight edges are allowed?

24.3-3

Suppose we change line 4 of Dijkstra’s algorithm to the following.

4 while jQj > 1

This change causes the while loop to execute jV j 1 times instead of jV j times. Is

this proposed algorithm correct?

24.3-4

Professor Gaedel has written a program that he claims implements Dijkstra’s al-

gorithm. The program produces :d and : for each vertex 2 V . Give an

O.V CE/-time algorithm to check the output of the professor’s program. It should

determine whether the d and attributes match those of some shortest-paths tree.

You may assume that all edge weights are nonnegative.

24.3-5

Professor Newman thinks that he has worked out a simpler proof of correctness

for Dijkstra’s algorithm. He claims that Dijkstra’s algorithm relaxes the edges of

every shortest path in the graph in the order in which they appear on the path, and

therefore the path-relaxation property applies to every vertex reachable from the

source. Show that the professor is mistaken by constructing a directed graph for

which Dijkstra’s algorithm could relax the edges of a shortest path out of order.

24.3-6

We are given a directed graph G D .V; E/ on which each edge .u; / 2 E has an

associated value r.u; /, which is a real number in the range 0 r.u; / 1 that

represents the reliability of a communication channel from vertex u to vertex .

We interpret r.u; / as the probability that the channel from u to will not fail,

and we assume that these probabilities are independent. Give an efﬁcient algorithm

to ﬁnd the most reliable path between two given vertices.

24.3-7

Let G D .V; E/ be a weighted, directed graph with positive weight function

w W E ! f1; 2; : : : ; W g for some positive integer W , and assume that no two ver-

tices have the same shortest-path weights from source vertex s. Now suppose that

we deﬁne an unweighted, directed graph G 0 D .V [ V 0 ; E 0 / by replacing each

edge .u; / 2 E with w.u; / unit-weight edges in series. How many vertices

does G 0

have? Now suppose that we run a breadth-ﬁrst search on G 0

. Show that

664 Chapter 24 Single-Source Shortest Paths

the order in which the breadth-ﬁrst search of G 0

colors vertices in V black is the

same as the order in which Dijkstra’s algorithm extracts the vertices of V from the

priority queue when it runs on G.

24.3-8

Let G D .V; E/ be a weighted, directed graph with nonnegative weight function

w W E ! f0; 1; : : : ; W g for some nonnegative integer W . Modify Dijkstra’s algo-

rithm to compute the shortest paths from a given source vertex s in O.W V C E/

time.

24.3-9

Modify your algorithm from Exercise 24.3-8 to run in O..V C E/ lg W / time.

(Hint: How many distinct shortest-path estimates can there be in V S at any

point in time?)

24.3-10

Suppose that we are given a weighted, directed graph G D .V; E/ in which edges

that leave the source vertex s may have negative weights, all other edge weights

are nonnegative, and there are no negative-weight cycles. Argue that Dijkstra’s

algorithm correctly ﬁnds shortest paths from s in this graph.

24.4 Difference constraints and shortest paths

Chapter 29 studies the general linear-programming problem, in which we wish to

optimize a linear function subject to a set of linear inequalities. In this section, we

investigate a special case of linear programming that we reduce to ﬁnding shortest

paths from a single source. We can then solve the single-source shortest-paths

problem that results by running the Bellman-Ford algorithm, thereby also solving

the linear-programming problem.

Linear programming

In the general linear-programming problem, we are given an m n matrix A,

an m-vector b, and an n-vector c. We wish to ﬁnd a vector x of n elements that

maximizes the objective function

P n

iD1

c i x i subject to the m constraints given by

Ax b.

Although the simplex algorithm, which is the focus of Chapter 29, does not

always run in time polynomial in the size of its input, there are other linear-

programming algorithms that do run in polynomial time. We offer here two reasons

to understand the setup of linear-programming problems. First, if we know that we

24.4 Difference constraints and shortest paths 665

can cast a given problem as a polynomial-sized linear-programming problem, then

we immediately have a polynomial-time algorithm to solve the problem. Second,

faster algorithms exist for many special cases of linear programming. For exam-

ple, the single-pair shortest-path problem (Exercise 24.4-4) and the maximum-ﬂow

problem (Exercise 26.1-5) are special cases of linear programming.

Sometimes we don’t really care about the objective function; we just wish to ﬁnd

any feasible solution, that is, any vector x that satisﬁes Ax b, or to determine

that no feasible solution exists. We shall focus on one such feasibility problem.

Systems of difference constraints

In a system of difference constraints, each row of the linear-programming matrix A

contains one 1 and one 1, and all other entries of A are 0. Thus, the constraints

given by Ax b are a set of m difference constraints involving n unknowns, in

which each constraint is a simple linear inequality of the form

x j x i b k ;

where 1 i; j n, i ¤ j , and 1 k m.

For example, consider the problem of ﬁnding a 5-vector x D .x i / that satisﬁes 1 1 0 0 0

1 0 0 0 1

0 1 0 0 1

1 0 1 0 0

1 0 0 1 0

0 0 1 1 0

0 0 1 0 1

0 0 0 1 1

˘

ˇ

x 1

x 2

x 3

x 4

x 5

0

1

1

5

4

1

3

3

˘

:

This problem is equivalent to ﬁnding values for the unknowns x 1 ; x 2 ; x 3 ; x 4 ; x 5 ,

satisfying the following 8 difference constraints:

x 1 x 2 0 , (24.3)

x 1 x 5 1 , (24.4)

x 2 x 5 1 , (24.5)

x 3 x 1 5 , (24.6)

x 4 x 1 4 , (24.7)

x 4 x 3 1 , (24.8)

x 5 x 3 3 , (24.9)

x 5 x 4 3 . (24.10)

666 Chapter 24 Single-Source Shortest Paths

One solution to this problem is x D .5; 3; 0; 1; 4/, which you can verify di-

rectly by checking each inequality. In fact, this problem has more than one solution.

Another is x 0 D .0; 2; 5; 4; 1/. These two solutions are related: each component

of x 0

is 5 larger than the corresponding component of x. This fact is not mere

coincidence.

Lemma 24.8

Let x D .x 1 ; x 2 ; : : : ; x n / be a solution to a system Ax b of difference con-

straints, and let d be any constant. Then x C d D .x 1 C d; x 2 C d; : : : ; x n C d/

is a solution to Ax b as well.

Proof For each x i and x j , we have .x j C d/ .x i C d/ D x j x i . Thus, if x

satisﬁes Ax b, so does x C d.

Systems of difference constraints occur in many different applications. For ex-

ample, the unknowns x i may be times at which events are to occur. Each constraint

states that at least a certain amount of time, or at most a certain amount of time,

must elapse between two events. Perhaps the events are jobs to be performed dur-

ing the assembly of a product. If we apply an adhesive that takes 2 hours to set at

time x 1 and we have to wait until it sets to install a part at time x 2 , then we have the

constraint that x 2 x 1 C 2 or, equivalently, that x 1 x 2 2. Alternatively, we

might require that the part be installed after the adhesive has been applied but no

later than the time that the adhesive has set halfway. In this case, we get the pair of

constraints x 2 x 1 and x 2 x 1 C1 or, equivalently, x 1 x 2 0 and x 2 x 1 1.

Constraint graphs

We can interpret systems of difference constraints from a graph-theoretic point

of view. In a system Ax b of difference constraints, we view the m n

linear-programming matrix A as the transpose of an incidence matrix (see Exer-

cise 22.1-7) for a graph with n vertices and m edges. Each vertex i in the graph,

for i D 1; 2; : : : ; n, corresponds to one of the n unknown variables x i . Each di-

rected edge in the graph corresponds to one of the m inequalities involving two

unknowns.

More formally, given a system Ax b of difference constraints, the correspond-

ing constraint graph is a weighted, directed graph G D .V; E/, where

V D f 0 ; 1 ; : : : ; n g

and

E D f. i ; j / W x j x i b k is a constraintg

[ f. 0 ; 1 /; . 0 ; 2 /; . 0 ; 3 /; : : : ; . 0 ; n /g :

24.4 Difference constraints and shortest paths 667

0

0

0

0

0

0 –1

1

5

4

–1

–3 –3

0

–5

–3

0 –1

–4

v

3

v

2

v

1

v

5

v

0

v

4

Figure 24.8 The constraint graph corresponding to the system (24.3)–(24.10) of difference con-

straints. The value of ı.0; i / appears in each vertex i . One feasible solution to the system is

x D .5; 3; 0; 1; 4/.

The constraint graph contains the additional vertex 0 , as we shall see shortly, to

guarantee that the graph has some vertex which can reach all other vertices. Thus,

the vertex set V consists of a vertex i for each unknown x i , plus an additional

vertex 0 . The edge set E contains an edge for each difference constraint, plus

an edge . 0 ; i / for each unknown x i . If x j x i b k is a difference constraint,

then the weight of edge . i ; j / is w. i ; j / D b k . The weight of each edge leav-

ing 0 is 0. Figure 24.8 shows the constraint graph for the system (24.3)–(24.10)

of difference constraints.

The following theorem shows that we can ﬁnd a solution to a system of differ-

ence constraints by ﬁnding shortest-path weights in the corresponding constraint

graph.

Theorem 24.9

Given a system Ax b of difference constraints, let G D .V; E/ be the corre-

sponding constraint graph. If G contains no negative-weight cycles, then

x D .ı. 0 ; 1 /; ı. 0 ; 2 /; ı. 0 ; 3 /; : : : ; ı. 0 ; n // (24.11)

is a feasible solution for the system. If G contains a negative-weight cycle, then

there is no feasible solution for the system.

Proof We ﬁrst show that if the constraint graph contains no negative-weight

cycles, then equation (24.11) gives a feasible solution. Consider any edge

. i ; j / 2 E. By the triangle inequality, ı. 0 ; j / ı. 0 ; i / C w. i ; j / or,

equivalently, ı. 0 ; j / ı. 0 ; i / w. i ; j /. Thus, letting x i D ı. 0 ; i / and

668 Chapter 24 Single-Source Shortest Paths

x j D ı. 0 ; j / satisﬁes the difference constraint x j x i w. i ; j / that corre-

sponds to edge . i ; j /.

Now we show that if the constraint graph contains a negative-weight cycle, then

the system of difference constraints has no feasible solution. Without loss of gen-

erality, let the negative-weight cycle be c D h 1 ; 2 ; : : : ; k i, where 1 D k .

(The vertex 0 cannot be on cycle c, because it has no entering edges.) Cycle c

corresponds to the following difference constraints:

x 2 x 1 w. 1 ; 2 / ;

x 3 x 2 w. 2 ; 3 / ;

:

:

:

x k1 x k2 w. k2 ; k1 / ;

x k x k1 w. k1 ; k / :

We will assume that x has a solution satisfying each of these k inequalities and then

derive a contradiction. The solution must also satisfy the inequality that results

when we sum the k inequalities together. If we sum the left-hand sides, each

unknown x i is added in once and subtracted out once (remember that 1 D k

implies x 1 D x k ), so that the left-hand side of the sum is 0. The right-hand side

sums to w.c/, and thus we obtain 0 w.c/. But since c is a negative-weight cycle,

w.c/ < 0, and we obtain the contradiction that 0 w.c/ < 0.

Solving systems of difference constraints

Theorem 24.9 tells us that we can use the Bellman-Ford algorithm to solve a

system of difference constraints. Because the constraint graph contains edges

from the source vertex 0 to all other vertices, any negative-weight cycle in the

constraint graph is reachable from 0 . If the Bellman-Ford algorithm returns

TRUE, then the shortest-path weights give a feasible solution to the system. In

Figure 24.8, for example, the shortest-path weights provide the feasible solution

x D .5; 3; 0; 1; 4/, and by Lemma 24.8, x D .d 5; d 3; d; d 1; d 4/

is also a feasible solution for any constant d. If the Bellman-Ford algorithm returns

FALSE, there is no feasible solution to the system of difference constraints.

A system of difference constraints with m constraints on n unknowns produces

a graph with n C 1 vertices and n C m edges. Thus, using the Bellman-Ford

algorithm, we can solve the system in O..n C 1/.n C m// D O.n 2 C nm/ time.

Exercise 24.4-5 asks you to modify the algorithm to run in O.nm/ time, even if m

is much less than n.

24.4 Difference constraints and shortest paths 669

Exercises

24.4-1

Find a feasible solution or determine that no feasible solution exists for the follow-

ing system of difference constraints:

x 1 x 2 1 ,

x 1 x 4 4 ,

x 2 x 3 2 ,

x 2 x 5 7 ,

x 2 x 6 5 ,

x 3 x 6 10 ,

x 4 x 2 2 ,

x 5 x 1 1 ,

x 5 x 4 3 ,

x 6 x 3 8 .

24.4-2

Find a feasible solution or determine that no feasible solution exists for the follow-

ing system of difference constraints:

x 1 x 2 4 ,

x 1 x 5 5 ,

x 2 x 4 6 ,

x 3 x 2 1 ,

x 4 x 1 3 ,

x 4 x 3 5 ,

x 4 x 5 10 ,

x 5 x 3 4 ,

x 5 x 4 8 .

24.4-3

Can any shortest-path weight from the new vertex 0 in a constraint graph be posi-

tive? Explain.

24.4-4

Express the single-pair shortest-path problem as a linear program.

670 Chapter 24 Single-Source Shortest Paths

24.4-5

Show how to modify the Bellman-Ford algorithm slightly so that when we use it

to solve a system of difference constraints with m inequalities on n unknowns, the

running time is O.nm/.

24.4-6

Suppose that in addition to a system of difference constraints, we want to handle

equality constraints of the form x i D x j C b k . Show how to adapt the Bellman-

Ford algorithm to solve this variety of constraint system.

24.4-7

Show how to solve a system of difference constraints by a Bellman-Ford-like algo-

rithm that runs on a constraint graph without the extra vertex 0 .

24.4-8 ?

Let Ax b be a system of m difference constraints in n unknowns. Show that the

Bellman-Ford algorithm, when run on the corresponding constraint graph, maxi-

mizes

P n

iD1

x i subject to Ax b and x i 0 for all x i .

24.4-9 ?

Show that the Bellman-Ford algorithm, when run on the constraint graph for a sys-

tem Ax b of difference constraints, minimizes the quantity .max fx i gmin fx i g/

subject to Ax b. Explain how this fact might come in handy if the algorithm is

used to schedule construction jobs.

24.4-10

Suppose that every row in the matrix A of a linear program Ax b corresponds to

a difference constraint, a single-variable constraint of the form x i b k , or a single-

variable constraint of the form x i b k . Show how to adapt the Bellman-Ford

algorithm to solve this variety of constraint system.

24.4-11

Give an efﬁcient algorithm to solve a system Ax b of difference constraints

when all of the elements of b are real-valued and all of the unknowns x i must be

integers.

24.4-12 ?

Give an efﬁcient algorithm to solve a system Ax b of difference constraints

when all of the elements of b are real-valued and a speciﬁed subset of some, but

not necessarily all, of the unknowns x i must be integers.

24.5 Proofs of shortest-paths properties 671

24.5 Proofs of shortest-paths properties

Throughout this chapter, our correctness arguments have relied on the triangle

inequality, upper-bound property, no-path property, convergence property, path-

relaxation property, and predecessor-subgraph property. We stated these properties

without proof at the beginning of this chapter. In this section, we prove them.

The triangle inequality

In studying breadth-ﬁrst search (Section 22.2), we proved as Lemma 22.1 a sim-

ple property of shortest distances in unweighted graphs. The triangle inequality

generalizes the property to weighted graphs.

Lemma 24.10 (Triangle inequality)

Let G D .V; E/ be a weighted, directed graph with weight function w W E ! R

and source vertex s. Then, for all edges .u; / 2 E, we have

ı.s; / ı.s; u/ C w.u; / :

Proof Suppose that p is a shortest path from source s to vertex . Then p has

no more weight than any other path from s to . Speciﬁcally, path p has no more

weight than the particular path that takes a shortest path from source s to vertex u

and then takes edge .u; /.

Exercise 24.5-3 asks you to handle the case in which there is no shortest path

from s to .

Effects of relaxation on shortest-path estimates

The next group of lemmas describes how shortest-path estimates are affected when

we execute a sequence of relaxation steps on the edges of a weighted, directed

graph that has been initialized by INITIALIZE-SINGLE-SOURCE.

Lemma 24.11 (Upper-bound property)

Let G D .V; E/ be a weighted, directed graph with weight function w W E ! R .

Let s 2 V be the source vertex, and let the graph be initialized by INITIALIZE-

SINGLE-SOURCE.G; s/. Then, :d ı.s; / for all 2 V , and this invariant is

maintained over any sequence of relaxation steps on the edges of G. Moreover,

once :d achieves its lower bound ı.s; /, it never changes.

672 Chapter 24 Single-Source Shortest Paths

Proof We prove the invariant :d ı.s; / for all vertices 2 V by induction

over the number of relaxation steps.

For the basis, :d ı.s; / is certainly true after initialization, since :d D 1

implies :d ı.s; / for all 2 V fsg, and since s:d D 0 ı.s; s/ (note that

ı.s; s/ D 1 if s is on a negative-weight cycle and 0 otherwise).

For the inductive step, consider the relaxation of an edge .u; /. By the inductive

hypothesis, x:d ı.s; x/ for all x 2 V prior to the relaxation. The only d value

that may change is :d. If it changes, we have

:d D u:d C w.u; /

ı.s; u/ C w.u; / (by the inductive hypothesis)

ı.s; / (by the triangle inequality) ,

and so the invariant is maintained.

To see that the value of :d never changes once :d D ı.s; /, note that having

achieved its lower bound, :d cannot decrease because we have just shown that

:d ı.s; /, and it cannot increase because relaxation steps do not increase d

values.

Corollary 24.12 (No-path property)

Suppose that in a weighted, directed graph G D .V; E/ with weight function

w W E ! R , no path connects a source vertex s 2 V to a given vertex 2 V .

Then, after the graph is initialized by INITIALIZE-SINGLE-SOURCE.G; s/, we

have :d D ı.s; / D 1, and this equality is maintained as an invariant over

any sequence of relaxation steps on the edges of G.

Proof By the upper-bound property, we always have 1 D ı.s; / :d, and

thus :d D 1 D ı.s; /.

Lemma 24.13

Let G D .V; E/ be a weighted, directed graph with weight function w W E ! R ,

and let .u; / 2 E. Then, immediately after relaxing edge .u; / by executing

RELAX.u; ; w/, we have :d u:d C w.u; /.

Proof If, just prior to relaxing edge .u; /, we have :d > u:d C w.u; /, then

:d D u:d C w.u; / afterward. If, instead, :d u:d C w.u; / just before

the relaxation, then neither u:d nor :d changes, and so :d u:d C w.u; /

afterward.

Lemma 24.14 (Convergence property)

Let G D .V; E/ be a weighted, directed graph with weight function w W E ! R ,

let s 2 V be a source vertex, and let s u ! be a shortest path in G for

24.5 Proofs of shortest-paths properties 673

some vertices u; 2 V . Suppose that G is initialized by INITIALIZE-SINGLE-

SOURCE.G; s/ and then a sequence of relaxation steps that includes the call

RELAX.u; ; w/ is executed on the edges of G. If u:d D ı.s; u/ at any time

prior to the call, then :d D ı.s; / at all times after the call.

Proof By the upper-bound property, if u:d D ı.s; u/ at some point prior to re-

laxing edge .u; /, then this equality holds thereafter. In particular, after relaxing

edge .u; /, we have

:d u:d C w.u; / (by Lemma 24.13)

D ı.s; u/ C w.u; /

D ı.s; / (by Lemma 24.1) .

By the upper-bound property, :d ı.s; /, from which we conclude that

:d D ı.s; /, and this equality is maintained thereafter.

Lemma 24.15 (Path-relaxation property)

Let G D .V; E/ be a weighted, directed graph with weight function w W E ! R ,

and let s 2 V be a source vertex. Consider any shortest path p D h 0 ; 1 ; : : : ; k i

from s D 0 to k . If G is initialized by INITIALIZE-SINGLE-SOURCE.G; s/ and

then a sequence of relaxation steps occurs that includes, in order, relaxing the edges

. 0 ; 1 /; . 1 ; 2 /; : : : ; . k1 ; k /, then k :d D ı.s; k / after these relaxations and

at all times afterward. This property holds no matter what other edge relaxations

occur, including relaxations that are intermixed with relaxations of the edges of p.

Proof We show by induction that after the ith edge of path p is relaxed, we have

i :d D ı.s; i /. For the basis, i D 0, and before any edges of p have been relaxed,

we have from the initialization that 0 :d D s:d D 0 D ı.s; s/. By the upper-bound

property, the value of s:d never changes after initialization.

For the inductive step, we assume that i1 :d D ı.s; i1 /, and we examine

what happens when we relax edge . i1 ; i /. By the convergence property, after

relaxing this edge, we have i :d D ı.s; i /, and this equality is maintained at all

times thereafter.

Relaxation and shortest-paths trees

We now show that once a sequence of relaxations has caused the shortest-path es-

timates to converge to shortest-path weights, the predecessor subgraph G induced

by the resulting values is a shortest-paths tree for G. We start with the follow-

ing lemma, which shows that the predecessor subgraph always forms a rooted tree

whose root is the source.

674 Chapter 24 Single-Source Shortest Paths

Lemma 24.16

Let G D .V; E/ be a weighted, directed graph with weight function w W E ! R ,

let s 2 V be a source vertex, and assume that G contains no negative-weight

cycles that are reachable from s. Then, after the graph is initialized by INITIALIZE-

SINGLE-SOURCE.G; s/, the predecessor subgraph G forms a rooted tree with

root s, and any sequence of relaxation steps on edges of G maintains this property

as an invariant.

Proof Initially, the only vertex in G is the source vertex, and the lemma is triv-

ially true. Consider a predecessor subgraph G that arises after a sequence of

relaxation steps. We shall ﬁrst prove that G is acyclic. Suppose for the sake of

contradiction that some relaxation step creates a cycle in the graph G . Let the cy-

cle be c D h 0 ; 1 ; : : : ; k i, where k D 0 . Then, i : D i1 for i D 1; 2; : : : ; k

and, without loss of generality, we can assume that relaxing edge . k1 ; k / created

the cycle in G .

We claim that all vertices on cycle c are reachable from the source s. Why?

Each vertex on c has a non-NIL predecessor, and so each vertex on c was assigned

a ﬁnite shortest-path estimate when it was assigned its non-NIL value. By the

upper-bound property, each vertex on cycle c has a ﬁnite shortest-path weight,

which implies that it is reachable from s.

We shall examine the shortest-path estimates on c just prior to the call

RELAX. k1 ; k ; w/ and show that c is a negative-weight cycle, thereby contra-

dicting the assumption that G contains no negative-weight cycles that are reachable

from the source. Just before the call, we have i : D i1 for i D 1; 2; : : : ; k 1.

Thus, for i D 1; 2; : : : ; k 1, the last update to i :d was by the assignment

i :d D i1 :dCw. i1 ; i /. If i1 :d changed since then, it decreased. Therefore,

just before the call RELAX. k1 ; k ; w/, we have

i :d i1 :d C w. i1 ; i / for all i D 1; 2; : : : ; k 1 : (24.12)

Because k : is changed by the call, immediately beforehand we also have the

strict inequality

k :d > k1 :d C w. k1 ; k / :

Summing this strict inequality with the k 1 inequalities (24.12), we obtain the

sum of the shortest-path estimates around cycle c:

k X

iD1

i :d >

k X

iD1

. i1 :d C w. i1 ; i //

D

k X

iD1

i1 :d C

k X

iD1

w. i1 ; i / :

24.5 Proofs of shortest-paths properties 675

s

u

x

y

z

v

Figure 24.9 Showing that a simple path in G from source s to vertex is unique. If there are two

paths p1 (s u x ! ´ ) and p2 (s u y ! ´ ), where x ¤ y, then ´: D x

and ´: D y, a contradiction.

But

k X

iD1

i :d D

k X

iD1

i1 :d ;

since each vertex in the cycle c appears exactly once in each summation. This

equality implies

0 >

k X

iD1

w. i1 ; i / :

Thus, the sum of weights around the cycle c is negative, which provides the desired

contradiction.

We have now proven that G is a directed, acyclic graph. To show that it forms

a rooted tree with root s, it sufﬁces (see Exercise B.5-2) to prove that for each

vertex 2 V , there is a unique simple path from s to in G .

We ﬁrst must show that a path from s exists for each vertex in V . The ver-

tices in V are those with non-NIL values, plus s. The idea here is to prove by

induction that a path exists from s to all vertices in V . We leave the details as

Exercise 24.5-6.

To complete the proof of the lemma, we must now show that for any vertex

2 V , the graph G contains at most one simple path from s to . Suppose other-

wise. That is, suppose that, as Figure 24.9 illustrates, G contains two simple paths

from s to some vertex : p 1 , which we decompose into s u x ! ´ ,

and p 2 , which we decompose into s u y ! ´ , where x ¤ y (though u

could be s and ´ could be ). But then, ´: D x and ´: D y, which implies

the contradiction that x D y. We conclude that G contains a unique simple path

from s to , and thus G forms a rooted tree with root s.

We can now show that if, after we have performed a sequence of relaxation steps,

all vertices have been assigned their true shortest-path weights, then the predeces-

sor subgraph G is a shortest-paths tree.

676 Chapter 24 Single-Source Shortest Paths

Lemma 24.17 (Predecessor-subgraph property)

Let G D .V; E/ be a weighted, directed graph with weight function w W E ! R ,

let s 2 V be a source vertex, and assume that G contains no negative-weight cycles

that are reachable from s. Let us call INITIALIZE-SINGLE-SOURCE.G; s/ and then

execute any sequence of relaxation steps on edges of G that produces :d D ı.s; /

for all 2 V . Then, the predecessor subgraph G is a shortest-paths tree rooted

at s.

Proof We must prove that the three properties of shortest-paths trees given on

page 647 hold for G . To show the ﬁrst property, we must show that V is the set

of vertices reachable from s. By deﬁnition, a shortest-path weight ı.s; / is ﬁnite

if and only if is reachable from s, and thus the vertices that are reachable from s

are exactly those with ﬁnite d values. But a vertex 2 V fsg has been assigned

a ﬁnite value for :d if and only if : ¤ NIL. Thus, the vertices in V are exactly

those reachable from s.

The second property follows directly from Lemma 24.16.

It remains, therefore, to prove the last property of shortest-paths trees: for each

vertex 2 V , the unique simple path s

p in G is a shortest path from s to

in G. Let p D h 0 ; 1 ; : : : ; k i, where 0 D s and k D . For i D 1; 2; : : : ; k,

we have both i :d D ı.s; i / and i :d i1 :d C w. i1 ; i /, from which we

conclude w. i1 ; i / ı.s; i / ı.s; i1 /. Summing the weights along path p

yields

w.p/ D

k X

iD1

w. i1 ; i /

k X

iD1

.ı.s; i / ı.s; i1 //

D ı.s; k / ı.s; 0 / (because the sum telescopes)

D ı.s; k / (because ı.s; 0 / D ı.s; s/ D 0) .

Thus, w.p/ ı.s; k /. Since ı.s; k / is a lower bound on the weight of any path

from s to k , we conclude that w.p/ D ı.s; k /, and thus p is a shortest path

from s to D k .

Exercises

24.5-1

Give two shortest-paths trees for the directed graph of Figure 24.2 (on page 648)

other than the two shown.

24.5 Proofs of shortest-paths properties 677

24.5-2

Give an example of a weighted, directed graph G D .V; E/ with weight function

w W E ! R and source vertex s such that G satisﬁes the following property: For

every edge .u; / 2 E, there is a shortest-paths tree rooted at s that contains .u; /

and another shortest-paths tree rooted at s that does not contain .u; /.

24.5-3

Embellish the proof of Lemma 24.10 to handle cases in which shortest-path

weights are 1 or 1.

24.5-4

Let G D .V; E/ be a weighted, directed graph with source vertex s, and let G

be initialized by INITIALIZE-SINGLE-SOURCE.G; s/. Prove that if a sequence of

relaxation steps sets s: to a non-NIL value, then G contains a negative-weight

cycle.

24.5-5

Let G D .V; E/ be a weighted, directed graph with no negative-weight edges. Let

s 2 V be the source vertex, and suppose that we allow : to be the predecessor

of on any shortest path to from source s if 2 V fsg is reachable from s,

and NIL otherwise. Give an example of such a graph G and an assignment of

values that produces a cycle in G . (By Lemma 24.16, such an assignment cannot

be produced by a sequence of relaxation steps.)

24.5-6

Let G D .V; E/ be a weighted, directed graph with weight function w W E ! R

and no negative-weight cycles. Let s 2 V be the source vertex, and let G be initial-

ized by INITIALIZE-SINGLE-SOURCE.G; s/. Prove that for every vertex 2 V ,

there exists a path from s to in G and that this property is maintained as an

invariant over any sequence of relaxations.

24.5-7

Let G D .V; E/ be a weighted, directed graph that contains no negative-weight

cycles. Let s 2 V be the source vertex, and let G be initialized by INITIALIZE-

SINGLE-SOURCE.G; s/. Prove that there exists a sequence of jV j 1 relaxation

steps that produces :d D ı.s; / for all 2 V .

24.5-8

Let G be an arbitrary weighted, directed graph with a negative-weight cycle reach-

able from the source vertex s. Show how to construct an inﬁnite sequence of relax-

ations of the edges of G such that every relaxation causes a shortest-path estimate

to change.

678 Chapter 24 Single-Source Shortest Paths

Problems

24-1 Yen’s improvement to Bellman-Ford

Suppose that we order the edge relaxations in each pass of the Bellman-Ford al-

gorithm as follows. Before the ﬁrst pass, we assign an arbitrary linear order

1 ; 2 ; : : : ; jV j to the vertices of the input graph G D .V; E/. Then, we parti-

tion the edge set E into E f [ E b , where E f D f. i ; j / 2 E W i < j g and

E b D f. i ; j / 2 E W i > j g. (Assume that G contains no self-loops, so that every

edge is in either E f or E b .) Deﬁne G f D .V; E f / and G b D .V; E b /.

a. Prove that G f is acyclic with topological sort h 1 ; 2 ; : : : ; jV j i and that G b is

acyclic with topological sort h jV j ; jV j1 ; : : : ; 1 i.

Suppose that we implement each pass of the Bellman-Ford algorithm in the fol-

lowing way. We visit each vertex in the order 1 ; 2 ; : : : ; jV j , relaxing edges of E f

that leave the vertex. We then visit each vertex in the order jV j ; jV j1 ; : : : ; 1 ,

relaxing edges of E b that leave the vertex.

b. Prove that with this scheme, if G contains no negative-weight cycles that are

reachable from the source vertex s, then after only djV j =2e passes over the

edges, :d D ı.s; / for all vertices 2 V .

c. Does this scheme improve the asymptotic running time of the Bellman-Ford

algorithm?

24-2 Nesting boxes

A d-dimensional box with dimensions .x 1 ; x 2 ; : : : ; x d / nests within another box

with dimensions .y 1 ; y 2 ; : : : ; y d / if there exists a permutation on f1; 2; : : : ; dg

such that x .1/ < y 1 , x .2/ < y 2 , . . . , x .d/ < y d .

a. Argue that the nesting relation is transitive.

b. Describe an efﬁcient method to determine whether or not one d-dimensional

box nests inside another.

c. Suppose that you are given a set of n d-dimensional boxes fB 1 ; B 2 ; : : : ; B n g.

Give an efﬁcient algorithm to ﬁnd the longest sequence hB i 1 ; B i 2 ; : : : ; B i k i of

boxes such that B i j nests within B i j C1 for j D 1; 2; : : : ; k 1. Express the

running time of your algorithm in terms of n and d.

Problems for Chapter 24 679

24-3 Arbitrage

Arbitrage is the use of discrepancies in currency exchange rates to transform one

unit of a currency into more than one unit of the same currency. For example,

suppose that 1 U.S. dollar buys 49 Indian rupees, 1 Indian rupee buys 2 Japanese

yen, and 1 Japanese yen buys 0:0107 U.S. dollars. Then, by converting currencies,

a trader can start with 1 U.S. dollar and buy 49 2 0:0107 D 1:0486 U.S. dollars,

thus turning a proﬁt of 4:86 percent.

Suppose that we are given n currencies c 1 ; c 2 ; : : : ; c n and an n n table R of

exchange rates, such that one unit of currency c i buys RŒi; j units of currency c j .

a. Give an efﬁcient algorithm to determine whether or not there exists a sequence

of currencies hc i 1 ; c i 2 ; : : : ; c i k i such that

RŒi 1 ; i 2 RŒi 2 ; i 3 RŒi k1 ; i k RŒi k ; i 1 > 1 :

Analyze the running time of your algorithm.

b. Give an efﬁcient algorithm to print out such a sequence if one exists. Analyze

the running time of your algorithm.

24-4 Gabow’s scaling algorithm for single-source shortest paths

A scaling algorithm solves a problem by initially considering only the highest-

order bit of each relevant input value (such as an edge weight). It then reﬁnes the

initial solution by looking at the two highest-order bits. It progressively looks at

more and more high-order bits, reﬁning the solution each time, until it has exam-

ined all bits and computed the correct solution.

In this problem, we examine an algorithm for computing the shortest paths from

a single source by scaling edge weights. We are given a directed graph G D .V; E/

with nonnegative integer edge weights w. Let W D max .u;/2E fw.u; /g. Our

goal is to develop an algorithm that runs in O.E lg W / time. We assume that all

vertices are reachable from the source.

The algorithm uncovers the bits in the binary representation of the edge weights

one at a time, from the most signiﬁcant bit to the least signiﬁcant bit. Speciﬁcally,

let k D dlg.W C 1/e be the number of bits in the binary representation of W ,

and for i D 1; 2; : : : ; k, let w i .u; / D

w.u; /=2 ki

˘

. That is, w i .u; / is the

“scaled-down” version of w.u; / given by the i most signiﬁcant bits of w.u; /.

(Thus, w k .u; / D w.u; / for all .u; / 2 E.) For example, if k D 5 and

w.u; / D 25, which has the binary representation h11001i, then w 3 .u; / D

h110i D 6. As another example with k D 5, if w.u; / D h00100i D 4, then

w 3 .u; / D h001i D 1. Let us deﬁne ı i .u; / as the shortest-path weight from

vertex u to vertex using weight function w i . Thus, ı k .u; / D ı.u; / for all

u; 2 V . For a given source vertex s, the scaling algorithm ﬁrst computes the

680 Chapter 24 Single-Source Shortest Paths

shortest-path weights ı 1 .s; / for all 2 V , then computes ı 2 .s; / for all 2 V ,

and so on, until it computes ı k .s; / for all 2 V . We assume throughout that

jEj jV j 1, and we shall see that computing ı i from ı i1 takes O.E/ time, so

that the entire algorithm takes O.kE/ D O.E lg W / time.

a. Suppose that for all vertices 2 V , we have ı.s; / jEj. Show that we can

compute ı.s; / for all 2 V in O.E/ time.

b. Show that we can compute ı 1 .s; / for all 2 V in O.E/ time.

Let us now focus on computing ı i from ı i1 .

c. Prove that for i D 2; 3; : : : ; k, we have either w i .u; / D 2w i1 .u; / or

w i .u; / D 2w i1 .u; / C 1. Then, prove that

2ı i1 .s; / ı i .s; / 2ı i1 .s; / C jV j 1

for all 2 V .

d. Deﬁne for i D 2; 3; : : : ; k and all .u; / 2 E,

yw i .u; / D w i .u; / C 2ı i1 .s; u/ 2ı i1 .s; / :

Prove that for i D 2; 3; : : : ; k and all u; 2 V , the “reweighted” value yw i .u; /

of edge .u; / is a nonnegative integer.

e. Now, deﬁne

y

ı i .s; / as the shortest-path weight from s to using the weight

function yw i . Prove that for i D 2; 3; : : : ; k and all 2 V ,

ı i .s; / D

y

ı i .s; / C 2ı i1 .s; /

and that

y

ı i .s; / jEj.

f. Show how to compute ı i .s; / from ı i1 .s; / for all 2 V in O.E/ time, and

conclude that we can compute ı.s; / for all 2 V in O.E lg W / time.

24-5 Karp’s minimum mean-weight cycle algorithm

Let G D .V; E/ be a directed graph with weight function w W E ! R , and let

n D jV j. We deﬁne the mean weight of a cycle c D he 1 ; e 2 ; : : : ; e k i of edges in E

to be

.c/ D

1

k

k X

iD1

w.e i / :

Problems for Chapter 24 681

Let D min c .c/, where c ranges over all directed cycles in G. We call a cycle c

for which .c/ D

a minimum mean-weight cycle. This problem investigates

an efﬁcient algorithm for computing

.

Assume without loss of generality that every vertex 2 V is reachable from a

source vertex s 2 V . Let ı.s; / be the weight of a shortest path from s to , and let

ı k .s; / be the weight of a shortest path from s to consisting of exactly k edges.

If there is no path from s to with exactly k edges, then ı k .s; / D 1.

a. Show that if D 0, then G contains no negative-weight cycles and ı.s; / D

min 0kn1 ı k .s; / for all vertices 2 V .

b. Show that if D 0, then

max

0kn1

ı n .s; / ı k .s; /

n k

0

for all vertices 2 V . (Hint: Use both properties from part (a).)

c. Let c be a 0-weight cycle, and let u and be any two vertices on c. Suppose

that D 0 and that the weight of the simple path from u to along the cycle

is x. Prove that ı.s; / D ı.s; u/ C x. (Hint: The weight of the simple path

from to u along the cycle is x.)

d. Show that if D 0, then on each minimum mean-weight cycle there exists a

vertex such that

max

0kn1

ı n .s; / ı k .s; /

n k

D 0 :

(Hint: Show how to extend a shortest path to any vertex on a minimum mean-

weight cycle along the cycle to make a shortest path to the next vertex on the

cycle.)

e. Show that if D 0, then

min

2V

max

0kn1

ı n .s; / ı k .s; /

n k

D 0 :

f. Show that if we add a constant t to the weight of each edge of G, then

increases by t. Use this fact to show that

D min

2V

max

0kn1

ı n .s; / ı k .s; /

n k

:

g. Give an O.VE/-time algorithm to compute

.

682 Chapter 24 Single-Source Shortest Paths

24-6 Bitonic shortest paths

A sequence is bitonic if it monotonically increases and then monotonically de-

creases, or if by a circular shift it monotonically increases and then monotonically

decreases. For example the sequences h1; 4; 6; 8; 3; 2i, h9; 2; 4; 10; 5i, and

h1; 2; 3; 4i are bitonic, but h1; 3; 12; 4; 2; 10i is not bitonic. (See Problem 15-3 for

the bitonic euclidean traveling-salesman problem.)

Suppose that we are given a directed graph G D .V; E/ with weight function

w W E ! R , where all edge weights are unique, and we wish to ﬁnd single-source

shortest paths from a source vertex s. We are given one additional piece of infor-

mation: for each vertex 2 V , the weights of the edges along any shortest path

from s to form a bitonic sequence.

Give the most efﬁcient algorithm you can to solve this problem, and analyze its

running time.

Chapter notes

Dijkstra’s algorithm [88] appeared in 1959, but it contained no mention of a priority

queue. The Bellman-Ford algorithm is based on separate algorithms by Bellman

[38] and Ford [109]. Bellman describes the relation of shortest paths to difference

constraints. Lawler [224] describes the linear-time algorithm for shortest paths in

a dag, which he considers part of the folklore.

When edge weights are relatively small nonnegative integers, we have more ef-

ﬁcient algorithms to solve the single-source shortest-paths problem. The sequence

of values returned by the EXTRACT-MIN calls in Dijkstra’s algorithm monoton-

ically increases over time. As discussed in the chapter notes for Chapter 6, in

this case several data structures can implement the various priority-queue opera-

tions more efﬁciently than a binary heap or a Fibonacci heap. Ahuja, Mehlhorn,

Orlin, and Tarjan [8] give an algorithm that runs in O.E C V

p

lg W / time on

graphs with nonnegative edge weights, where W is the largest weight of any edge

in the graph. The best bounds are by Thorup [337], who gives an algorithm that

runs in O.E lg lg V / time, and by Raman [291], who gives an algorithm that runs

in O

E C V min

˚

.lg V / 1=3C ; .lg W / 1=4C

time. These two algorithms use an

amount of space that depends on the word size of the underlying machine. Al-

though the amount of space used can be unbounded in the size of the input, it can

be reduced to be linear in the size of the input using randomized hashing.

For undirected graphs with integer weights, Thorup [336] gives an O.V C E/-

time algorithm for single-source shortest paths. In contrast to the algorithms men-

tioned in the previous paragraph, this algorithm is not an implementation of Dijk-

Notes for Chapter 24 683

stra’s algorithm, since the sequence of values returned by EXTRACT-MIN calls

does not monotonically increase over time.

For graphs with negative edge weights, an algorithm due to Gabow and Tar-

jan [122] runs in O.

p

V E lg.V W // time, and one by Goldberg [137] runs in

O.

p

V E lg W / time, where W D max .u;/2E fjw.u; /jg.

Cherkassky, Goldberg, and Radzik [64] conducted extensive experiments com-

paring various shortest-path algorithms.

25 All-Pairs Shortest Paths

In this chapter, we consider the problem of ﬁnding shortest paths between all pairs

of vertices in a graph. This problem might arise in making a table of distances be-

tween all pairs of cities for a road atlas. As in Chapter 24, we are given a weighted,

directed graph G D .V; E/ with a weight function w W E ! R that maps edges

to real-valued weights. We wish to ﬁnd, for every pair of vertices u; 2 V , a

shortest (least-weight) path from u to , where the weight of a path is the sum of

the weights of its constituent edges. We typically want the output in tabular form:

the entry in u’s row and ’s column should be the weight of a shortest path from u

to .

We can solve an all-pairs shortest-paths problem by running a single-source

shortest-paths algorithm jV j times, once for each vertex as the source. If all

edge weights are nonnegative, we can use Dijkstra’s algorithm. If we use

the linear-array implementation of the min-priority queue, the running time is

O.V 3 C VE/ D O.V 3 /. The binary min-heap implementation of the min-priority

queue yields a running time of O.VE lg V /, which is an improvement if the graph

is sparse. Alternatively, we can implement the min-priority queue with a Fibonacci

heap, yielding a running time of O.V 2

lg V C VE/.

If the graph has negative-weight edges, we cannot use Dijkstra’s algorithm. In-

stead, we must run the slower Bellman-Ford algorithm once from each vertex. The

resulting running time is O.V 2 E/, which on a dense graph is O.V 4 /. In this chap-

ter we shall see how to do better. We also investigate the relation of the all-pairs

shortest-paths problem to matrix multiplication and study its algebraic structure.

Unlike the single-source algorithms, which assume an adjacency-list represen-

tation of the graph, most of the algorithms in this chapter use an adjacency-

matrix representation. (Johnson’s algorithm for sparse graphs, in Section 25.3,

uses adjacency lists.) For convenience, we assume that the vertices are numbered

1; 2; : : : ; jV j, so that the input is an n n matrix W representing the edge weights

of an n-vertex directed graph G D .V; E/. That is, W D .w ij /, where

Chapter 25 All-Pairs Shortest Paths 685

w ij D

0 if i D j ;

the weight of directed edge .i; j / if i ¤ j and .i; j / 2 E ;

1 if i ¤ j and .i; j / 62 E :

(25.1)

We allow negative-weight edges, but we assume for the time being that the input

graph contains no negative-weight cycles.

The tabular output of the all-pairs shortest-paths algorithms presented in this

chapter is an n n matrix D D .d ij /, where entry d ij contains the weight of a

shortest path from vertex i to vertex j . That is, if we let ı.i; j / denote the shortest-

path weight from vertex i to vertex j (as in Chapter 24), then d ij D ı.i; j / at

termination.

To solve the all-pairs shortest-paths problem on an input adjacency matrix, we

need to compute not only the shortest-path weights but also a predecessor matrix

… D . ij /, where ij is NIL if either i D j or there is no path from i to j ,

and otherwise ij is the predecessor of j on some shortest path from i. Just as

the predecessor subgraph G from Chapter 24 is a shortest-paths tree for a given

source vertex, the subgraph induced by the ith row of the … matrix should be a

shortest-paths tree with root i. For each vertex i 2 V , we deﬁne the predecessor

subgraph of G for i as G ;i D .V ;i ; E ;i / , where

V ;i D fj 2 V W ij ¤ NILg [ fig

and

E ;i D f. ij ; j / W j 2 V ;i figg :

If G ;i is a shortest-paths tree, then the following procedure, which is a modiﬁed

version of the PRINT-PATH procedure from Chapter 22, prints a shortest path from

vertex i to vertex j .

PRINT-ALL-PAIRS-SHORTEST-PATH.…; i; j /

1 if i == j

2 print i

3 elseif ij == NIL

4 print “no path from” i “to” j “exists”

5 else PRINT-ALL-PAIRS-SHORTEST-PATH.…; i; ij /

6 print j

In order to highlight the essential features of the all-pairs algorithms in this chapter,

we won’t cover the creation and properties of predecessor matrices as extensively

as we dealt with predecessor subgraphs in Chapter 24. Some of the exercises cover

the basics.

686 Chapter 25 All-Pairs Shortest Paths

Chapter outline

Section 25.1 presents a dynamic-programming algorithm based on matrix multi-

plication to solve the all-pairs shortest-paths problem. Using the technique of “re-

peated squaring,” we can achieve a running time of ‚.V 3

lg V /. Section 25.2 gives

another dynamic-programming algorithm, the Floyd-Warshall algorithm, which

runs in time ‚.V 3 /. Section 25.2 also covers the problem of ﬁnding the tran-

sitive closure of a directed graph, which is related to the all-pairs shortest-paths

problem. Finally, Section 25.3 presents Johnson’s algorithm, which solves the all-

pairs shortest-paths problem in O.V 2

lg V C VE/ time and is a good choice for

large, sparse graphs.

Before proceeding, we need to establish some conventions for adjacency-matrix

representations. First, we shall generally assume that the input graph G D .V; E/

has n vertices, so that n D jV j. Second, we shall use the convention of denoting

matrices by uppercase letters, such as W , L, or D, and their individual elements

by subscripted lowercase letters, such as w ij , l ij , or d ij . Some matrices will have

parenthesized superscripts, as in L .m/ D

l

.m/

ij

or D .m/ D

d

.m/

ij

, to indicate

iterates. Finally, for a given n n matrix A, we shall assume that the value of n is

stored in the attribute A:rows.

25.1 Shortest paths and matrix multiplication

This section presents a dynamic-programming algorithm for the all-pairs shortest-

paths problem on a directed graph G D .V; E/. Each major loop of the dynamic

program will invoke an operation that is very similar to matrix multiplication, so

that the algorithm will look like repeated matrix multiplication. We shall start by

developing a ‚.V 4 /-time algorithm for the all-pairs shortest-paths problem and

then improve its running time to ‚.V 3

lg V /.

Before proceeding, let us brieﬂy recap the steps given in Chapter 15 for devel-

oping a dynamic-programming algorithm.

1. Characterize the structure of an optimal solution.

2. Recursively deﬁne the value of an optimal solution.

3. Compute the value of an optimal solution in a bottom-up fashion.

We reserve the fourth step—constructing an optimal solution from computed in-

formation—for the exercises.

25.1 Shortest paths and matrix multiplication 687

The structure of a shortest path

We start by characterizing the structure of an optimal solution. For the all-pairs

shortest-paths problem on a graph G D .V; E/, we have proven (Lemma 24.1)

that all subpaths of a shortest path are shortest paths. Suppose that we represent

the graph by an adjacency matrix W D .w ij /. Consider a shortest path p from

vertex i to vertex j , and suppose that p contains at most m edges. Assuming that

there are no negative-weight cycles, m is ﬁnite. If i D j , then p has weight 0

and no edges. If vertices i and j are distinct, then we decompose path p into

i

p

0

k ! j , where path p 0

now contains at most m 1 edges. By Lemma 24.1,

p 0

is a shortest path from i to k, and so ı.i; j / D ı.i; k/ C w kj .

A recursive solution to the all-pairs shortest-paths problem

Now, let l

.m/

ij

be the minimum weight of any path from vertex i to vertex j that

contains at most m edges. When m D 0, there is a shortest path from i to j with

no edges if and only if i D j . Thus,

l

.0/

ij

D

(

0 if i D j ;

1 if i ¤ j :

For m 1, we compute l

.m/

ij

as the minimum of l

.m1/

ij

(the weight of a shortest

path from i to j consisting of at most m1 edges) and the minimum weight of any

path from i to j consisting of at most m edges, obtained by looking at all possible

predecessors k of j . Thus, we recursively deﬁne

l

.m/

ij

D min

l

.m1/

ij

; min

1kn

˚

l

.m1/

ik

C w kj

D min

1kn

˚

l

.m1/

ik

C w kj

: (25.2)

The latter equality follows since w jj D 0 for all j .

What are the actual shortest-path weights ı.i; j /? If the graph contains

no negative-weight cycles, then for every pair of vertices i and j for which

ı.i; j / < 1, there is a shortest path from i to j that is simple and thus contains at

most n 1 edges. A path from vertex i to vertex j with more than n 1 edges

cannot have lower weight than a shortest path from i to j . The actual shortest-path

weights are therefore given by

ı.i; j / D l

.n1/

ij

D l

.n/

ij

D l

.nC1/

ij

D : (25.3)

688 Chapter 25 All-Pairs Shortest Paths

Computing the shortest-path weights bottom up

Taking as our input the matrix W D .w ij /, we now compute a series of matrices

L .1/ ; L .2/ ; : : : ; L .n1/

, where for m D 1; 2; : : : ; n 1, we have L .m/ D

l

.m/

ij

.

The ﬁnal matrix L .n1/

contains the actual shortest-path weights. Observe that

l

.1/

ij

D w ij for all vertices i; j 2 V , and so L .1/ D W .

The heart of the algorithm is the following procedure, which, given matrices

L .m1/

and W , returns the matrix L .m/

. That is, it extends the shortest paths com-

puted so far by one more edge.

EXTEND-SHORTEST-PATHS.L; W /

1 n D L:rows

2 let L 0 D

l 0

ij

be a new n n matrix

3 for i D 1 to n

4 for j D 1 to n

5 l 0

ij

D 1

6 for k D 1 to n

7 l 0

ij

D min.l 0

ij

; l ik C w kj /

8 return L 0

The procedure computes a matrix L 0 D .l 0

ij

/, which it returns at the end. It does so

by computing equation (25.2) for all i and j , using L for L .m1/

and L 0

for L .m/

.

(It is written without the superscripts to make its input and output matrices inde-

pendent of m.) Its running time is ‚.n 3 / due to the three nested for loops.

Now we can see the relation to matrix multiplication. Suppose we wish to com-

pute the matrix product C D A B of two n n matrices A and B. Then, for

i; j D 1; 2; : : : ; n, we compute

c ij D

n X

kD1

a ik b kj : (25.4)

Observe that if we make the substitutions

l

.m1/

! a ;

w ! b ;

l

.m/

! c ;

min ! C ;

C !

in equation (25.2), we obtain equation (25.4). Thus, if we make these changes to

EXTEND-SHORTEST-PATHS and also replace 1 (the identity for min) by 0 (the

25.1 Shortest paths and matrix multiplication 689

identity for C), we obtain the same ‚.n 3 /-time procedure for multiplying square

matrices that we saw in Section 4.2:

SQUARE-MATRIX-MULTIPLY.A; B/

1 n D A:rows

2 let C be a new n n matrix

3 for i D 1 to n

4 for j D 1 to n

5 c ij D 0

6 for k D 1 to n

7 c ij D c ij C a ik b kj

8 return C

Returning to the all-pairs shortest-paths problem, we compute the shortest-path

weights by extending shortest paths edge by edge. Letting A B denote the ma-

trix “product” returned by EXTEND-SHORTEST-PATHS.A; B/, we compute the se-

quence of n 1 matrices

L .1/ D L .0/ W D W ;

L .2/ D L .1/ W D W 2 ;

L .3/ D L .2/ W D W 3 ;

:

:

:

L .n1/ D L .n2/ W D W n1 :

As we argued above, the matrix L .n1/ D W n1

contains the shortest-path weights.

The following procedure computes this sequence in ‚.n 4 / time.

SLOW-ALL-PAIRS-SHORTEST-PATHS.W /

1 n D W:rows

2 L .1/ D W

3 for m D 2 to n 1

4 let L .m/

be a new n n matrix

5 L .m/ D EXTEND-SHORTEST-PATHS.L .m1/ ; W /

6 return L .n1/

Figure 25.1 shows a graph and the matrices L .m/

computed by the procedure

SLOW-ALL-PAIRS-SHORTEST-PATHS.

Improving the running time

Our goal, however, is not to compute all the L .m/

matrices: we are interested

only in matrix L .n1/

. Recall that in the absence of negative-weight cycles, equa-

690 Chapter 25 All-Pairs Shortest Paths

2

1 3

5 4

3 4

8

2

6

7 1

–4 –5

L

.1/

D

0 3 8 1 4

1 0 1 1 7

1 4 0 1 1

2 1 5 0 1

1 1 1 6 0

˘

L

.2/

D

0 3 8 2 4

3 0 4 1 7

1 4 0 5 11

2 1 5 0 2

8 1 1 6 0

˘

L

.3/

D

0 3 3 2 4

3 0 4 1 1

7 4 0 5 11

2 1 5 0 2

8 5 1 6 0

˘

L

.4/

D

0 1 3 2 4

3 0 4 1 1

7 4 0 5 3

2 1 5 0 2

8 5 1 6 0

˘

Figure 25.1 A directed graph and the sequence of matrices L

.m/

computed by SLOW-ALL-PAIRS-

SHORTEST-PATHS. You might want to verify that L

.5/

, deﬁned as L

.4/

W , equals L

.4/

, and thus

L

.m/

D L

.4/

for all m 4.

tion (25.3) implies L .m/ D L .n1/

for all integers m n 1. Just as tradi-

tional matrix multiplication is associative, so is matrix multiplication deﬁned by

the EXTEND-SHORTEST-PATHS procedure (see Exercise 25.1-4). Therefore, we

can compute L .n1/

with only dlg.n 1/e matrix products by computing the se-

quence

L .1/ D W ;

L .2/ D W 2 D W W ;

L .4/ D W 4 D W 2 W 2

L .8/ D W 8 D W 4 W 4 ;

:

:

:

L .2

dlg.n1/e

/ D W 2

dlg.n1/e

D W 2

dlg.n1/e1

W 2

dlg.n1/e1

:

Since 2 dlg.n1/e n 1, the ﬁnal product L .2

dlg.n1/e

/

is equal to L .n1/

.

The following procedure computes the above sequence of matrices by using this

technique of repeated squaring.

25.1 Shortest paths and matrix multiplication 691

1 2

3

5 –1 2

1 2 3

4 5 6

–4 –8 10 7

Figure 25.2 A weighted, directed graph for use in Exercises 25.1-1, 25.2-1, and 25.3-1.

FASTER-ALL-PAIRS-SHORTEST-PATHS.W /

1 n D W:rows

2 L .1/ D W

3 m D 1

4 while m < n 1

5 let L .2m/

be a new n n matrix

6 L .2m/ D EXTEND-SHORTEST-PATHS.L .m/ ; L .m/ /

7 m D 2m

8 return L .m/

In each iteration of the while loop of lines 4–7, we compute L .2m/ D

L .m/

2

,

starting with m D 1. At the end of each iteration, we double the value

of m. The ﬁnal iteration computes L .n1/

by actually computing L .2m/

for some

n 1 2m < 2n 2. By equation (25.3), L .2m/ D L .n1/

. The next time the test

in line 4 is performed, m has been doubled, so now m n 1, the test fails, and

the procedure returns the last matrix it computed.

Because each of the dlg.n 1/e matrix products takes ‚.n 3 / time, FASTER-

ALL-PAIRS-SHORTEST-PATHS runs in ‚.n 3

lg n/ time. Observe that the code

is tight, containing no elaborate data structures, and the constant hidden in the

‚-notation is therefore small.

Exercises

25.1-1

Run SLOW-ALL-PAIRS-SHORTEST-PATHS on the weighted, directed graph of

Figure 25.2, showing the matrices that result for each iteration of the loop. Then

do the same for FASTER-ALL-PAIRS-SHORTEST-PATHS.

25.1-2

Why do we require that w ii D 0 for all 1 i n?

692 Chapter 25 All-Pairs Shortest Paths

25.1-3

What does the matrix

L

.0/

D

0 1 1 1

1 0 1 1

1 1 0 1

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

1 1 1 0

used in the shortest-paths algorithms correspond to in regular matrix multiplica-

tion?

25.1-4

Show that matrix multiplication deﬁned by EXTEND-SHORTEST-PATHS is asso-

ciative.

25.1-5

Show how to express the single-source shortest-paths problem as a product of ma-

trices and a vector. Describe how evaluating this product corresponds to a Bellman-

Ford-like algorithm (see Section 24.1).

25.1-6

Suppose we also wish to compute the vertices on shortest paths in the algorithms of

this section. Show how to compute the predecessor matrix … from the completed

matrix L of shortest-path weights in O.n 3 / time.

25.1-7

We can also compute the vertices on shortest paths as we compute the shortest-

path weights. Deﬁne

.m/

ij

as the predecessor of vertex j on any minimum-weight

path from i to j that contains at most m edges. Modify the EXTEND-SHORTEST-

PATHS and SLOW-ALL-PAIRS-SHORTEST-PATHS procedures to compute the ma-

trices … .1/ ; … .2/ ; : : : ; … .n1/

as the matrices L .1/ ; L .2/ ; : : : ; L .n1/

are computed.

25.1-8

The FASTER-ALL-PAIRS-SHORTEST-PATHS procedure, as written, requires us to

store dlg.n 1/e matrices, each with n 2

elements, for a total space requirement of

‚.n 2

lg n/. Modify the procedure to require only ‚.n 2 / space by using only two

n n matrices.

25.1-9

Modify FASTER-ALL-PAIRS-SHORTEST-PATHS so that it can determine whether

the graph contains a negative-weight cycle.

25.2 The Floyd-Warshall algorithm 693

25.1-10

Give an efﬁcient algorithm to ﬁnd the length (number of edges) of a minimum-

length negative-weight cycle in a graph.

25.2 The Floyd-Warshall algorithm

In this section, we shall use a different dynamic-programming formulation to solve

the all-pairs shortest-paths problem on a directed graph G D .V; E/. The result-

ing algorithm, known as the Floyd-Warshall algorithm, runs in ‚.V 3 / time. As

before, negative-weight edges may be present, but we assume that there are no

negative-weight cycles. As in Section 25.1, we follow the dynamic-programming

process to develop the algorithm. After studying the resulting algorithm, we

present a similar method for ﬁnding the transitive closure of a directed graph.

The structure of a shortest path

In the Floyd-Warshall algorithm, we characterize the structure of a shortest path

differently from how we characterized it in Section 25.1. The Floyd-Warshall algo-

rithm considers the intermediate vertices of a shortest path, where an intermediate

vertex of a simple path p D h 1 ; 2 ; : : : ; l i is any vertex of p other than 1 or l ,

that is, any vertex in the set f 2 ; 3 ; : : : ; l1 g.

The Floyd-Warshall algorithm relies on the following observation. Under our

assumption that the vertices of G are V D f1; 2; : : : ; ng, let us consider a subset

f1; 2; : : : ; kg of vertices for some k. For any pair of vertices i; j 2 V , consider all

paths from i to j whose intermediate vertices are all drawn from f1; 2; : : : ; kg, and

let p be a minimum-weight path from among them. (Path p is simple.) The Floyd-

Warshall algorithm exploits a relationship between path p and shortest paths from i

to j with all intermediate vertices in the set f1; 2; : : : ; k 1g. The relationship

depends on whether or not k is an intermediate vertex of path p.

If k is not an intermediate vertex of path p, then all intermediate vertices of

path p are in the set f1; 2; : : : ; k 1g. Thus, a shortest path from vertex i

to vertex j with all intermediate vertices in the set f1; 2; : : : ; k 1g is also a

shortest path from i to j with all intermediate vertices in the set f1; 2; : : : ; kg.

If k is an intermediate vertex of path p, then we decompose p into i

p 1 k

p 2 j ,

as Figure 25.3 illustrates. By Lemma 24.1, p 1 is a shortest path from i to k

with all intermediate vertices in the set f1; 2; : : : ; kg. In fact, we can make a

slightly stronger statement. Because vertex k is not an intermediate vertex of

path p 1 , all intermediate vertices of p 1 are in the set f1; 2; : : : ; k 1g. There-

694 Chapter 25 All-Pairs Shortest Paths

i

k

j

p 1 p 2

p: all intermediate vertices in f1; 2; : : : ; kg

all intermediate vertices in f1; 2; : : : ; k 1g all intermediate vertices in f1; 2; : : : ; k 1g

Figure 25.3 Path p is a shortest path from vertex i to vertex j , and k is the highest-numbered

intermediate vertex of p. Path p1, the portion of path p from vertex i to vertex k, has all intermediate

vertices in the set f1; 2; : : : ; k 1g. The same holds for path p2 from vertex k to vertex j .

fore, p 1 is a shortest path from i to k with all intermediate vertices in the set

f1; 2; : : : ; k 1g. Similarly, p 2 is a shortest path from vertex k to vertex j with

all intermediate vertices in the set f1; 2; : : : ; k 1g.

A recursive solution to the all-pairs shortest-paths problem

Based on the above observations, we deﬁne a recursive formulation of shortest-

path estimates that differs from the one in Section 25.1. Let d

.k/

ij

be the weight

of a shortest path from vertex i to vertex j for which all intermediate vertices

are in the set f1; 2; : : : ; kg. When k D 0, a path from vertex i to vertex j with

no intermediate vertex numbered higher than 0 has no intermediate vertices at all.

Such a path has at most one edge, and hence d

.0/

ij

D w ij . Following the above

discussion, we deﬁne d

.k/

ij

recursively by

d

.k/

ij

D

(

w ij if k D 0 ;

min

d

.k1/

ij

; d

.k1/

ik

C d

.k1/

kj

if k 1 :

(25.5)

Because for any path, all intermediate vertices are in the set f1; 2; : : : ; ng, the ma-

trix D .n/ D

d

.n/

ij

gives the ﬁnal answer: d

.n/

ij

D ı.i; j / for all i; j 2 V .

Computing the shortest-path weights bottom up

Based on recurrence (25.5), we can use the following bottom-up procedure to com-

pute the values d

.k/

ij

in order of increasing values of k. Its input is an n n matrix W

deﬁned as in equation (25.1). The procedure returns the matrix D .n/

of shortest-

path weights.

25.2 The Floyd-Warshall algorithm 695

FLOYD-WARSHALL.W /

1 n D W:rows

2 D .0/ D W

3 for k D 1 to n

4 let D .k/ D

d

.k/

ij

be a new n n matrix

5 for i D 1 to n

6 for j D 1 to n

7 d

.k/

ij

D min

d

.k1/

ij

; d

.k1/

ik

C d

.k1/

kj

8 return D .n/

Figure 25.4 shows the matrices D .k/

computed by the Floyd-Warshall algorithm

for the graph in Figure 25.1.

The running time of the Floyd-Warshall algorithm is determined by the triply

nested for loops of lines 3–7. Because each execution of line 7 takes O.1/ time,

the algorithm runs in time ‚.n 3 /. As in the ﬁnal algorithm in Section 25.1, the

code is tight, with no elaborate data structures, and so the constant hidden in the

‚-notation is small. Thus, the Floyd-Warshall algorithm is quite practical for even

moderate-sized input graphs.

Constructing a shortest path

There are a variety of different methods for constructing shortest paths in the Floyd-

Warshall algorithm. One way is to compute the matrix D of shortest-path weights

and then construct the predecessor matrix … from the D matrix. Exercise 25.1-6

asks you to implement this method so that it runs in O.n 3 / time. Given the pre-

decessor matrix …, the PRINT-ALL-PAIRS-SHORTEST-PATH procedure will print

the vertices on a given shortest path.

Alternatively, we can compute the predecessor matrix … while the algorithm

computes the matrices D .k/

. Speciﬁcally, we compute a sequence of matrices

… .0/ ; … .1/ ; : : : ; … .n/

, where … D … .n/

and we deﬁne

.k/

ij

as the predecessor of

vertex j on a shortest path from vertex i with all intermediate vertices in the set

f1; 2; : : : ; kg.

We can give a recursive formulation of

.k/

ij

. When k D 0, a shortest path from i

to j has no intermediate vertices at all. Thus,

.0/

ij

D

(

NIL if i D j or w ij D 1 ;

i if i ¤ j and w ij < 1 :

(25.6)

For k 1, if we take the path i k j , where k ¤ j , then the predecessor

of j we choose is the same as the predecessor of j we chose on a shortest path

from k with all intermediate vertices in the set f1; 2; : : : ; k 1g. Otherwise, we

696 Chapter 25 All-Pairs Shortest Paths

D

.0/

D

0 3 8 1 4

1 0 1 1 7

1 4 0 1 1

2 1 5 0 1

1 1 1 6 0

˘

…

.0/

D

NIL 1 1 NIL 1

NIL NIL NIL 2 2

NIL 3 NIL NIL NIL

4 NIL 4 NIL NIL

NIL NIL NIL 5 NIL

˘

D

.1/

D

0 3 8 1 4

1 0 1 1 7

1 4 0 1 1

2 5 5 0 2

1 1 1 6 0

˘

…

.1/

D

NIL 1 1 NIL 1

NIL NIL NIL 2 2

NIL 3 NIL NIL NIL

4 1 4 NIL 1

NIL NIL NIL 5 NIL

˘

D

.2/

D

0 3 8 4 4

1 0 1 1 7

1 4 0 5 11

2 5 5 0 2

1 1 1 6 0

˘

…

.2/

D

NIL 1 1 2 1

NIL NIL NIL 2 2

NIL 3 NIL 2 2

4 1 4 NIL 1

NIL NIL NIL 5 NIL

˘

D

.3/

D

0 3 8 4 4

1 0 1 1 7

1 4 0 5 11

2 1 5 0 2

1 1 1 6 0

˘

…

.3/

D

NIL 1 1 2 1

NIL NIL NIL 2 2

NIL 3 NIL 2 2

4 3 4 NIL 1

NIL NIL NIL 5 NIL

˘

D

.4/

D

0 3 1 4 4

3 0 4 1 1

7 4 0 5 3

2 1 5 0 2

8 5 1 6 0

˘

…

.4/

D

NIL 1 4 2 1

4 NIL 4 2 1

4 3 NIL 2 1

4 3 4 NIL 1

4 3 4 5 NIL

˘

D

.5/

D

0 1 3 2 4

3 0 4 1 1

7 4 0 5 3

2 1 5 0 2

8 5 1 6 0

˘

…

.5/

D

NIL 3 4 5 1

4 NIL 4 2 1

4 3 NIL 2 1

4 3 4 NIL 1

4 3 4 5 NIL

˘

Figure 25.4 The sequence of matrices D

.k/

and …

.k/

computed by the Floyd-Warshall algorithm

for the graph in Figure 25.1.

25.2 The Floyd-Warshall algorithm 697

choose the same predecessor of j that we chose on a shortest path from i with all

intermediate vertices in the set f1; 2; : : : ; k 1g. Formally, for k 1,

.k/

ij

D

(

.k1/

ij

if d

.k1/

ij

d

.k1/

ik

C d

.k1/

kj

;

.k1/

kj

if d

.k1/

ij

> d

.k1/

ik

C d

.k1/

kj

:

(25.7)

We leave the incorporation of the … .k/

matrix computations into the FLOYD-

WARSHALL procedure as Exercise 25.2-3. Figure 25.4 shows the sequence of … .k/

matrices that the resulting algorithm computes for the graph of Figure 25.1. The

exercise also asks for the more difﬁcult task of proving that the predecessor sub-

graph G ;i is a shortest-paths tree with root i. Exercise 25.2-7 asks for yet another

way to reconstruct shortest paths.

Transitive closure of a directed graph

Given a directed graph G D .V; E/ with vertex set V D f1; 2; : : : ; ng, we might

wish to determine whether G contains a path from i to j for all vertex pairs

i; j 2 V . We deﬁne the transitive closure of G as the graph G D .V; E /, where

E

D f.i; j / W there is a path from vertex i to vertex j in Gg :

One way to compute the transitive closure of a graph in ‚.n 3 / time is to assign

a weight of 1 to each edge of E and run the Floyd-Warshall algorithm. If there is a

path from vertex i to vertex j , we get d ij < n. Otherwise, we get d ij D 1.

There is another, similar way to compute the transitive closure of G in ‚.n 3 /

time that can save time and space in practice. This method substitutes the logical

operations \_ (logical OR) and ^ (logical AND) for the arithmetic operations min

and C in the Floyd-Warshall algorithm. For i; j; k D 1; 2; : : : ; n, we deﬁne t

.k/

ij

to

be 1 if there exists a path in graph G from vertex i to vertex j with all intermediate

vertices in the set f1; 2; : : : ; kg, and 0 otherwise. We construct the transitive closure

G D .V; E / by putting edge .i; j / into E

if and only if t

.n/

ij

D 1. A recursive

deﬁnition of t

.k/

ij

, analogous to recurrence (25.5), is

t

.0/

ij

D

(

0 if i ¤ j and .i; j / 62 E ;

1 if i D j or .i; j / 2 E ;

and for k 1,

t

.k/

ij

D t

.k1/

ij

\_

t

.k1/

ik

^ t

.k1/

kj

: (25.8)

As in the Floyd-Warshall algorithm, we compute the matrices T .k/ D

t

.k/

ij

in

order of increasing k.

698 Chapter 25 All-Pairs Shortest Paths

1 2

4 3

T

.0/

D

1 0 0 0

0 1 1 1

0 1 1 0

1 0 1 1

T

.1/

D

1 0 0 0

0 1 1 1

0 1 1 0

1 0 1 1

T

.2/

D

1 0 0 0

0 1 1 1

0 1 1 1

1 0 1 1

T

.3/

D

1 0 0 0

0 1 1 1

0 1 1 1

1 1 1 1

T

.4/

D

1 0 0 0

1 1 1 1

1 1 1 1

1 1 1 1

Figure 25.5 A directed graph and the matrices T

.k/

computed by the transitive-closure algorithm.

TRANSITIVE-CLOSURE.G/

1 n D jG:Vj

2 let T .0/ D

t

.0/

ij

be a new n n matrix

3 for i D 1 to n

4 for j D 1 to n

5 if i == j or .i; j / 2 G:E

6 t

.0/

ij

D 1

7 else t

.0/

ij

D 0

8 for k D 1 to n

9 let T .k/ D

t

.k/

ij

be a new n n matrix

10 for i D 1 to n

11 for j D 1 to n

12 t

.k/

ij

D t

.k1/

ij

\_

t

.k1/

ik

^ t

.k1/

kj

13 return T .n/

Figure 25.5 shows the matrices T .k/

computed by the TRANSITIVE-CLOSURE

procedure on a sample graph. The TRANSITIVE-CLOSURE procedure, like the

Floyd-Warshall algorithm, runs in ‚.n 3 / time. On some computers, though, log-

ical operations on single-bit values execute faster than arithmetic operations on

integer words of data. Moreover, because the direct transitive-closure algorithm

uses only boolean values rather than integer values, its space requirement is less

25.2 The Floyd-Warshall algorithm 699

than the Floyd-Warshall algorithm’s by a factor corresponding to the size of a word

of computer storage.

Exercises

25.2-1

Run the Floyd-Warshall algorithm on the weighted, directed graph of Figure 25.2.

Show the matrix D .k/

that results for each iteration of the outer loop.

25.2-2

Show how to compute the transitive closure using the technique of Section 25.1.

25.2-3

Modify the FLOYD-WARSHALL procedure to compute the … .k/

matrices according

to equations (25.6) and (25.7). Prove rigorously that for all i 2 V , the predecessor

subgraph G ;i is a shortest-paths tree with root i. (Hint: To show that G ;i is

acyclic, ﬁrst show that

.k/

ij

D l implies d

.k/

ij

d

.k/

il

C w lj , according to the

deﬁnition of

.k/

ij

. Then, adapt the proof of Lemma 24.16.)

25.2-4

As it appears above, the Floyd-Warshall algorithm requires ‚.n 3 / space, since we

compute d

.k/

ij

for i; j; k D 1; 2; : : : ; n. Show that the following procedure, which

simply drops all the superscripts, is correct, and thus only ‚.n 2 / space is required.

FLOYD-WARSHALL

0 .W /

1 n D W:rows

2 D D W

3 for k D 1 to n

4 for i D 1 to n

5 for j D 1 to n

6 d ij D min .d ij ; d ik C d kj /

7 return D

25.2-5

Suppose that we modify the way in which equation (25.7) handles equality:

.k/

ij

D

(

.k1/

ij

if d

.k1/

ij

< d

.k1/

ik

C d

.k1/

kj

;

.k1/

kj

if d

.k1/

ij

d

.k1/

ik

C d

.k1/

kj

:

Is this alternative deﬁnition of the predecessor matrix … correct?

700 Chapter 25 All-Pairs Shortest Paths

25.2-6

How can we use the output of the Floyd-Warshall algorithm to detect the presence

of a negative-weight cycle?

25.2-7

Another way to reconstruct shortest paths in the Floyd-Warshall algorithm uses

values

.k/

ij

for i; j; k D 1; 2; : : : ; n, where

.k/

ij

is the highest-numbered interme-

diate vertex of a shortest path from i to j in which all intermediate vertices are

in the set f1; 2; : : : ; kg. Give a recursive formulation for

.k/

ij

, modify the FLOYD-

WARSHALL procedure to compute the

.k/

ij

values, and rewrite the PRINT-ALL-

PAIRS-SHORTEST-PATH procedure to take the matrix ˆ D

.n/

ij

as an input.

How is the matrix ˆ like the s table in the matrix-chain multiplication problem of

Section 15.2?

25.2-8

Give an O.VE/-time algorithm for computing the transitive closure of a directed

graph G D .V; E/.

25.2-9

Suppose that we can compute the transitive closure of a directed acyclic graph in

f .jV j ; jEj/ time, where f is a monotonically increasing function of jV j and jEj.

Show that the time to compute the transitive closure G D .V; E / of a general

directed graph G D .V; E/ is then f .jV j ; jEj/ C O.V C E /.

25.3 Johnson’s algorithm for sparse graphs

Johnson’s algorithm ﬁnds shortest paths between all pairs in O.V 2

lg V C VE/

time. For sparse graphs, it is asymptotically faster than either repeated squaring of

matrices or the Floyd-Warshall algorithm. The algorithm either returns a matrix of

shortest-path weights for all pairs of vertices or reports that the input graph contains

a negative-weight cycle. Johnson’s algorithm uses as subroutines both Dijkstra’s

algorithm and the Bellman-Ford algorithm, which Chapter 24 describes.

Johnson’s algorithm uses the technique of reweighting, which works as follows.

If all edge weights w in a graph G D .V; E/ are nonnegative, we can ﬁnd short-

est paths between all pairs of vertices by running Dijkstra’s algorithm once from

each vertex; with the Fibonacci-heap min-priority queue, the running time of this

all-pairs algorithm is O.V 2

lg V C VE/. If G has negative-weight edges but no

negative-weight cycles, we simply compute a new set of nonnegative edge weights

25.3 Johnson’s algorithm for sparse graphs 701

that allows us to use the same method. The new set of edge weights yw must satisfy

two important properties:

1. For all pairs of vertices u; 2 V , a path p is a shortest path from u to using

weight function w if and only if p is also a shortest path from u to using

weight function yw.

2. For all edges .u; /, the new weight yw.u; / is nonnegative.

As we shall see in a moment, we can preprocess G to determine the new weight

function yw in O.VE/ time.

Preserving shortest paths by reweighting

The following lemma shows how easily we can reweight the edges to satisfy the

ﬁrst property above. We use ı to denote shortest-path weights derived from weight

function w and

y

ı to denote shortest-path weights derived from weight function yw.

Lemma 25.1 (Reweighting does not change shortest paths)

Given a weighted, directed graph G D .V; E/ with weight function w W E ! R ,

let h W V ! R be any function mapping vertices to real numbers. For each edge

.u; / 2 E, deﬁne

yw.u; / D w.u; / C h.u/ h./ : (25.9)

Let p D h 0 ; 1 ; : : : ; k i be any path from vertex 0 to vertex k . Then p is a

shortest path from 0 to k with weight function w if and only if it is a shortest path

with weight function yw. That is, w.p/ D ı. 0 ; k / if and only if yw.p/ D

y

ı. 0 ; k /.

Furthermore, G has a negative-weight cycle using weight function w if and only

if G has a negative-weight cycle using weight function yw.

Proof We start by showing that

yw.p/ D w.p/ C h. 0 / h. k / : (25.10)

We have

yw.p/ D

k X

iD1

yw. i1 ; i /

D

k X

iD1

.w. i1 ; i / C h. i1 / h. i //

D

k X

iD1

w. i1 ; i / C h. 0 / h. k / (because the sum telescopes)

D w.p/ C h. 0 / h. k / :

702 Chapter 25 All-Pairs Shortest Paths

Therefore, any path p from 0 to k has yw.p/ D w.p/ C h. 0 / h. k /. Be-

cause h. 0 / and h. k / do not depend on the path, if one path from 0 to k is

shorter than another using weight function w, then it is also shorter using yw. Thus,

w.p/ D ı. 0 ; k / if and only if yw.p/ D

y

ı. 0 ; k /.

Finally, we show that G has a negative-weight cycle using weight function w if

and only if G has a negative-weight cycle using weight function yw. Consider any

cycle c D h 0 ; 1 ; : : : ; k i, where 0 D k . By equation (25.10),

yw.c/ D w.c/ C h. 0 / h. k /

D w.c/ ;

and thus c has negative weight using w if and only if it has negative weight us-

ing yw.

Producing nonnegative weights by reweighting

Our next goal is to ensure that the second property holds: we want yw.u; / to be

nonnegative for all edges .u; / 2 E. Given a weighted, directed graph G D

.V; E/ with weight function w W E ! R , we make a new graph G 0 D .V 0 ; E 0 /,

where V 0 D V [ fsg for some new vertex s 62 V and E 0 D E [ f.s; / W 2 V g.

We extend the weight function w so that w.s; / D 0 for all 2 V . Note that

because s has no edges that enter it, no shortest paths in G 0

, other than those with

source s, contain s. Moreover, G 0

has no negative-weight cycles if and only if G

has no negative-weight cycles. Figure 25.6(a) shows the graph G 0

corresponding

to the graph G of Figure 25.1.

Now suppose that G and G 0

have no negative-weight cycles. Let us deﬁne

h./ D ı.s; / for all 2 V 0

. By the triangle inequality (Lemma 24.10),

we have h./ h.u/ C w.u; / for all edges .u; / 2 E 0

. Thus, if we de-

ﬁne the new weights yw by reweighting according to equation (25.9), we have

yw.u; / D w.u; / C h.u/ h./ 0, and we have satisﬁed the second property.

Figure 25.6(b) shows the graph G 0

from Figure 25.6(a) with reweighted edges.

Computing all-pairs shortest paths

Johnson’s algorithm to compute all-pairs shortest paths uses the Bellman-Ford al-

gorithm (Section 24.1) and Dijkstra’s algorithm (Section 24.3) as subroutines. It

assumes implicitly that the edges are stored in adjacency lists. The algorithm re-

turns the usual jV j jV j matrix D D d ij , where d ij D ı.i; j /, or it reports that

the input graph contains a negative-weight cycle. As is typical for an all-pairs

shortest-paths algorithm, we assume that the vertices are numbered from 1 to jV j.

25.3 Johnson’s algorithm for sparse graphs 703

2

1

5

4

3 4

8

2

6

7 1

0

0

0

0

0

0

0

2/1

2/–3

2/2 0/–4

2/3 0/–4

0/1 2/–1

2/7

0/4

0/5 2/3

2/2

0/–1

0/–5

2/–2

4/8

2/5

2/1

2/6

(a)

(c)

(b)

–4

–4

–1

–5

–5

3

2

1

5

4

4 0

13

2

2

10 0

5

1

0

4

0

0

0

0

–4

–1

–5

0

3

2

1

5 4

4 0

13

2

2

10 0

0 0

3

(d)

2

1

5 4

4 0

13

2

2

10 0

0 0

3

(e)

2

1

5 4

4 0

13

2

2

10 0

0 0

3

(f)

2

1

5 4

4 0

13

2

2

10 0

0 0

3

(g)

2

1

5 4

4 0

13

2

2

10 0

0 0

3

0/0 0/0

0/0

0/0

0/0

0 0

Figure 25.6 Johnson’s all-pairs shortest-paths algorithm run on the graph of Figure 25.1. Ver-

tex numbers appear outside the vertices. (a) The graph G

0

with the original weight function w.

The new vertex s is black. Within each vertex is h./ D ı.s; /. (b) After reweighting each

edge .u; / with weight function yw.u; / D w.u; / C h.u/ h./. (c)–(g) The result of running

is black, and shaded edges are in the shortest-paths tree computed by the algorithm. Within each

vertex are the values

y

ı.u; / and ı.u; /, separated by a slash. The value du D ı.u; / is equal to

y

ı. / C h./ h.u/

Dijkstra’s algorithm on each vertex of G using weight function wy. In each part, the source vertex u

. u;

704 Chapter 25 All-Pairs Shortest Paths

JOHNSON.G; w/

1 compute G 0

, where G 0 :V D G:V [ fsg,

G 0 :E D G:E [ f.s; / W 2 G:Vg, and

w.s; / D 0 for all 2 G:V

2 if BELLMAN-FORD.G 0 ; w; s/ == FALSE

3 print “the input graph contains a negative-weight cycle”

4 else for each vertex 2 G 0 :V

5 set h./ to the value of ı.s; /

computed by the Bellman-Ford algorithm

6 for each edge .u; / 2 G 0 :E

7 yw.u; / D w.u; / C h.u/ h./

8 let D D .d u / be a new n n matrix

9 for each vertex u 2 G:V

10 run DIJKSTRA.G; yw; u/ to compute

y

ı.u; / for all 2 G:V

11 for each vertex 2 G:V

12 d u D

y

ı.u; / C h./ h.u/

13 return D

This code simply performs the actions we speciﬁed earlier. Line 1 produces G 0

.

Line 2 runs the Bellman-Ford algorithm on G 0

with weight function w and source

vertex s. If G 0

, and hence G, contains a negative-weight cycle, line 3 reports the

problem. Lines 4–12 assume that G 0

contains no negative-weight cycles. Lines 4–5

set h./ to the shortest-path weight ı.s; / computed by the Bellman-Ford algo-

rithm for all 2 V 0

. Lines 6–7 compute the new weights yw. For each pair of ver-

tices u; 2 V , the for loop of lines 9–12 computes the shortest-path weight

y

ı.u; /

by calling Dijkstra’s algorithm once from each vertex in V . Line 12 stores in

matrix entry d u the correct shortest-path weight ı.u; /, calculated using equa-

tion (25.10). Finally, line 13 returns the completed D matrix. Figure 25.6 depicts

the execution of Johnson’s algorithm.

If we implement the min-priority queue in Dijkstra’s algorithm by a Fibonacci

heap, Johnson’s algorithm runs in O.V 2

lg V CVE/ time. The simpler binary min-

heap implementation yields a running time of O.VE lg V /, which is still asymp-

totically faster than the Floyd-Warshall algorithm if the graph is sparse.

Exercises

25.3-1

Use Johnson’s algorithm to ﬁnd the shortest paths between all pairs of vertices in

the graph of Figure 25.2. Show the values of h and yw computed by the algorithm.

Problems for Chapter 25 705

25.3-2

What is the purpose of adding the new vertex s to V , yielding V 0

?

25.3-3

Suppose that w.u; / 0 for all edges .u; / 2 E. What is the relationship

between the weight functions w and yw?

25.3-4

Professor Greenstreet claims that there is a simpler way to reweight edges than

the method used in Johnson’s algorithm. Letting w D min .u;/2E fw.u; /g, just

deﬁne yw.u; / D w.u; / w

for all edges .u; / 2 E. What is wrong with the

professor’s method of reweighting?

25.3-5

Suppose that we run Johnson’s algorithm on a directed graph G with weight func-

tion w. Show that if G contains a 0-weight cycle c, then yw.u; / D 0 for every

edge .u; / in c.

25.3-6

Professor Michener claims that there is no need to create a new source vertex in

line 1 of JOHNSON. He claims that instead we can just use G 0 D G and let s be any

vertex. Give an example of a weighted, directed graph G for which incorporating

the professor’s idea into JOHNSON causes incorrect answers. Then show that if G

is strongly connected (every vertex is reachable from every other vertex), the results

returned by JOHNSON with the professor’s modiﬁcation are correct.

Problems

25-1 Transitive closure of a dynamic graph

Suppose that we wish to maintain the transitive closure of a directed graph G D

.V; E/ as we insert edges into E. That is, after each edge has been inserted, we

want to update the transitive closure of the edges inserted so far. Assume that the

graph G has no edges initially and that we represent the transitive closure as a

boolean matrix.

a. Show how to update the transitive closure G D .V; E / of a graph G D .V; E/

in O.V 2 / time when a new edge is added to G.

b. Give an example of a graph G and an edge e such that .V 2 / time is required

to update the transitive closure after the insertion of e into G, no matter what

algorithm is used.

706 Chapter 25 All-Pairs Shortest Paths

c. Describe an efﬁcient algorithm for updating the transitive closure as edges are

inserted into the graph. For any sequence of n insertions, your algorithm should

run in total time

P n

iD1

t i D O.V 3 /, where t i is the time to update the transitive

closure upon inserting the ith edge. Prove that your algorithm attains this time

bound.

25-2 Shortest paths in -dense graphs

A graph G D .V; E/ is -dense if jEj D ‚.V 1C / for some constant in the

range 0 < 1. By using d-ary min-heaps (see Problem 6-2) in shortest-paths

algorithms on -dense graphs, we can match the running times of Fibonacci-heap-

based algorithms without using as complicated a data structure.

a. What are the asymptotic running times for INSERT, EXTRACT-MIN, and

DECREASE-KEY, as a function of d and the number n of elements in a d-ary

min-heap? What are these running times if we choose d D ‚.n ˛ / for some

constant 0 < ˛ 1? Compare these running times to the amortized costs of

these operations for a Fibonacci heap.

b. Show how to compute shortest paths from a single source on an -dense directed

graph G D .V; E/ with no negative-weight edges in O.E/ time. (Hint: Pick d

as a function of .)

c. Show how to solve the all-pairs shortest-paths problem on an -dense directed

graph G D .V; E/ with no negative-weight edges in O.VE/ time.

d. Show how to solve the all-pairs shortest-paths problem in O.VE/ time on an

-dense directed graph G D .V; E/ that may have negative-weight edges but

has no negative-weight cycles.

Chapter notes

Lawler [224] has a good discussion of the all-pairs shortest-paths problem, al-

though he does not analyze solutions for sparse graphs. He attributes the matrix-

multiplication algorithm to the folklore. The Floyd-Warshall algorithm is due to

Floyd [105], who based it on a theorem of Warshall [349] that describes how to

compute the transitive closure of boolean matrices. Johnson’s algorithm is taken

from [192].

Several researchers have given improved algorithms for computing shortest

paths via matrix multiplication. Fredman [111] shows how to solve the all-

pairs shortest paths problem using O.V 5=2 / comparisons between sums of edge

Notes for Chapter 25 707

weights and obtains an algorithm that runs in O.V 3 .lg lg V= lg V / 1=3 / time, which

is slightly better than the running time of the Floyd-Warshall algorithm. Han [159]

reduced the running time to O.V 3 .lg lg V= lg V / 5=4 /. Another line of research

demonstrates that we can apply algorithms for fast matrix multiplication (see the

chapter notes for Chapter 4) to the all-pairs shortest paths problem. Let O.n ! / be

the running time of the fastest algorithm for multiplying n n matrices; currently

! < 2:376 [78]. Galil and Margalit [123, 124] and Seidel [308] designed algo-

rithms that solve the all-pairs shortest paths problem in undirected, unweighted

graphs in .V ! p.V // time, where p.n/ denotes a particular function that is poly-

logarithmically bounded in n. In dense graphs, these algorithms are faster than

the O.VE/ time needed to perform jV j breadth-ﬁrst searches. Several researchers

have extended these results to give algorithms for solving the all-pairs shortest

paths problem in undirected graphs in which the edge weights are integers in the

range f1; 2; : : : ; W g. The asymptotically fastest such algorithm, by Shoshan and

Zwick [316], runs in time O.W V ! p.V W //.

Karger, Koller, and Phillips [196] and independently McGeoch [247] have given

a time bound that depends on E

, the set of edges in E that participate in some

shortest path. Given a graph with nonnegative edge weights, their algorithms run in

O.VE C V 2

lg V / time and improve upon running Dijkstra’s algorithm jV j times

when jE j D o.E/.

Baswana, Hariharan, and Sen [33] examined decremental algorithms for main-

taining all-pairs shortest paths and transitive-closure information. Decremen-

tal algorithms allow a sequence of intermixed edge deletions and queries; by

comparison, Problem 25-1, in which edges are inserted, asks for an incremen-

tal algorithm. The algorithms by Baswana, Hariharan, and Sen are randomized

and, when a path exists, their transitive-closure algorithm can fail to report it

with probability 1=n c

for an arbitrary c > 0. The query times are O.1/ with

high probability. For transitive closure, the amortized time for each update is

O.V 4=3

lg

1=3

V /. For all-pairs shortest paths, the update times depend on the

queries. For queries just giving the shortest-path weights, the amortized time per

update is O.V 3 =E lg

2

V /. To report the actual shortest path, the amortized up-

date time is min.O.V 3=2

p

lg V /; O.V 3 =E lg

2

V //. Demetrescu and Italiano [84]

showed how to handle update and query operations when edges are both inserted

and deleted, as long as each given edge has a bounded range of possible values

drawn from the real numbers.

Aho, Hopcroft, and Ullman [5] deﬁned an algebraic structure known as a “closed

semiring,” which serves as a general framework for solving path problems in di-

rected graphs. Both the Floyd-Warshall algorithm and the transitive-closure algo-

rithm from Section 25.2 are instantiations of an all-pairs algorithm based on closed

semirings. Maggs and Plotkin [240] showed how to ﬁnd minimum spanning trees

using a closed semiring.

26 Maximum Flow

Just as we can model a road map as a directed graph in order to ﬁnd the shortest

path from one point to another, we can also interpret a directed graph as a “ﬂow

network” and use it to answer questions about material ﬂows. Imagine a mate-

rial coursing through a system from a source, where the material is produced, to

a sink, where it is consumed. The source produces the material at some steady

rate, and the sink consumes the material at the same rate. The “ﬂow” of the mate-

rial at any point in the system is intuitively the rate at which the material moves.

Flow networks can model many problems, including liquids ﬂowing through pipes,

parts through assembly lines, current through electrical networks, and information

through communication networks.

We can think of each directed edge in a ﬂow network as a conduit for the mate-

rial. Each conduit has a stated capacity, given as a maximum rate at which the ma-

terial can ﬂow through the conduit, such as 200 gallons of liquid per hour through

a pipe or 20 amperes of electrical current through a wire. Vertices are conduit

junctions, and other than the source and sink, material ﬂows through the vertices

without collecting in them. In other words, the rate at which material enters a ver-

tex must equal the rate at which it leaves the vertex. We call this property “ﬂow

conservation,” and it is equivalent to Kirchhoff’s current law when the material is

electrical current.

In the maximum-ﬂow problem, we wish to compute the greatest rate at which

we can ship material from the source to the sink without violating any capacity

constraints. It is one of the simplest problems concerning ﬂow networks and, as

we shall see in this chapter, this problem can be solved by efﬁcient algorithms.

Moreover, we can adapt the basic techniques used in maximum-ﬂow algorithms to

solve other network-ﬂow problems.

This chapter presents two general methods for solving the maximum-ﬂow prob-

lem. Section 26.1 formalizes the notions of ﬂow networks and ﬂows, formally

deﬁning the maximum-ﬂow problem. Section 26.2 describes the classical method

of Ford and Fulkerson for ﬁnding maximum ﬂows. An application of this method,

26.1 Flow networks 709

ﬁnding a maximum matching in an undirected bipartite graph, appears in Sec-

tion 26.3. Section 26.4 presents the push-relabel method, which underlies many of

the fastest algorithms for network-ﬂow problems. Section 26.5 covers the “relabel-

to-front” algorithm, a particular implementation of the push-relabel method that

runs in time O.V 3 /. Although this algorithm is not the fastest algorithm known,

it illustrates some of the techniques used in the asymptotically fastest algorithms,

and it is reasonably efﬁcient in practice.

26.1 Flow networks

In this section, we give a graph-theoretic deﬁnition of ﬂow networks, discuss their

properties, and deﬁne the maximum-ﬂow problem precisely. We also introduce

some helpful notation.

Flow networks and ﬂows

A ﬂow network G D .V; E/ is a directed graph in which each edge .u; / 2 E

has a nonnegative capacity c.u; / 0. We further require that if E contains an

edge .u; /, then there is no edge .; u/ in the reverse direction. (We shall see

shortly how to work around this restriction.) If .u; / 62 E, then for convenience

we deﬁne c.u; / D 0, and we disallow self-loops. We distinguish two vertices

in a ﬂow network: a source s and a sink t. For convenience, we assume that each

vertex lies on some path from the source to the sink. That is, for each vertex 2 V ,

the ﬂow network contains a path s t. The graph is therefore connected

and, since each vertex other than s has at least one entering edge, jEj jV j 1.

Figure 26.1 shows an example of a ﬂow network.

We are now ready to deﬁne ﬂows more formally. Let G D .V; E/ be a ﬂow

network with a capacity function c. Let s be the source of the network, and let t be

the sink. A ﬂow in G is a real-valued function f W V V ! R that satisﬁes the

following two properties:

Capacity constraint: For all u; 2 V , we require 0 f .u; / c.u; /.

Flow conservation: For all u 2 V fs; tg, we require

X

2V

f .; u/ D

X

2V

f .u; / :

When .u; / 62 E, there can be no ﬂow from u to , and f .u; / D 0.

710 Chapter 26 Maximum Flow

s t

16

12

20

7

9

4

13

14

4

Edmonton

Calgary

Saskatoon

Regina

Vancouver Winnipeg

s t

11/16

12/12

15/20

7/7

4/9

1/4

8/13

11/14

4/4

(a) (b)

v

1

v

1

v

2

v

2

v

3

v

3

v

4

v

4

Figure 26.1 (a) A ﬂow network G D .V; E/ for the Lucky Puck Company’s trucking problem.

The Vancouver factory is the source s, and the Winnipeg warehouse is the sink t. The company ships

pucks through intermediate cities, but only c.u; / crates per day can go from city u to city . Each

edge is labeled with its capacity. (b) A ﬂow f in G with value jf j D 19. Each edge .u; / is labeled

by f .u; /=c.u; /. The slash notation merely separates the ﬂow and capacity; it does not indicate

division.

We call the nonnegative quantity f .u; / the ﬂow from vertex u to vertex . The

value jf j of a ﬂow f is deﬁned as

jf j D

X

2V

f .s; /

X

2V

f .; s/ ; (26.1)

that is, the total ﬂow out of the source minus the ﬂow into the source. (Here, the jj

notation denotes ﬂow value, not absolute value or cardinality.) Typically, a ﬂow

network will not have any edges into the source, and the ﬂow into the source, given

by the summation

P

2V

f .; s/, will be 0. We include it, however, because when

we introduce residual networks later in this chapter, the ﬂow into the source will

become signiﬁcant. In the maximum-ﬂow problem, we are given a ﬂow network G

with source s and sink t, and we wish to ﬁnd a ﬂow of maximum value.

Before seeing an example of a network-ﬂow problem, let us brieﬂy explore the

deﬁnition of ﬂow and the two ﬂow properties. The capacity constraint simply

says that the ﬂow from one vertex to another must be nonnegative and must not

exceed the given capacity. The ﬂow-conservation property says that the total ﬂow

into a vertex other than the source or sink must equal the total ﬂow out of that

vertex—informally, “ﬂow in equals ﬂow out.”

An example of ﬂow

A ﬂow network can model the trucking problem shown in Figure 26.1(a). The

Lucky Puck Company has a factory (source s) in Vancouver that manufactures

hockey pucks, and it has a warehouse (sink t) in Winnipeg that stocks them. Lucky

26.1 Flow networks 711

s t

16

12

20

7

9

4

13

14

4(a)(b)

v

1

v

2

v

3

v

4

10

s t

16

12

20

7

9

4

13

14

4

v

1

v

2

v

3

v

4

v ′

10

10

Figure 26.2 Converting a network with antiparallel edges to an equivalent one with no antiparallel

edges. (a) A ﬂow network containing both the edges .1; 2/ and .2; 1/. (b) An equivalent network

with no antiparallel edges. We add the new vertex

0

, and we replace edge .1; 2/ by the pair of

edges .1;

0

/ and .

0

; 2/, both with the same capacity as .1; 2/.

Puck leases space on trucks from another ﬁrm to ship the pucks from the factory

to the warehouse. Because the trucks travel over speciﬁed routes (edges) between

cities (vertices) and have a limited capacity, Lucky Puck can ship at most c.u; /

crates per day between each pair of cities u and in Figure 26.1(a). Lucky Puck

has no control over these routes and capacities, and so the company cannot alter

the ﬂow network shown in Figure 26.1(a). They need to determine the largest

number p of crates per day that they can ship and then to produce this amount, since

there is no point in producing more pucks than they can ship to their warehouse.

Lucky Puck is not concerned with how long it takes for a given puck to get from

the factory to the warehouse; they care only that p crates per day leave the factory

and p crates per day arrive at the warehouse.

We can model the “ﬂow” of shipments with a ﬂow in this network because the

number of crates shipped per day from one city to another is subject to a capacity

constraint. Additionally, the model must obey ﬂow conservation, for in a steady

state, the rate at which pucks enter an intermediate city must equal the rate at which

they leave. Otherwise, crates would accumulate at intermediate cities.

Modeling problems with antiparallel edges

Suppose that the trucking ﬁrm offered Lucky Puck the opportunity to lease space

for 10 crates in trucks going from Edmonton to Calgary. It would seem natural to

add this opportunity to our example and form the network shown in Figure 26.2(a).

This network suffers from one problem, however: it violates our original assump-

tion that if an edge . 1 ; 2 / 2 E, then . 2 ; 1 / 62 E. We call the two edges . 1 ; 2 /

and . 2 ; 1 / antiparallel. Thus, if we wish to model a ﬂow problem with antipar-

allel edges, we must transform the network into an equivalent one containing no

712 Chapter 26 Maximum Flow

antiparallel edges. Figure 26.2(b) displays this equivalent network. We choose

one of the two antiparallel edges, in this case . 1 ; 2 /, and split it by adding a new

vertex 0

and replacing edge . 1 ; 2 / with the pair of edges . 1 ; 0 / and . 0 ; 2 /.

We also set the capacity of both new edges to the capacity of the original edge.

The resulting network satisﬁes the property that if an edge is in the network, the

reverse edge is not. Exercise 26.1-1 asks you to prove that the resulting network is

equivalent to the original one.

Thus, we see that a real-world ﬂow problem might be most naturally modeled

by a network with antiparallel edges. It will be convenient to disallow antipar-

allel edges, however, and so we have a straightforward way to convert a network

containing antiparallel edges into an equivalent one with no antiparallel edges.

Networks with multiple sources and sinks

A maximum-ﬂow problem may have several sources and sinks, rather than just

one of each. The Lucky Puck Company, for example, might actually have a set

of m factories fs 1 ; s 2 ; : : : ; s m g and a set of n warehouses ft 1 ; t 2 ; : : : ; t n g, as shown

in Figure 26.3(a). Fortunately, this problem is no harder than ordinary maximum

ﬂow.

We can reduce the problem of determining a maximum ﬂow in a network with

multiple sources and multiple sinks to an ordinary maximum-ﬂow problem. Fig-

ure 26.3(b) shows how to convert the network from (a) to an ordinary ﬂow network

with only a single source and a single sink. We add a supersource s and add a

directed edge .s; s i / with capacity c.s; s i / D 1 for each i D 1; 2; : : : ; m. We also

create a new supersink t and add a directed edge .t i ; t/ with capacity c.t i ; t/ D 1

for each i D 1; 2; : : : ; n. Intuitively, any ﬂow in the network in (a) corresponds to

a ﬂow in the network in (b), and vice versa. The single source s simply provides

as much ﬂow as desired for the multiple sources s i , and the single sink t likewise

consumes as much ﬂow as desired for the multiple sinks t i . Exercise 26.1-2 asks

you to prove formally that the two problems are equivalent.

Exercises

26.1-1

Show that splitting an edge in a ﬂow network yields an equivalent network. More

formally, suppose that ﬂow network G contains edge .u; /, and we create a new

ﬂow network G 0

by creating a new vertex x and replacing .u; / by new edges

.u; x/ and .x; / with c.u; x/ D c.x; / D c.u; /. Show that a maximum ﬂow

in G 0

has the same value as a maximum ﬂow in G.

26.1 Flow networks 713

10

(a)

12

5

8

14

7

11

2

3

15

6

20

13

18

10

12

5

8

14

7

11

2

3

15

6

20

13

18

∞ ∞

∞

∞

∞

∞

∞

∞

s

1

s

1

s

2

s

2

s

3

s

3

s

4

s

4

s

5

s

5

t

1

t

1

t

2

t

2

t

3

t

3

(b)

s t

Figure 26.3 Converting a multiple-source, multiple-sink maximum-ﬂow problem into a problem

with a single source and a single sink. (a) A ﬂow network with ﬁve sources S D fs1; s2; s3; s4; s5g

and three sinks T D ft1; t2; t3g. (b) An equivalent single-source, single-sink ﬂow network. We add

a supersource s and an edge with inﬁnite capacity from s to each of the multiple sources. We also

add a supersink t and an edge with inﬁnite capacity from each of the multiple sinks to t.

26.1-2

Extend the ﬂow properties and deﬁnitions to the multiple-source, multiple-sink

problem. Show that any ﬂow in a multiple-source, multiple-sink ﬂow network

corresponds to a ﬂow of identical value in the single-source, single-sink network

obtained by adding a supersource and a supersink, and vice versa.

26.1-3

Suppose that a ﬂow network G D .V; E/ violates the assumption that the network

contains a path s t for all vertices 2 V . Let u be a vertex for which there

is no path s u t. Show that there must exist a maximum ﬂow f in G such

that f .u; / D f .; u/ D 0 for all vertices 2 V .

714 Chapter 26 Maximum Flow

26.1-4

Let f be a ﬂow in a network, and let ˛ be a real number. The scalar ﬂow product,

denoted ˛f , is a function from V V to R deﬁned by

.˛f /.u; / D ˛ f .u; / :

Prove that the ﬂows in a network form a convex set. That is, show that if f 1 and f 2

are ﬂows, then so is ˛f 1 C .1 ˛/f 2 for all ˛ in the range 0 ˛ 1.

26.1-5

State the maximum-ﬂow problem as a linear-programming problem.

26.1-6

Professor Adam has two children who, unfortunately, dislike each other. The prob-

lem is so severe that not only do they refuse to walk to school together, but in fact

each one refuses to walk on any block that the other child has stepped on that day.

The children have no problem with their paths crossing at a corner. Fortunately

both the professor’s house and the school are on corners, but beyond that he is not

sure if it is going to be possible to send both of his children to the same school.

The professor has a map of his town. Show how to formulate the problem of de-

termining whether both his children can go to the same school as a maximum-ﬂow

problem.

26.1-7

Suppose that, in addition to edge capacities, a ﬂow network has vertex capacities.

That is each vertex has a limit l./ on how much ﬂow can pass though . Show

how to transform a ﬂow network G D .V; E/ with vertex capacities into an equiv-

alent ﬂow network G 0 D .V 0 ; E 0 / without vertex capacities, such that a maximum

ﬂow in G 0

has the same value as a maximum ﬂow in G. How many vertices and

edges does G 0

have?

26.2 The Ford-Fulkerson method

This section presents the Ford-Fulkerson method for solving the maximum-ﬂow

problem. We call it a “method” rather than an “algorithm” because it encompasses

several implementations with differing running times. The Ford-Fulkerson method

depends on three important ideas that transcend the method and are relevant to

many ﬂow algorithms and problems: residual networks, augmenting paths, and

cuts. These ideas are essential to the important max-ﬂow min-cut theorem (The-

orem 26.6), which characterizes the value of a maximum ﬂow in terms of cuts of

26.2 The Ford-Fulkerson method 715

the ﬂow network. We end this section by presenting one speciﬁc implementation

of the Ford-Fulkerson method and analyzing its running time.

The Ford-Fulkerson method iteratively increases the value of the ﬂow. We start

with f .u; / D 0 for all u; 2 V , giving an initial ﬂow of value 0. At each

iteration, we increase the ﬂow value in G by ﬁnding an “augmenting path” in an

associated “residual network” G f . Once we know the edges of an augmenting

path in G f , we can easily identify speciﬁc edges in G for which we can change

the ﬂow so that we increase the value of the ﬂow. Although each iteration of the

Ford-Fulkerson method increases the value of the ﬂow, we shall see that the ﬂow

on any particular edge of G may increase or decrease; decreasing the ﬂow on some

edges may be necessary in order to enable an algorithm to send more ﬂow from the

source to the sink. We repeatedly augment the ﬂow until the residual network has

no more augmenting paths. The max-ﬂow min-cut theorem will show that upon

termination, this process yields a maximum ﬂow.

FORD-FULKERSON-METHOD.G; s; t/

1 initialize ﬂow f to 0

2 while there exists an augmenting path p in the residual network G f

3 augment ﬂow f along p

4 return f

In order to implement and analyze the Ford-Fulkerson method, we need to intro-

duce several additional concepts.

Residual networks

Intuitively, given a ﬂow network G and a ﬂow f , the residual network G f consists

of edges with capacities that represent how we can change the ﬂow on edges of G.

An edge of the ﬂow network can admit an amount of additional ﬂow equal to the

edge’s capacity minus the ﬂow on that edge. If that value is positive, we place

that edge into G f with a “residual capacity” of c f .u; / D c.u; / f .u; /.

The only edges of G that are in G f are those that can admit more ﬂow; those

edges .u; / whose ﬂow equals their capacity have c f .u; / D 0, and they are not

in G f .

The residual network G f may also contain edges that are not in G, however.

As an algorithm manipulates the ﬂow, with the goal of increasing the total ﬂow, it

might need to decrease the ﬂow on a particular edge. In order to represent a pos-

sible decrease of a positive ﬂow f .u; / on an edge in G, we place an edge .; u/

into G f with residual capacity c f .; u/ D f .u; /—that is, an edge that can admit

ﬂow in the opposite direction to .u; /, at most canceling out the ﬂow on .u; /.

These reverse edges in the residual network allow an algorithm to send back ﬂow

716 Chapter 26 Maximum Flow

it has already sent along an edge. Sending ﬂow back along an edge is equiva-

lent to decreasing the ﬂow on the edge, which is a necessary operation in many

algorithms.

More formally, suppose that we have a ﬂow network G D .V; E/ with source s

and sink t. Let f be a ﬂow in G, and consider a pair of vertices u; 2 V . We

deﬁne the residual capacity c f .u; / by

c f .u; / D

c.u; / f .u; / if .u; / 2 E ;

f .; u/ if .; u/ 2 E ;

0 otherwise :

(26.2)

Because of our assumption that .u; / 2 E implies .; u/ 62 E, exactly one case in

equation (26.2) applies to each ordered pair of vertices.

As an example of equation (26.2), if c.u; / D 16 and f .u; / D 11, then we

can increase f .u; / by up to c f .u; / D 5 units before we exceed the capacity

constraint on edge .u; /. We also wish to allow an algorithm to return up to 11

units of ﬂow from to u, and hence c f .; u/ D 11.

Given a ﬂow network G D .V; E/ and a ﬂow f , the residual network of G

induced by f is G f D .V; E f /, where

E f D f.u; / 2 V V W c f .u; / > 0g : (26.3)

That is, as promised above, each edge of the residual network, or residual edge,

can admit a ﬂow that is greater than 0. Figure 26.4(a) repeats the ﬂow network G

and ﬂow f of Figure 26.1(b), and Figure 26.4(b) shows the corresponding residual

network G f . The edges in E f are either edges in E or their reversals, and thus

jE f j 2 jEj :

Observe that the residual network G f is similar to a ﬂow network with capacities

given by c f . It does not satisfy our deﬁnition of a ﬂow network because it may

contain both an edge .u; / and its reversal .; u/. Other than this difference, a

residual network has the same properties as a ﬂow network, and we can deﬁne a

ﬂow in the residual network as one that satisﬁes the deﬁnition of a ﬂow, but with

respect to capacities c f in the network G f .

A ﬂow in a residual network provides a roadmap for adding ﬂow to the original

ﬂow network. If f is a ﬂow in G and f 0

is a ﬂow in the corresponding residual

network G f , we deﬁne f " f 0

, the augmentation of ﬂow f by f 0

, to be a function

from V V to R , deﬁned by

.f " f

0

/.u; / D

(

f .u; / C f 0 .u; / f 0 .; u/ if .u; / 2 E ;

0 otherwise :

(26.4)

26.2 The Ford-Fulkerson method 717

9

15

s t

5

12

5

7

5

3

1

8

11

4

s t

11/16

12/12

19/20

7/7

9

1/4

12/13

11/14

4/4

(b)

(c)

11

5

3

4

s t

11/16

12/12

15/20

7/7

4/9

1/4

8/13

11/14

4/4

(d)

19

s t

5

12

1

7

3

1

12

11

4

11

1

3

v

1

v

1

v

1

v

1

v

2

v

2

v

2

v

2

v

3

v

3

v

3

v

3

v

4

v

4

v

4

v

4

(a)

Figure 26.4 (a) The ﬂow network G and ﬂow f of Figure 26.1(b). (b) The residual network Gf

with augmenting path p shaded; its residual capacity is cf .p/ D cf .2; 3/ D 4. Edges with

residual capacity equal to 0, such as .1; 3/, are not shown, a convention we follow in the remainder

of this section. (c) The ﬂow in G that results from augmenting along path p by its residual capacity 4.

Edges carrying no ﬂow, such as .3; 2/, are labeled only by their capacity, another convention we

follow throughout. (d) The residual network induced by the ﬂow in (c).

The intuition behind this deﬁnition follows the deﬁnition of the residual network.

We increase the ﬂow on .u; / by f 0 .u; / but decrease it by f 0 .; u/ because

pushing ﬂow on the reverse edge in the residual network signiﬁes decreasing the

ﬂow in the original network. Pushing ﬂow on the reverse edge in the residual

network is also known as cancellation. For example, if we send 5 crates of hockey

pucks from u to and send 2 crates from to u, we could equivalently (from the

perspective of the ﬁnal result) just send 3 creates from u to and none from to u.

Cancellation of this type is crucial for any maximum-ﬂow algorithm.

Lemma 26.1

Let G D .V; E/ be a ﬂow network with source s and sink t, and let f be a ﬂow

in G. Let G f be the residual network of G induced by f , and let f 0

be a ﬂow

in G f . Then the function f " f 0

deﬁned in equation (26.4) is a ﬂow in G with

value jf " f 0 j D jf j C jf 0 j.

Proof We ﬁrst verify that f " f 0

obeys the capacity constraint for each edge in E

and ﬂow conservation at each vertex in V fs; tg.

718 Chapter 26 Maximum Flow

For the capacity constraint, ﬁrst observe that if .u; / 2 E, then c f .; u/ D

f .u; /. Therefore, we have f 0 .; u/ c f .; u/ D f .u; /, and hence

.f " f

0

/.u; / D f .u; / C f

0

.u; / f

0

.; u/ (by equation (26.4))

f .u; / C f

0

.u; / f .u; / (because f 0 .; u/ f .u; /)

D f

0

.u; /

0 :

In addition,

.f " f

0

/.u; /

D f .u; / C f

0

.u; / f

0

.; u/ (by equation (26.4))

f .u; / C f

0

.u; / (because ﬂows are nonnegative)

f .u; / C c f .u; / (capacity constraint)

D f .u; / C c.u; / f .u; / (deﬁnition of c f )

D c.u; / :

For ﬂow conservation, because both f and f 0

obey ﬂow conservation, we have

that for all u 2 V fs; tg,

X

2V

.f " f

0

/.u; / D

X

2V

.f .u; / C f

0

.u; / f

0

.; u//

D

X

2V

f .u; / C

X

2V

f

0

.u; /

X

2V

f

0

.; u/

D

X

2V

f .; u/ C

X

2V

f

0

.; u/

X

2V

f

0

.u; /

D

X

2V

.f .; u/ C f

0

.; u/ f

0

.u; //

D

X

2V

.f " f

0

/.; u/ ;

where the third line follows from the second by ﬂow conservation.

Finally, we compute the value of f " f 0

. Recall that we disallow antiparallel

edges in G (but not in G f ), and hence for each vertex 2 V , we know that there

can be an edge .s; / or .; s/, but never both. We deﬁne V 1 D f W .s; / 2 Eg

to be the set of vertices with edges from s, and V 2 D f W .; s/ 2 Eg to be the

set of vertices with edges to s. We have V 1 [ V 2 V and, because we disallow

antiparallel edges, V 1 \ V 2 D ;. We now compute

jf " f

0

j D

X

2V

.f " f

0

/ .s; /

X

2V

.f " f

0

/ .; s/

D

X

2V 1

.f " f

0

/ .s; /

X

2V 2

.f " f

0

/ .; s/ ; (26.5)

26.2 The Ford-Fulkerson method 719

where the second line follows because .f " f 0 /.w; x/ is 0 if .w; x/ 62 E. We now

apply the deﬁnition of f " f 0

to equation (26.5), and then reorder and group terms

to obtain

jf " f

0

j

D

X

2V 1

.f .s; / C f

0

.s; / f

0

.; s//

X

2V 2

.f .; s/ C f

0

.; s/ f

0

.s; //

D

X

2V 1

f .s; / C

X

2V 1

f

0

.s; /

X

2V 1

f

0

.; s/

X

2V 2

f .; s/

X

2V 2

f

0

.; s/ C

X

2V 2

f

0

.s; /

D

X

2V 1

f .s; /

X

2V 2

f .; s/

C

X

2V 1

f

0

.s; / C

X

2V 2

f

0

.s; /

X

2V 1

f

0

.; s/

X

2V 2

f

0

.; s/

D

X

2V 1

f .s; /

X

2V 2

f .; s/ C

X

2V 1 [V 2

f

0

.s; /

X

2V 1 [V 2

f

0

.; s/ : (26.6)

In equation (26.6), we can extend all four summations to sum over V , since each

additional term has value 0. (Exercise 26.2-1 asks you to prove this formally.) We

thus have

jf " f

0

j D

X

2V

f .s; /

X

2V

f .; s/ C

X

2V

f

0

.s; /

X

2V

f

0

.; s/ (26.7)

D jf j C jf

0

j :

Augmenting paths

Given a ﬂow network G D .V; E/ and a ﬂow f , an augmenting path p is a

simple path from s to t in the residual network G f . By the deﬁnition of the resid-

ual network, we may increase the ﬂow on an edge .u; / of an augmenting path

by up to c f .u; / without violating the capacity constraint on whichever of .u; /

and .; u/ is in the original ﬂow network G.

The shaded path in Figure 26.4(b) is an augmenting path. Treating the residual

network G f in the ﬁgure as a ﬂow network, we can increase the ﬂow through each

edge of this path by up to 4 units without violating a capacity constraint, since the

smallest residual capacity on this path is c f . 2 ; 3 / D 4. We call the maximum

amount by which we can increase the ﬂow on each edge in an augmenting path p

the residual capacity of p, given by

c f .p/ D min fc f .u; / W .u; / is on pg :

720 Chapter 26 Maximum Flow

The following lemma, whose proof we leave as Exercise 26.2-7, makes the above

argument more precise.

Lemma 26.2

Let G D .V; E/ be a ﬂow network, let f be a ﬂow in G, and let p be an augmenting

path in G f . Deﬁne a function f p W V V ! R by

f p .u; / D

(

c f .p/ if .u; / is on p ;

0 otherwise :

(26.8)

Then, f p is a ﬂow in G f with value jf p j D c f .p/ > 0.

The following corollary shows that if we augment f by f p , we get another ﬂow

in G whose value is closer to the maximum. Figure 26.4(c) shows the result of

augmenting the ﬂow f from Figure 26.4(a) by the ﬂow f p in Figure 26.4(b), and

Figure 26.4(d) shows the ensuing residual network.

Corollary 26.3

Let G D .V; E/ be a ﬂow network, let f be a ﬂow in G, and let p be an

augmenting path in G f . Let f p be deﬁned as in equation (26.8), and suppose

that we augment f by f p . Then the function f " f p is a ﬂow in G with value

jf " f p j D jf j C jf p j > jf j.

Proof Immediate from Lemmas 26.1 and 26.2.

Cuts of ﬂow networks

The Ford-Fulkerson method repeatedly augments the ﬂow along augmenting paths

until it has found a maximum ﬂow. How do we know that when the algorithm

terminates, we have actually found a maximum ﬂow? The max-ﬂow min-cut theo-

rem, which we shall prove shortly, tells us that a ﬂow is maximum if and only if its

residual network contains no augmenting path. To prove this theorem, though, we

must ﬁrst explore the notion of a cut of a ﬂow network.

A cut .S; T / of ﬂow network G D .V; E/ is a partition of V into S and

T D V S such that s 2 S and t 2 T . (This deﬁnition is similar to the def-

inition of “cut” that we used for minimum spanning trees in Chapter 23, except

that here we are cutting a directed graph rather than an undirected graph, and we

insist that s 2 S and t 2 T .) If f is a ﬂow, then the net ﬂow f .S; T / across the

cut .S; T / is deﬁned to be

f .S; T / D

X

u2S

X

2T

f .u; /

X

u2S

X

2T

f .; u/ : (26.9)

26.2 The Ford-Fulkerson method 721

s t11/16

12/12

15/20

7/7

4/9

1/4

8/13

11/14

4/4

S T

v

4

v

3

v

1

v

2

Figure 26.5 A cut .S; T / in the ﬂow network of Figure 26.1(b), where S D fs; 1; 2g and

T D f3; 4; tg. The vertices in S are black, and the vertices in T are white. The net ﬂow

across .S; T / is f .S; T / D 19, and the capacity is c.S; T / D 26.

The capacity of the cut .S; T / is

c.S; T / D

X

u2S

X

2T

c.u; / : (26.10)

A minimum cut of a network is a cut whose capacity is minimum over all cuts of

the network.

The asymmetry between the deﬁnitions of ﬂow and capacity of a cut is inten-

tional and important. For capacity, we count only the capacities of edges going

from S to T , ignoring edges in the reverse direction. For ﬂow, we consider the

ﬂow going from S to T minus the ﬂow going in the reverse direction from T to S.

The reason for this difference will become clear later in this section.

Figure 26.5 shows the cut .fs; 1 ; 2 g ; f 3 ; 4 ; tg/ in the ﬂow network of Fig-

ure 26.1(b). The net ﬂow across this cut is

f . 1 ; 3 / C f . 2 ; 4 / f . 3 ; 2 / D 12 C 11 4

D 19 ;

and the capacity of this cut is

c. 1 ; 3 / C c. 2 ; 4 / D 12 C 14

D 26 :

The following lemma shows that, for a given ﬂow f , the net ﬂow across any cut

is the same, and it equals jf j, the value of the ﬂow.

Lemma 26.4

Let f be a ﬂow in a ﬂow network G with source s and sink t, and let .S; T / be any

cut of G. Then the net ﬂow across .S; T / is f .S; T / D jf j.

722 Chapter 26 Maximum Flow

Proof We can rewrite the ﬂow-conservation condition for any node u 2 V fs; tg

as

X

2V

f .u; /

X

2V

f .; u/ D 0 : (26.11)

Taking the deﬁnition of jf j from equation (26.1) and adding the left-hand side of

equation (26.11), which equals 0, summed over all vertices in S fsg, gives

jf j D

X

2V

f .s; /

X

2V

f .; s/ C

X

u2Sfsg

X

2V

f .u; /

X

2V

f .; u/

!

:

Expanding the right-hand summation and regrouping terms yields

jf j D

X

2V

f .s; /

X

2V

f .; s/ C

X

u2Sfsg

X

2V

f .u; /

X

u2Sfsg

X

2V

f .; u/

D

X

2V

f .s; / C

X

u2Sfsg

f .u; /

!

X

2V

f .; s/ C

X

u2Sfsg

f .; u/

!

D

X

2V

X

u2S

f .u; /

X

2V

X

u2S

f .; u/ :

Because V D S [ T and S \ T D ;, we can split each summation over V into

summations over S and T to obtain

jf j D

X

2S

X

u2S

f .u; / C

X

2T

X

u2S

f .u; /

X

2S

X

u2S

f .; u/

X

2T

X

u2S

f .; u/

D

X

2T

X

u2S

f .u; /

X

2T

X

u2S

f .; u/

C

X

2S

X

u2S

f .u; /

X

2S

X

u2S

f .; u/

!

:

The two summations within the parentheses are actually the same, since for all

vertices x; y 2 V , the term f .x; y/ appears once in each summation. Hence, these

summations cancel, and we have

jf j D

X

u2S

X

2T

f .u; /

X

u2S

X

2T

f .; u/

D f .S; T / :

A corollary to Lemma 26.4 shows how we can use cut capacities to bound the

value of a ﬂow.

26.2 The Ford-Fulkerson method 723

Corollary 26.5

The value of any ﬂow f in a ﬂow network G is bounded from above by the capacity

of any cut of G.

Proof Let .S; T / be any cut of G and let f be any ﬂow. By Lemma 26.4 and the

capacity constraint,

jf j D f .S; T /

D

X

u2S

X

2T

f .u; /

X

u2S

X

2T

f .; u/

X

u2S

X

2T

f .u; /

X

u2S

X

2T

c.u; /

D c.S; T / :

Corollary 26.5 yields the immediate consequence that the value of a maximum

ﬂow in a network is bounded from above by the capacity of a minimum cut of

the network. The important max-ﬂow min-cut theorem, which we now state and

prove, says that the value of a maximum ﬂow is in fact equal to the capacity of a

minimum cut.

Theorem 26.6 (Max-ﬂow min-cut theorem)

If f is a ﬂow in a ﬂow network G D .V; E/ with source s and sink t, then the

following conditions are equivalent:

1. f is a maximum ﬂow in G.

2. The residual network G f contains no augmenting paths.

3. jf j D c.S; T / for some cut .S; T / of G.

Proof .1/ ) .2/: Suppose for the sake of contradiction that f is a maximum

ﬂow in G but that G f has an augmenting path p. Then, by Corollary 26.3, the

ﬂow found by augmenting f by f p , where f p is given by equation (26.8), is a ﬂow

in G with value strictly greater than jf j, contradicting the assumption that f is a

maximum ﬂow.

.2/ ) .3/: Suppose that G f has no augmenting path, that is, that G f contains

no path from s to t. Deﬁne

S D f 2 V W there exists a path from s to in G f g

and T D V S. The partition .S; T / is a cut: we have s 2 S trivially and t 62 S

because there is no path from s to t in G f . Now consider a pair of vertices

724 Chapter 26 Maximum Flow

u 2 S and 2 T . If .u; / 2 E, we must have f .u; / D c.u; /, since

otherwise .u; / 2 E f , which would place in set S. If .; u/ 2 E, we must

have f .; u/ D 0, because otherwise c f .u; / D f .; u/ would be positive and

we would have .u; / 2 E f , which would place in S. Of course, if neither .u; /

nor .; u/ is in E, then f .u; / D f .; u/ D 0. We thus have

f .S; T / D

X

u2S

X

2T

f .u; /

X

2T

X

u2S

f .; u/

D

X

u2S

X

2T

c.u; /

X

2T

X

u2S

0

D c.S; T / :

By Lemma 26.4, therefore, jf j D f .S; T / D c.S; T /.

.3/ ) .1/: By Corollary 26.5, jf j c.S; T / for all cuts .S; T /. The condition

jf j D c.S; T / thus implies that f is a maximum ﬂow.

The basic Ford-Fulkerson algorithm

In each iteration of the Ford-Fulkerson method, we ﬁnd some augmenting path p

and use p to modify the ﬂow f . As Lemma 26.2 and Corollary 26.3 suggest, we

replace f by f " f p , obtaining a new ﬂow whose value is jf j C jf p j. The follow-

ing implementation of the method computes the maximum ﬂow in a ﬂow network

G D .V; E/ by updating the ﬂow attribute .u; /:f for each edge .u; / 2 E. 1

If .u; / 62 E, we assume implicitly that .u; /:f D 0. We also assume that we

are given the capacities c.u; / along with the ﬂow network, and c.u; / D 0

if .u; / 62 E. We compute the residual capacity c f .u; / in accordance with the

formula (26.2). The expression c f .p/ in the code is just a temporary variable that

stores the residual capacity of the path p.

FORD-FULKERSON.G; s; t/

1 for each edge .u; / 2 G:E

2 .u; /:f D 0

3 while there exists a path p from s to t in the residual network G f

4 c f .p/ D min fc f .u; / W .u; / is in pg

5 for each edge .u; / in p

6 if .u; / 2 E

7 .u; /:f D .u; /:f C c f .p/

8 else .; u/:f D .; u/:f c f .p/

1

Recall from Section 22.1 that we represent an attribute f for edge .u; / with the same style of

notation—.u; /:f —that we use for an attribute of any other object.

26.2 The Ford-Fulkerson method 725

The FORD-FULKERSON algorithm simply expands on the FORD-FULKERSON-

METHOD pseudocode given earlier. Figure 26.6 shows the result of each iteration

in a sample run. Lines 1–2 initialize the ﬂow f to 0. The while loop of lines 3–8

repeatedly ﬁnds an augmenting path p in G f and augments ﬂow f along p by

the residual capacity c f .p/. Each residual edge in path p is either an edge in the

original network or the reversal of an edge in the original network. Lines 6–8

update the ﬂow in each case appropriately, adding ﬂow when the residual edge is

an original edge and subtracting it otherwise. When no augmenting paths exist, the

ﬂow f is a maximum ﬂow.

Analysis of Ford-Fulkerson

The running time of FORD-FULKERSON depends on how we ﬁnd the augmenting

path p in line 3. If we choose it poorly, the algorithm might not even terminate: the

value of the ﬂow will increase with successive augmentations, but it need not even

converge to the maximum ﬂow value. 2 If we ﬁnd the augmenting path by using a

breadth-ﬁrst search (which we saw in Section 22.2), however, the algorithm runs in

polynomial time. Before proving this result, we obtain a simple bound for the case

in which we choose the augmenting path arbitrarily and all capacities are integers.

In practice, the maximum-ﬂow problem often arises with integral capacities. If

the capacities are rational numbers, we can apply an appropriate scaling transfor-

mation to make them all integral. If f

denotes a maximum ﬂow in the transformed

network, then a straightforward implementation of FORD-FULKERSON executes

the while loop of lines 3–8 at most jf j times, since the ﬂow value increases by at

least one unit in each iteration.

We can perform the work done within the while loop efﬁciently if we implement

the ﬂow network G D .V; E/ with the right data structure and ﬁnd an augmenting

path by a linear-time algorithm. Let us assume that we keep a data structure cor-

responding to a directed graph G 0 D .V; E 0 /, where E 0 D f.u; / W .u; / 2 E or

.; u/ 2 Eg. Edges in the network G are also edges in G 0

, and therefore we can

easily maintain capacities and ﬂows in this data structure. Given a ﬂow f on G,

the edges in the residual network G f consist of all edges .u; / of G 0

such that

c f .u; / > 0, where c f conforms to equation (26.2). The time to ﬁnd a path in

a residual network is therefore O.V C E 0 / D O.E/ if we use either depth-ﬁrst

search or breadth-ﬁrst search. Each iteration of the while loop thus takes O.E/

time, as does the initialization in lines 1–2, making the total running time of the

FORD-FULKERSON algorithm O.E jf j/.

2

The Ford-Fulkerson method might fail to terminate only if edge capacities are irrational numbers.

726 Chapter 26 Maximum Flow

12

4

4 4/4 4

v

1

4

16

4

10

s t

16

12

20

7

9

4

13

14

4v

1

s t

4/16

4/12

20

7

4/9

13

4/14

4/4

s t

7

5

4

4

v

1

8

4

13

20

v

1

s t

4/16

8/12

4/20

7

4/9

4/13

4/14

4/4

4

10

s t

7

5

8

4

v

1

4

9

v

1

s t

8/16

8/12

8/20

7

9

4/13

4/14

4/4

v

2

v

2

v

2

v

2

v

2

v

2

v

3

v

3

v

3

v

3

v

3

v

3

v

4

v

4

v

4

v

4

v

4

v

4

(b)

(a)

(c)

12

4 4

4

4

4

Figure 26.6 The execution of the basic Ford-Fulkerson algorithm. (a)–(e) Successive iterations of

the while loop. The left side of each part shows the residual network Gf from line 3 with a shaded

augmenting path p. The right side of each part shows the new ﬂow f that results from augmenting f

by fp. The residual network in (a) is the input network G.

When the capacities are integral and the optimal ﬂow value jf j is small, the

running time of the Ford-Fulkerson algorithm is good. Figure 26.7(a) shows an ex-

ample of what can happen on a simple ﬂow network for which jf j is large. A max-

imum ﬂow in this network has value 2,000,000: 1,000,000 units of ﬂow traverse

the path s ! u ! t, and another 1,000,000 units traverse the path s ! ! t. If

the ﬁrst augmenting path found by FORD-FULKERSON is s ! u ! ! t, shown

in Figure 26.7(a), the ﬂow has value 1 after the ﬁrst iteration. The resulting resid-

ual network appears in Figure 26.7(b). If the second iteration ﬁnds the augment-

ing path s ! ! u ! t, as shown in Figure 26.7(b), the ﬂow then has value 2.

Figure 26.7(c) shows the resulting residual network. We can continue, choosing

the augmenting path s ! u ! ! t in the odd-numbered iterations and the aug-

menting path s ! ! u ! t in the even-numbered iterations. We would perform

a total of 2,000,000 augmentations, increasing the ﬂow value by only 1 unit in each.

26.2 The Ford-Fulkerson method 727

4

12

11

2

11

2

8

8

9

4

4

9

8

4

4

9

8

s t

12

7

4

4v

1

s t

8/16

8/12

15/20

7/7

9

11/13

11/14

4/4

v

1

10

19

s t

12

1

7

11

4

3

v

2

v

3

v

3

v

3

v

4

v

4

v

4

(d)

(f)

4

9

8

4

4

15

s t

5

7

11

4v

1

s t

12/16

12/12

19/20

7/7

9

11/13

11/14

4/4

v

1

3

v

2

v

3

v

3

v

4

v

4

(e)

4

v

2

v

2

v

1

v

2

8

8

Figure 26.6, continued (f) The residual network at the last while loop test. It has no augmenting

paths, and the ﬂow f shown in (e) is therefore a maximum ﬂow. The value of the maximum ﬂow

found is 23.

The Edmonds-Karp algorithm

We can improve the bound on FORD-FULKERSON by ﬁnding the augmenting

path p in line 3 with a breadth-ﬁrst search. That is, we choose the augmenting

path as a shortest path from s to t in the residual network, where each edge has

unit distance (weight). We call the Ford-Fulkerson method so implemented the

Edmonds-Karp algorithm. We now prove that the Edmonds-Karp algorithm runs

in O.VE 2 / time.

The analysis depends on the distances to vertices in the residual network G f .

The following lemma uses the notation ı f .u; / for the shortest-path distance

from u to in G f , where each edge has unit distance.

Lemma 26.7

If the Edmonds-Karp algorithm is run on a ﬂow network G D .V; E/ with source s

and sink t, then for all vertices 2 V fs; tg, the shortest-path distance ı f .s; /

in the residual network G f increases monotonically with each ﬂow augmentation.

728 Chapter 26 Maximum Flow

1

999,999

999,999

1

s t

1,000,000

1,000,000

1

1,000,000

1,000,000

999,999

1

1

999,999

u

v

s t

1,000,000

1

1,000,000

u

v999,999

1

999,999

1

s t

1

u

v

(a) (b) (c)

Figure 26.7 (a) A ﬂow network for which FORD-FULKERSON can take ‚.E jf

j/ time,

where f

is a maximum ﬂow, shown here with jf

j D 2,000,000. The shaded path is an aug-

menting path with residual capacity 1. (b) The resulting residual network, with another augmenting

path whose residual capacity is 1. (c) The resulting residual network.

Proof We will suppose that for some vertex 2 V fs; tg, there is a ﬂow aug-

mentation that causes the shortest-path distance from s to to decrease, and then

we will derive a contradiction. Let f be the ﬂow just before the ﬁrst augmentation

that decreases some shortest-path distance, and let f 0

be the ﬂow just afterward.

Let be the vertex with the minimum ı f 0 .s; / whose distance was decreased by

the augmentation, so that ı f 0 .s; / < ı f .s; /. Let p D s u ! be a shortest

path from s to in G f 0 , so that .u; / 2 E f 0 and

ı f 0 .s; u/ D ı f 0 .s; / 1 : (26.12)

Because of how we chose , we know that the distance of vertex u from the source s

did not decrease, i.e.,

ı f 0 .s; u/ ı f .s; u/ : (26.13)

We claim that .u; / 62 E f . Why? If we had .u; / 2 E f , then we would also have

ı f .s; / ı f .s; u/ C 1 (by Lemma 24.10, the triangle inequality)

ı f 0 .s; u/ C 1 (by inequality (26.13))

D ı f 0 .s; / (by equation (26.12)) ,

which contradicts our assumption that ı f 0 .s; / < ı f .s; /.

How can we have .u; / 62 E f and .u; / 2 E f 0 ? The augmentation must

have increased the ﬂow from to u. The Edmonds-Karp algorithm always aug-

ments ﬂow along shortest paths, and therefore the shortest path from s to u in G f

has .; u/ as its last edge. Therefore,

ı f .s; / D ı f .s; u/ 1

ı f 0 .s; u/ 1 (by inequality (26.13))

D ı f 0 .s; / 2 (by equation (26.12)) ,

26.2 The Ford-Fulkerson method 729

which contradicts our assumption that ı f 0 .s; / < ı f .s; /. We conclude that our

assumption that such a vertex exists is incorrect.

The next theorem bounds the number of iterations of the Edmonds-Karp algo-

rithm.

Theorem 26.8

If the Edmonds-Karp algorithm is run on a ﬂow network G D .V; E/ with source s

and sink t, then the total number of ﬂow augmentations performed by the algorithm

is O.VE/.

Proof We say that an edge .u; / in a residual network G f is critical on an aug-

menting path p if the residual capacity of p is the residual capacity of .u; /, that

is, if c f .p/ D c f .u; /. After we have augmented ﬂow along an augmenting path,

any critical edge on the path disappears from the residual network. Moreover, at

least one edge on any augmenting path must be critical. We will show that each of

the jEj edges can become critical at most jV j =2 times.

Let u and be vertices in V that are connected by an edge in E. Since augment-

ing paths are shortest paths, when .u; / is critical for the ﬁrst time, we have

ı f .s; / D ı f .s; u/ C 1 :

Once the ﬂow is augmented, the edge .u; / disappears from the residual network.

It cannot reappear later on another augmenting path until after the ﬂow from u to

is decreased, which occurs only if .; u/ appears on an augmenting path. If f 0

is

the ﬂow in G when this event occurs, then we have

ı f 0 .s; u/ D ı f 0 .s; / C 1 :

Since ı f .s; / ı f 0 .s; / by Lemma 26.7, we have

ı f 0 .s; u/ D ı f 0 .s; / C 1

ı f .s; / C 1

D ı f .s; u/ C 2 :

Consequently, from the time .u; / becomes critical to the time when it next

becomes critical, the distance of u from the source increases by at least 2. The

distance of u from the source is initially at least 0. The intermediate vertices on a

shortest path from s to u cannot contain s, u, or t (since .u; / on an augmenting

path implies that u ¤ t). Therefore, until u becomes unreachable from the source,

if ever, its distance is at most jV j 2. Thus, after the ﬁrst time that .u; / becomes

critical, it can become critical at most .jV j 2/=2 D jV j =2 1 times more, for a

total of at most jV j =2 times. Since there are O.E/ pairs of vertices that can have an

edge between them in a residual network, the total number of critical edges during

730 Chapter 26 Maximum Flow

the entire execution of the Edmonds-Karp algorithm is O.VE/. Each augmenting

path has at least one critical edge, and hence the theorem follows.

Because we can implement each iteration of FORD-FULKERSON in O.E/ time

when we ﬁnd the augmenting path by breadth-ﬁrst search, the total running time of

the Edmonds-Karp algorithm is O.VE 2 /. We shall see that push-relabel algorithms

can yield even better bounds. The algorithm of Section 26.4 gives a method for

achieving an O.V 2 E/ running time, which forms the basis for the O.V 3 /-time

algorithm of Section 26.5.

Exercises

26.2-1

Prove that the summations in equation (26.6) equal the summations in equa-

tion (26.7).

26.2-2

In Figure 26.1(b), what is the ﬂow across the cut .fs; 2 ; 4 g ; f 1 ; 3 ; tg/? What is

the capacity of this cut?

26.2-3

Show the execution of the Edmonds-Karp algorithm on the ﬂow network of Fig-

ure 26.1(a).

26.2-4

In the example of Figure 26.6, what is the minimum cut corresponding to the max-

imum ﬂow shown? Of the augmenting paths appearing in the example, which one

cancels ﬂow?

26.2-5

Recall that the construction in Section 26.1 that converts a ﬂow network with mul-

tiple sources and sinks into a single-source, single-sink network adds edges with

inﬁnite capacity. Prove that any ﬂow in the resulting network has a ﬁnite value

if the edges of the original network with multiple sources and sinks have ﬁnite

capacity.

26.2-6

Suppose that each source s i in a ﬂow network with multiple sources and sinks

produces exactly p i units of ﬂow, so that

P

2V

f .s i ; / D p i . Suppose also

that each sink t j consumes exactly q j units, so that

P

2V

f .; t j / D q j , where P

i

p i D

P

j

q j . Show how to convert the problem of ﬁnding a ﬂow f that obeys

26.2 The Ford-Fulkerson method 731

these additional constraints into the problem of ﬁnding a maximum ﬂow in a single-

source, single-sink ﬂow network.

26.2-7

Prove Lemma 26.2.

26.2-8

Suppose that we redeﬁne the residual network to disallow edges into s. Argue that

the procedure FORD-FULKERSON still correctly computes a maximum ﬂow.

26.2-9

Suppose that both f and f 0

are ﬂows in a network G and we compute ﬂow f " f 0

.

Does the augmented ﬂow satisfy the ﬂow conservation property? Does it satisfy

the capacity constraint?

26.2-10

Show how to ﬁnd a maximum ﬂow in a network G D .V; E/ by a sequence of at

most jEj augmenting paths. (Hint: Determine the paths after ﬁnding the maximum

ﬂow.)

26.2-11

The edge connectivity of an undirected graph is the minimum number k of edges

that must be removed to disconnect the graph. For example, the edge connectivity

of a tree is 1, and the edge connectivity of a cyclic chain of vertices is 2. Show

how to determine the edge connectivity of an undirected graph G D .V; E/ by

running a maximum-ﬂow algorithm on at most jV j ﬂow networks, each having

O.V / vertices and O.E/ edges.

26.2-12

Suppose that you are given a ﬂow network G, and G has edges entering the

source s. Let f be a ﬂow in G in which one of the edges .; s/ entering the source

has f .; s/ D 1. Prove that there must exist another ﬂow f 0

with f 0 .; s/ D 0

such that jf j D jf 0 j. Give an O.E/-time algorithm to compute f 0

, given f , and

assuming that all edge capacities are integers.

26.2-13

Suppose that you wish to ﬁnd, among all minimum cuts in a ﬂow network G with

integral capacities, one that contains the smallest number of edges. Show how to

modify the capacities of G to create a new ﬂow network G 0

in which any minimum

cut in G 0

is a minimum cut with the smallest number of edges in G.

732 Chapter 26 Maximum Flow

26.3 Maximum bipartite matching

Some combinatorial problems can easily be cast as maximum-ﬂow problems. The

multiple-source, multiple-sink maximum-ﬂow problem from Section 26.1 gave us

one example. Some other combinatorial problems seem on the surface to have little

to do with ﬂow networks, but can in fact be reduced to maximum-ﬂow problems.

This section presents one such problem: ﬁnding a maximum matching in a bipartite

graph. In order to solve this problem, we shall take advantage of an integrality

property provided by the Ford-Fulkerson method. We shall also see how to use

the Ford-Fulkerson method to solve the maximum-bipartite-matching problem on

a graph G D .V; E/ in O.VE/ time.

The maximum-bipartite-matching problem

Given an undirected graph G D .V; E/, a matching is a subset of edges M E

such that for all vertices 2 V , at most one edge of M is incident on . We

say that a vertex 2 V is matched by the matching M if some edge in M is

incident on ; otherwise, is unmatched. A maximum matching is a matching

of maximum cardinality, that is, a matching M such that for any matching M 0

,

we have jM j jM 0 j. In this section, we shall restrict our attention to ﬁnding

maximum matchings in bipartite graphs: graphs in which the vertex set can be

partitioned into V D L [ R, where L and R are disjoint and all edges in E

go between L and R. We further assume that every vertex in V has at least one

incident edge. Figure 26.8 illustrates the notion of a matching in a bipartite graph.

The problem of ﬁnding a maximum matching in a bipartite graph has many

practical applications. As an example, we might consider matching a set L of ma-

chines with a set R of tasks to be performed simultaneously. We take the presence

of edge .u; / in E to mean that a particular machine u 2 L is capable of per-

forming a particular task 2 R. A maximum matching provides work for as many

machines as possible.

Finding a maximum bipartite matching

We can use the Ford-Fulkerson method to ﬁnd a maximum matching in an undi-

rected bipartite graph G D .V; E/ in time polynomial in jV j and jEj. The trick is

to construct a ﬂow network in which ﬂows correspond to matchings, as shown in

Figure 26.8(c). We deﬁne the corresponding ﬂow network G 0 D .V 0 ; E 0 / for the

bipartite graph G as follows. We let the source s and sink t be new vertices not

in V , and we let V 0 D V [ fs; tg. If the vertex partition of G is V D L [ R, the

26.3 Maximum bipartite matching 733

L R L R

s t

(a) (c)

L R

(b)

Figure 26.8 A bipartite graph G D .V; E/ with vertex partition V D L [ R. (a) A matching

with cardinality 2, indicated by shaded edges. (b) A maximum matching with cardinality 3. (c) The

corresponding ﬂow network G

0

with a maximum ﬂow shown. Each edge has unit capacity. Shaded

edges have a ﬂow of 1, and all other edges carry no ﬂow. The shaded edges from L to R correspond

to those in the maximum matching from (b).

directed edges of G 0

are the edges of E, directed from L to R, along with jV j new

directed edges:

E

0

D f.s; u/ W u 2 Lg [ f.u; / W .u; / 2 Eg [ f.; t/ W 2 Rg :

To complete the construction, we assign unit capacity to each edge in E 0

. Since

each vertex in V has at least one incident edge, jEj jV j =2. Thus, jEj jE 0 j D

jEj C jV j 3 jEj, and so jE 0 j D ‚.E/.

The following lemma shows that a matching in G corresponds directly to a ﬂow

in G’s corresponding ﬂow network G 0

. We say that a ﬂow f on a ﬂow network

G D .V; E/ is integer-valued if f .u; / is an integer for all .u; / 2 V V .

Lemma 26.9

Let G D .V; E/ be a bipartite graph with vertex partition V D L [ R, and let

G 0 D .V 0 ; E 0 / be its corresponding ﬂow network. If M is a matching in G, then

there is an integer-valued ﬂow f in G 0

with value jf j D jM j. Conversely, if f

is an integer-valued ﬂow in G 0

, then there is a matching M in G with cardinality

jM j D jf j.

Proof We ﬁrst show that a matching M in G corresponds to an integer-valued

ﬂow f in G 0

. Deﬁne f as follows. If .u; / 2 M , then f .s; u/ D f .u; / D

f .; t/ D 1. For all other edges .u; / 2 E 0

, we deﬁne f .u; / D 0. It is simple

to verify that f satisﬁes the capacity constraint and ﬂow conservation.

734 Chapter 26 Maximum Flow

Intuitively, each edge .u; / 2 M corresponds to one unit of ﬂow in G 0

that

traverses the path s ! u ! ! t. Moreover, the paths induced by edges in M

are vertex-disjoint, except for s and t. The net ﬂow across cut .L [ fsg ; R [ ftg/

is equal to jM j; thus, by Lemma 26.4, the value of the ﬂow is jf j D jM j.

To prove the converse, let f be an integer-valued ﬂow in G 0

, and let

M D f.u; / W u 2 L; 2 R; and f .u; / > 0g :

Each vertex u 2 L has only one entering edge, namely .s; u/, and its capacity

is 1. Thus, each u 2 L has at most one unit of ﬂow entering it, and if one unit of

ﬂow does enter, by ﬂow conservation, one unit of ﬂow must leave. Furthermore,

since f is integer-valued, for each u 2 L, the one unit of ﬂow can enter on at most

one edge and can leave on at most one edge. Thus, one unit of ﬂow enters u if and

only if there is exactly one vertex 2 R such that f .u; / D 1, and at most one

edge leaving each u 2 L carries positive ﬂow. A symmetric argument applies to

each 2 R. The set M is therefore a matching.

To see that jM j D jf j, observe that for every matched vertex u 2 L, we have

f .s; u/ D 1, and for every edge .u; / 2 E M , we have f .u; / D 0. Conse-

quently, f .L [ fsg ; R [ ftg/, the net ﬂow across cut .L [ fsg ; R [ ftg/, is equal

to jM j. Applying Lemma 26.4, we have that jf j D f .L[fsg ; R[ftg/ D jM j.

Based on Lemma 26.9, we would like to conclude that a maximum matching

in a bipartite graph G corresponds to a maximum ﬂow in its corresponding ﬂow

network G 0

, and we can therefore compute a maximum matching in G by running

a maximum-ﬂow algorithm on G 0

. The only hitch in this reasoning is that the

maximum-ﬂow algorithm might return a ﬂow in G 0

for which some f .u; / is

not an integer, even though the ﬂow value jf j must be an integer. The following

theorem shows that if we use the Ford-Fulkerson method, this difﬁculty cannot

arise.

Theorem 26.10 (Integrality theorem)

If the capacity function c takes on only integral values, then the maximum ﬂow f

produced by the Ford-Fulkerson method has the property that jf j is an integer.

Moreover, for all vertices u and , the value of f .u; / is an integer.

Proof The proof is by induction on the number of iterations. We leave it as

Exercise 26.3-2.

We can now prove the following corollary to Lemma 26.9.

26.3 Maximum bipartite matching 735

Corollary 26.11

The cardinality of a maximum matching M in a bipartite graph G equals the value

of a maximum ﬂow f in its corresponding ﬂow network G 0

.

Proof We use the nomenclature from Lemma 26.9. Suppose that M is a max-

imum matching in G and that the corresponding ﬂow f in G 0

is not maximum.

Then there is a maximum ﬂow f 0

in G 0

such that jf 0 j > jf j. Since the ca-

pacities in G 0

are integer-valued, by Theorem 26.10, we can assume that f 0

is

integer-valued. Thus, f 0

corresponds to a matching M 0

in G with cardinality

jM 0 j D jf 0 j > jf j D jM j, contradicting our assumption that M is a maximum

matching. In a similar manner, we can show that if f is a maximum ﬂow in G 0

, its

corresponding matching is a maximum matching on G.

Thus, given a bipartite undirected graph G, we can ﬁnd a maximum matching by

creating the ﬂow network G 0

, running the Ford-Fulkerson method, and directly ob-

taining a maximum matching M from the integer-valued maximum ﬂow f found.

Since any matching in a bipartite graph has cardinality at most min.L; R/ D O.V /,

the value of the maximum ﬂow in G 0

is O.V /. We can therefore ﬁnd a maximum

matching in a bipartite graph in time O.VE 0 / D O.VE/, since jE 0 j D ‚.E/.

Exercises

26.3-1

Run the Ford-Fulkerson algorithm on the ﬂow network in Figure 26.8(c) and show

the residual network after each ﬂow augmentation. Number the vertices in L top

to bottom from 1 to 5 and in R top to bottom from 6 to 9. For each iteration, pick

the augmenting path that is lexicographically smallest.

26.3-2

Prove Theorem 26.10.

26.3-3

Let G D .V; E/ be a bipartite graph with vertex partition V D L [ R, and let G 0

be its corresponding ﬂow network. Give a good upper bound on the length of any

augmenting path found in G 0

during the execution of FORD-FULKERSON.

26.3-4 ?

A perfect matching is a matching in which every vertex is matched. Let G D

.V; E/ be an undirected bipartite graph with vertex partition V D L [ R, where

jLj D jRj. For any X V , deﬁne the neighborhood of X as

N.X/ D fy 2 V W .x; y/ 2 E for some x 2 Xg ;

736 Chapter 26 Maximum Flow

that is, the set of vertices adjacent to some member of X. Prove Hall’s theorem:

there exists a perfect matching in G if and only if jAj jN.A/j for every subset

A L.

26.3-5 ?

We say that a bipartite graph G D .V; E/, where V D L[R, is d-regular if every

vertex 2 V has degree exactly d. Every d-regular bipartite graph has jLj D jRj.

Prove that every d-regular bipartite graph has a matching of cardinality jLj by

arguing that a minimum cut of the corresponding ﬂow network has capacity jLj.

? 26.4 Push-relabel algorithms

In this section, we present the “push-relabel” approach to computing maximum

ﬂows. To date, many of the asymptotically fastest maximum-ﬂow algorithms are

push-relabel algorithms, and the fastest actual implementations of maximum-ﬂow

algorithms are based on the push-relabel method. Push-relabel methods also efﬁ-

ciently solve other ﬂow problems, such as the minimum-cost ﬂow problem. This

section introduces Goldberg’s “generic” maximum-ﬂow algorithm, which has a

simple implementation that runs in O.V 2 E/ time, thereby improving upon the

O.VE 2 / bound of the Edmonds-Karp algorithm. Section 26.5 reﬁnes the generic

algorithm to obtain another push-relabel algorithm that runs in O.V 3 / time.

Push-relabel algorithms work in a more localized manner than the Ford-

Fulkerson method. Rather than examine the entire residual network to ﬁnd an aug-

menting path, push-relabel algorithms work on one vertex at a time, looking only

at the vertex’s neighbors in the residual network. Furthermore, unlike the Ford-

Fulkerson method, push-relabel algorithms do not maintain the ﬂow-conservation

property throughout their execution. They do, however, maintain a preﬂow, which

is a function f W V V ! R that satisﬁes the capacity constraint and the following

relaxation of ﬂow conservation:

X

2V

f .; u/

X

2V

f .u; / 0

for all vertices u 2 V fsg. That is, the ﬂow into a vertex may exceed the ﬂow

out. We call the quantity

e.u/ D

X

2V

f .; u/

X

2V

f .u; / (26.14)

the excess ﬂow into vertex u. The excess at a vertex is the amount by which the

ﬂow in exceeds the ﬂow out. We say that a vertex u 2 V fs; tg is overﬂowing if

e.u/ > 0.

26.4 Push-relabel algorithms 737

We shall begin this section by describing the intuition behind the push-relabel

method. We shall then investigate the two operations employed by the method:

“pushing” preﬂow and “relabeling” a vertex. Finally, we shall present a generic

push-relabel algorithm and analyze its correctness and running time.

Intuition

You can understand the intuition behind the push-relabel method in terms of ﬂuid

ﬂows: we consider a ﬂow network G D .V; E/ to be a system of interconnected

pipes of given capacities. Applying this analogy to the Ford-Fulkerson method,

we might say that each augmenting path in the network gives rise to an additional

stream of ﬂuid, with no branch points, ﬂowing from the source to the sink. The

Ford-Fulkerson method iteratively adds more streams of ﬂow until no more can be

added.

The generic push-relabel algorithm has a rather different intuition. As before,

directed edges correspond to pipes. Vertices, which are pipe junctions, have two

interesting properties. First, to accommodate excess ﬂow, each vertex has an out-

ﬂow pipe leading to an arbitrarily large reservoir that can accumulate ﬂuid. Second,

each vertex, its reservoir, and all its pipe connections sit on a platform whose height

increases as the algorithm progresses.

Vertex heights determine how ﬂow is pushed: we push ﬂow only downhill, that

is, from a higher vertex to a lower vertex. The ﬂow from a lower vertex to a higher

vertex may be positive, but operations that push ﬂow push it only downhill. We

ﬁx the height of the source at jV j and the height of the sink at 0. All other vertex

heights start at 0 and increase with time. The algorithm ﬁrst sends as much ﬂow as

possible downhill from the source toward the sink. The amount it sends is exactly

enough to ﬁll each outgoing pipe from the source to capacity; that is, it sends the

capacity of the cut .s; V fsg/. When ﬂow ﬁrst enters an intermediate vertex, it

collects in the vertex’s reservoir. From there, we eventually push it downhill.

We may eventually ﬁnd that the only pipes that leave a vertex u and are not

already saturated with ﬂow connect to vertices that are on the same level as u or

are uphill from u. In this case, to rid an overﬂowing vertex u of its excess ﬂow, we

must increase its height—an operation called “relabeling” vertex u. We increase

its height to one unit more than the height of the lowest of its neighbors to which

it has an unsaturated pipe. After a vertex is relabeled, therefore, it has at least one

outgoing pipe through which we can push more ﬂow.

Eventually, all the ﬂow that can possibly get through to the sink has arrived there.

No more can arrive, because the pipes obey the capacity constraints; the amount of

ﬂow across any cut is still limited by the capacity of the cut. To make the preﬂow

a “legal” ﬂow, the algorithm then sends the excess collected in the reservoirs of

overﬂowing vertices back to the source by continuing to relabel vertices to above

738 Chapter 26 Maximum Flow

the ﬁxed height jV j of the source. As we shall see, once we have emptied all the

reservoirs, the preﬂow is not only a “legal” ﬂow, it is also a maximum ﬂow.

The basic operations

From the preceding discussion, we see that a push-relabel algorithm performs two

basic operations: pushing ﬂow excess from a vertex to one of its neighbors and

relabeling a vertex. The situations in which these operations apply depend on the

heights of vertices, which we now deﬁne precisely.

Let G D .V; E/ be a ﬂow network with source s and sink t, and let f be a

preﬂow in G. A function h W V ! N is a height function 3 if h.s/ D jV j,

h.t/ D 0, and

h.u/ h./ C 1

for every residual edge .u; / 2 E f . We immediately obtain the following lemma.

Lemma 26.12

Let G D .V; E/ be a ﬂow network, let f be a preﬂow in G, and let h be a height

function on V . For any two vertices u; 2 V , if h.u/ > h./ C 1, then .u; / is

not an edge in the residual network.

The push operation

The basic operation PUSH.u; / applies if u is an overﬂowing vertex, c f .u; / > 0,

and h.u/ D h./C1. The pseudocode below updates the preﬂow f and the excess

ﬂows for u and . It assumes that we can compute residual capacity c f .u; / in

constant time given c and f . We maintain the excess ﬂow stored at a vertex u as

the attribute u:e and the height of u as the attribute u:h. The expression f .u; /

is a temporary variable that stores the amount of ﬂow that we can push from u to .

3

In the literature, a height function is typically called a “distance function,” and the height of a vertex

is called a “distance label.” We use the term “height” because it is more suggestive of the intuition

behind the algorithm. We retain the use of the term “relabel” to refer to the operation that increases

the height of a vertex. The height of a vertex is related to its distance from the sink t, as would be

found in a breadth-ﬁrst search of the transpose G

T

.

26.4 Push-relabel algorithms 739

PUSH.u; /

1 // Applies when: u is overﬂowing, c f .u; / > 0, and u:h D :h C 1.

2 // Action: Push f .u; / D min.u:e; c f .u; // units of ﬂow from u to .

3 f .u; / D min.u:e; c f .u; //

4 if .u; / 2 E

5 .u; /:f D .u; /:f C f .u; /

6 else .; u/:f D .; u/:f f .u; /

7 u:e D u:e f .u; /

8 :e D :e C f .u; /

The code for PUSH operates as follows. Because vertex u has a positive excess u:e

and the residual capacity of .u; / is positive, we can increase the ﬂow from u to

by f .u; / D min.u:e; c f .u; // without causing u:e to become negative or the

capacity c.u; / to be exceeded. Line 3 computes the value f .u; /, and lines 4–6

update f . Line 5 increases the ﬂow on edge .u; /, because we are pushing ﬂow

over a residual edge that is also an original edge. Line 6 decreases the ﬂow on

edge .; u/, because the residual edge is actually the reverse of an edge in the

original network. Finally, lines 7–8 update the excess ﬂows into vertices u and .

Thus, if f is a preﬂow before PUSH is called, it remains a preﬂow afterward.

Observe that nothing in the code for PUSH depends on the heights of u and ,

yet we prohibit it from being invoked unless u:h D :h C 1. Thus, we push excess

ﬂow downhill only by a height differential of 1. By Lemma 26.12, no residual

edges exist between two vertices whose heights differ by more than 1, and thus,

as long as the attribute h is indeed a height function, we would gain nothing by

allowing ﬂow to be pushed downhill by a height differential of more than 1.

We call the operation PUSH.u; / a push from u to . If a push operation ap-

plies to some edge .u; / leaving a vertex u, we also say that the push operation

applies to u. It is a saturating push if edge .u; / in the residual network becomes

saturated (c f .u; / D 0 afterward); otherwise, it is a nonsaturating push. If an

edge becomes saturated, it disappears from the residual network. A simple lemma

characterizes one result of a nonsaturating push.

Lemma 26.13

After a nonsaturating push from u to , the vertex u is no longer overﬂowing.

Proof Since the push was nonsaturating, the amount of ﬂow f .u; / actually

pushed must equal u:e prior to the push. Since u:e is reduced by this amount, it

becomes 0 after the push.

740 Chapter 26 Maximum Flow

The relabel operation

The basic operation RELABEL.u/ applies if u is overﬂowing and if u:h :h for

all edges .u; / 2 E f . In other words, we can relabel an overﬂowing vertex u if

for every vertex for which there is residual capacity from u to , ﬂow cannot be

pushed from u to because is not downhill from u. (Recall that by deﬁnition,

neither the source s nor the sink t can be overﬂowing, and so s and t are ineligible

for relabeling.)

RELABEL.u/

1 // Applies when: u is overﬂowing and for all 2 V such that .u; / 2 E f ,

we have u:h :h.

2 // Action: Increase the height of u.

3 u:h D 1 C min f:h W .u; / 2 E f g

When we call the operation RELABEL.u/, we say that vertex u is relabeled. Note

that when u is relabeled, E f must contain at least one edge that leaves u, so that

the minimization in the code is over a nonempty set. This property follows from

the assumption that u is overﬂowing, which in turn tells us that

u:e D

X

2V

f .; u/

X

2V

f .u; / > 0 :

Since all ﬂows are nonnegative, we must therefore have at least one vertex such

that .; u/:f > 0. But then, c f .u; / > 0, which implies that .u; / 2 E f . The

operation RELABEL.u/ thus gives u the greatest height allowed by the constraints

on height functions.

The generic algorithm

The generic push-relabel algorithm uses the following subroutine to create an ini-

tial preﬂow in the ﬂow network.

INITIALIZE-PREFLOW.G; s/

1 for each vertex 2 G:V

2 :h D 0

3 :e D 0

4 for each edge .u; / 2 G:E

5 .u; /:f D 0

6 s:h D jG:Vj

7 for each vertex 2 s:Adj

8 .s; /:f D c.s; /

9 :e D c.s; /

10 s:e D s:e c.s; /

26.4 Push-relabel algorithms 741

INITIALIZE-PREFLOW creates an initial preﬂow f deﬁned by

.u; /:f D

(

c.u; / if u D s ;

0 otherwise :

(26.15)

That is, we ﬁll to capacity each edge leaving the source s, and all other edges carry

no ﬂow. For each vertex adjacent to the source, we initially have :e D c.s; /,

and we initialize s:e to the negative of the sum of these capacities. The generic

algorithm also begins with an initial height function h, given by

u:h D

(

jV j if u D s ;

0 otherwise :

(26.16)

Equation (26.16) deﬁnes a height function because the only edges .u; / for which

u:h > :h C 1 are those for which u D s, and those edges are saturated, which

means that they are not in the residual network.

Initialization, followed by a sequence of push and relabel operations, executed

in no particular order, yields the GENERIC-PUSH-RELABEL algorithm:

GENERIC-PUSH-RELABEL.G/

1 INITIALIZE-PREFLOW.G; s/

2 while there exists an applicable push or relabel operation

3 select an applicable push or relabel operation and perform it

The following lemma tells us that as long as an overﬂowing vertex exists, at least

one of the two basic operations applies.

Lemma 26.14 (An overﬂowing vertex can be either pushed or relabeled)

Let G D .V; E/ be a ﬂow network with source s and sink t, let f be a preﬂow,

and let h be any height function for f . If u is any overﬂowing vertex, then either a

push or relabel operation applies to it.

Proof For any residual edge .u; /, we have h.u/ h./ C 1 because h is a

height function. If a push operation does not apply to an overﬂowing vertex u,

then for all residual edges .u; /, we must have h.u/ < h./ C 1, which implies

h.u/ h./. Thus, a relabel operation applies to u.

Correctness of the push-relabel method

To show that the generic push-relabel algorithm solves the maximum-ﬂow prob-

lem, we shall ﬁrst prove that if it terminates, the preﬂow f is a maximum ﬂow.

We shall later prove that it terminates. We start with some observations about the

height function h.

742 Chapter 26 Maximum Flow

Lemma 26.15 (Vertex heights never decrease)

During the execution of the GENERIC-PUSH-RELABEL procedure on a ﬂow net-

work G D .V; E/, for each vertex u 2 V , the height u:h never decreases. More-

over, whenever a relabel operation is applied to a vertex u, its height u:h increases

by at least 1.

Proof Because vertex heights change only during relabel operations, it sufﬁces

to prove the second statement of the lemma. If vertex u is about to be rela-

beled, then for all vertices such that .u; / 2 E f , we have u:h :h. Thus,

u:h < 1 C min f:h W .u; / 2 E f g, and so the operation must increase u:h.

Lemma 26.16

Let G D .V; E/ be a ﬂow network with source s and sink t. Then the execution of

GENERIC-PUSH-RELABEL on G maintains the attribute h as a height function.

Proof The proof is by induction on the number of basic operations performed.

Initially, h is a height function, as we have already observed.

We claim that if h is a height function, then an operation RELABEL.u/ leaves h

a height function. If we look at a residual edge .u; / 2 E f that leaves u, then

the operation RELABEL.u/ ensures that u:h :h C 1 afterward. Now consider

a residual edge .w; u/ that enters u. By Lemma 26.15, w:h u:h C 1 before the

operation RELABEL.u/ implies w:h < u:h C 1 afterward. Thus, the operation

RELABEL.u/ leaves h a height function.

Now, consider an operation PUSH.u; /. This operation may add the edge .; u/

to E f , and it may remove .u; / from E f . In the former case, we have

:h D u:h 1 < u:h C 1, and so h remains a height function. In the latter case,

removing .u; / from the residual network removes the corresponding constraint,

and h again remains a height function.

The following lemma gives an important property of height functions.

Lemma 26.17

Let G D .V; E/ be a ﬂow network with source s and sink t, let f be a preﬂow

in G, and let h be a height function on V . Then there is no path from the source s

to the sink t in the residual network G f .

Proof Assume for the sake of contradiction that G f contains a path p from s to t,

where p D h 0 ; 1 ; : : : ; k i, 0 D s, and k D t. Without loss of generality, p

is a simple path, and so k < jV j. For i D 0; 1; : : : ; k 1, edge . i ; iC1 / 2 E f .

Because h is a height function, h. i / h. iC1 / C 1 for i D 0; 1; : : : ; k 1. Com-

bining these inequalities over path p yields h.s/ h.t/Ck. But because h.t/ D 0,

26.4 Push-relabel algorithms 743

we have h.s/ k < jV j, which contradicts the requirement that h.s/ D jV j in a

height function.

We are now ready to show that if the generic push-relabel algorithm terminates,

the preﬂow it computes is a maximum ﬂow.

Theorem 26.18 (Correctness of the generic push-relabel algorithm)

If the algorithm GENERIC-PUSH-RELABEL terminates when run on a ﬂow net-

work G D .V; E/ with source s and sink t, then the preﬂow f it computes is a

maximum ﬂow for G.

Proof We use the following loop invariant:

Each time the while loop test in line 2 in GENERIC-PUSH-RELABEL is

executed, f is a preﬂow.

Initialization: INITIALIZE-PREFLOW makes f a preﬂow.

Maintenance: The only operations within the while loop of lines 2–3 are push and

relabel. Relabel operations affect only height attributes and not the ﬂow values;

hence they do not affect whether f is a preﬂow. As argued on page 739, if f is

a preﬂow prior to a push operation, it remains a preﬂow afterward.

Termination: At termination, each vertex in V fs; tg must have an excess of 0,

because by Lemma 26.14 and the invariant that f is always a preﬂow, there are

no overﬂowing vertices. Therefore, f is a ﬂow. Lemma 26.16 shows that h is

a height function at termination, and thus Lemma 26.17 tells us that there is no

path from s to t in the residual network G f . By the max-ﬂow min-cut theorem

(Theorem 26.6), therefore, f is a maximum ﬂow.

Analysis of the push-relabel method

To show that the generic push-relabel algorithm indeed terminates, we shall bound

the number of operations it performs. We bound separately each of the three types

of operations: relabels, saturating pushes, and nonsaturating pushes. With knowl-

edge of these bounds, it is a straightforward problem to construct an algorithm that

runs in O.V 2 E/ time. Before beginning the analysis, however, we prove an im-

portant lemma. Recall that we allow edges into the source in the residual network.

Lemma 26.19

Let G D .V; E/ be a ﬂow network with source s and sink t, and let f be a preﬂow

in G. Then, for any overﬂowing vertex x, there is a simple path from x to s in the

residual network G f .

744 Chapter 26 Maximum Flow

Proof For an overﬂowing vertex x, let U D f W there exists a simple path from x

to in G f g, and suppose for the sake of contradiction that s 62 U . Let U D V U .

We take the deﬁnition of excess from equation (26.14), sum over all vertices

in U , and note that V D U [ U , to obtain

X

u2U

e.u/

D

X

u2U

X

2V

f .; u/

X

2V

f .u; /

!

D

X

u2U

X

2U

f .; u/ C

X

2U

f .; u/

!

X

2U

f .u; / C

X

2U

f .u; /

!!

D

X

u2U

X

2U

f .; u/ C

X

u2U

X

2U

f .; u/

X

u2U

X

2U

f .u; /

X

u2U

X

2U

f .u; /

D

X

u2U

X

2U

f .; u/

X

u2U

X

2U

f .u; / :

We know that the quantity

P

u2U

e.u/ must be positive because e.x/ > 0, x 2 U ,

all vertices other than s have nonnegative excess, and, by assumption, s 62 U . Thus,

we have

X

u2U

X

2U

f .; u/

X

u2U

X

2U

f .u; / > 0 : (26.17)

All edge ﬂows are nonnegative, and so for equation (26.17) to hold, we must have P

u2U

P

2U

f .; u/ > 0. Hence, there must exist at least one pair of vertices

u 0 2 U and 0 2 U with f . 0 ; u 0 / > 0. But, if f . 0 ; u 0 / > 0, there must be a

residual edge .u 0 ; 0 /, which means that there is a simple path from x to 0

(the

path x u 0 ! 0

), thus contradicting the deﬁnition of U .

The next lemma bounds the heights of vertices, and its corollary bounds the

number of relabel operations that are performed in total.

Lemma 26.20

Let G D .V; E/ be a ﬂow network with source s and sink t. At any time during

the execution of GENERIC-PUSH-RELABEL on G, we have u:h 2 jV j1 for all

vertices u 2 V .

Proof The heights of the source s and the sink t never change because these

vertices are by deﬁnition not overﬂowing. Thus, we always have s:h D jV j and

t:h D 0, both of which are no greater than 2 jV j 1.

Now consider any vertex u 2 V fs; tg. Initially, u:h D 0 2 jV j1. We shall

show that after each relabeling operation, we still have u:h 2 jV j 1. When u is

26.4 Push-relabel algorithms 745

relabeled, it is overﬂowing, and Lemma 26.19 tells us that there is a simple path p

from u to s in G f . Let p D h 0 ; 1 ;: : : ; k i, where 0 D u, k D s, and k jV j1

because p is simple. For i D 0; 1; : : : ; k 1, we have . i ; iC1 / 2 E f , and

therefore, by Lemma 26.16, i :h iC1 :h C 1. Expanding these inequalities over

path p yields u:h D 0 :h k :h C k s:h C .jV j 1/ D 2 jV j 1.

Corollary 26.21 (Bound on relabel operations)

Let G D .V; E/ be a ﬂow network with source s and sink t. Then, during the

execution of GENERIC-PUSH-RELABEL on G, the number of relabel operations is

at most 2 jV j 1 per vertex and at most .2 jV j 1/.jV j 2/ < 2 jV j

2

overall.

Proof Only the jV j2 vertices in V fs; tg may be relabeled. Let u 2 V fs; tg.

The operation RELABEL.u/ increases u:h. The value of u:h is initially 0 and by

Lemma 26.20, it grows to at most 2 jV j 1. Thus, each vertex u 2 V fs; tg

is relabeled at most 2 jV j 1 times, and the total number of relabel operations

performed is at most .2 jV j 1/.jV j 2/ < 2 jV j

2

.

Lemma 26.20 also helps us to bound the number of saturating pushes.

Lemma 26.22 (Bound on saturating pushes)

During the execution of GENERIC-PUSH-RELABEL on any ﬂow network G D

.V; E/, the number of saturating pushes is less than 2 jV j jEj.

Proof For any pair of vertices u; 2 V , we will count the saturating pushes

from u to and from to u together, calling them the saturating pushes between u

and . If there are any such pushes, at least one of .u; / and .; u/ is actually

an edge in E. Now, suppose that a saturating push from u to has occurred.

At that time, :h D u:h 1. In order for another push from u to to occur

later, the algorithm must ﬁrst push ﬂow from to u, which cannot happen until

:h D u:h C 1. Since u:h never decreases, in order for :h D u:h C 1, the

value of :h must increase by at least 2. Likewise, u:h must increase by at least 2

between saturating pushes from to u. Heights start at 0 and, by Lemma 26.20,

never exceed 2 jV j1, which implies that the number of times any vertex can have

its height increase by 2 is less than jV j. Since at least one of u:h and :h must

increase by 2 between any two saturating pushes between u and , there are fewer

than 2 jV j saturating pushes between u and . Multiplying by the number of edges

gives a bound of less than 2 jV j jEj on the total number of saturating pushes.

The following lemma bounds the number of nonsaturating pushes in the generic

push-relabel algorithm.

746 Chapter 26 Maximum Flow

Lemma 26.23 (Bound on nonsaturating pushes)

During the execution of GENERIC-PUSH-RELABEL on any ﬂow network G D

.V; E/, the number of nonsaturating pushes is less than 4 jV j

2

.jV j C jEj/.

Proof Deﬁne a potential function ˆ D

P

We./>0

:h. Initially, ˆ D 0, and the

value of ˆ may change after each relabeling, saturating push, and nonsaturating

push. We will bound the amount that saturating pushes and relabelings can con-

tribute to the increase of ˆ. Then we will show that each nonsaturating push must

decrease ˆ by at least 1, and will use these bounds to derive an upper bound on the

number of nonsaturating pushes.

Let us examine the two ways in which ˆ might increase. First, relabeling a

vertex u increases ˆ by less than 2 jV j, since the set over which the sum is taken is

the same and the relabeling cannot increase u’s height by more than its maximum

possible height, which, by Lemma 26.20, is at most 2 jV j 1. Second, a saturating

push from a vertex u to a vertex increases ˆ by less than 2 jV j, since no heights

change and only vertex , whose height is at most 2 jV j 1, can possibly become

overﬂowing.

Now we show that a nonsaturating push from u to decreases ˆ by at least 1.

Why? Before the nonsaturating push, u was overﬂowing, and may or may not

have been overﬂowing. By Lemma 26.13, u is no longer overﬂowing after the

push. In addition, unless is the source, it may or may not be overﬂowing after

the push. Therefore, the potential function ˆ has decreased by exactly u:h, and it

has increased by either 0 or :h. Since u:h :h D 1, the net effect is that the

potential function has decreased by at least 1.

Thus, during the course of the algorithm, the total amount of increase in ˆ is

due to relabelings and saturated pushes, and Corollary 26.21 and Lemma 26.22

constrain the increase to be less than .2 jV j/.2 jV j

2

/ C .2 jV j/.2 jV j jEj/ D

4 jV j

2

.jV j C jEj/. Since ˆ 0, the total amount of decrease, and therefore the

total number of nonsaturating pushes, is less than 4 jV j

2

.jV j C jEj/.

Having bounded the number of relabelings, saturating pushes, and nonsatu-

rating push, we have set the stage for the following analysis of the GENERIC-

PUSH-RELABEL procedure, and hence of any algorithm based on the push-relabel

method.

Theorem 26.24

During the execution of GENERIC-PUSH-RELABEL on any ﬂow network G D

.V; E/, the number of basic operations is O.V 2 E/.

Proof Immediate from Corollary 26.21 and Lemmas 26.22 and 26.23.

26.4 Push-relabel algorithms 747

Thus, the algorithm terminates after O.V 2 E/ operations. All that remains is

to give an efﬁcient method for implementing each operation and for choosing an

appropriate operation to execute.

Corollary 26.25

There is an implementation of the generic push-relabel algorithm that runs in

O.V 2 E/ time on any ﬂow network G D .V; E/.

Proof Exercise 26.4-2 asks you to show how to implement the generic algorithm

with an overhead of O.V / per relabel operation and O.1/ per push. It also asks

you to design a data structure that allows you to pick an applicable operation in

O.1/ time. The corollary then follows.

Exercises

26.4-1

Prove that, after the procedure INITIALIZE-PREFLOW.G; s/ terminates, we have

s:e jf j, where f

is a maximum ﬂow for G.

26.4-2

Show how to implement the generic push-relabel algorithm using O.V / time per

relabel operation, O.1/ time per push, and O.1/ time to select an applicable oper-

ation, for a total time of O.V 2 E/.

26.4-3

Prove that the generic push-relabel algorithm spends a total of only O.VE/ time

in performing all the O.V 2 / relabel operations.

26.4-4

Suppose that we have found a maximum ﬂow in a ﬂow network G D .V; E/ using

a push-relabel algorithm. Give a fast algorithm to ﬁnd a minimum cut in G.

26.4-5

Give an efﬁcient push-relabel algorithm to ﬁnd a maximum matching in a bipartite

graph. Analyze your algorithm.

26.4-6

Suppose that all edge capacities in a ﬂow network G D .V; E/ are in the set

f1; 2; : : : ; kg. Analyze the running time of the generic push-relabel algorithm in

terms of jV j, jEj, and k. (Hint: How many times can each edge support a nonsat-

urating push before it becomes saturated?)

748 Chapter 26 Maximum Flow

26.4-7

Show that we could change line 6 of INITIALIZE-PREFLOW to

6 s:h D jG:Vj 2

without affecting the correctness or asymptotic performance of the generic push-

relabel algorithm.

26.4-8

Let ı f .u; / be the distance (number of edges) from u to in the residual net-

work G f . Show that the GENERIC-PUSH-RELABEL procedure maintains the

properties that u:h < jV j implies u:h ı f .u; t/ and that u:h jV j implies

u:h jV j ı f .u; s/.

26.4-9 ?

As in the previous exercise, let ı f .u; / be the distance from u to in the residual

network G f . Show how to modify the generic push-relabel algorithm to maintain

the property that u:h < jV j implies u:h D ı f .u; t/ and that u:h jV j implies

u:h jV j D ı f .u; s/. The total time that your implementation dedicates to main-

taining this property should be O.VE/.

26.4-10

Show that the number of nonsaturating pushes executed by the GENERIC-PUSH-

RELABEL procedure on a ﬂow network G D .V; E/ is at most 4 jV j

2

jEj for

jV j 4.

? 26.5 The relabel-to-front algorithm

The push-relabel method allows us to apply the basic operations in any order at

all. By choosing the order carefully and managing the network data structure efﬁ-

ciently, however, we can solve the maximum-ﬂow problem faster than the O.V 2 E/

bound given by Corollary 26.25. We shall now examine the relabel-to-front algo-

rithm, a push-relabel algorithm whose running time is O.V 3 /, which is asymptot-

ically at least as good as O.V 2 E/, and even better for dense networks.

The relabel-to-front algorithm maintains a list of the vertices in the network.

Beginning at the front, the algorithm scans the list, repeatedly selecting an over-

ﬂowing vertex u and then “discharging” it, that is, performing push and relabel

operations until u no longer has a positive excess. Whenever we relabel a ver-

tex, we move it to the front of the list (hence the name “relabel-to-front”) and the

algorithm begins its scan anew.

26.5 The relabel-to-front algorithm 749

The correctness and analysis of the relabel-to-front algorithm depend on the

notion of “admissible” edges: those edges in the residual network through which

ﬂow can be pushed. After proving some properties about the network of admissible

edges, we shall investigate the discharge operation and then present and analyze the

relabel-to-front algorithm itself.

Admissible edges and networks

If G D .V; E/ is a ﬂow network with source s and sink t, f is a preﬂow in G, and h

is a height function, then we say that .u; / is an admissible edge if c f .u; / > 0

and h.u/ D h./ C 1. Otherwise, .u; / is inadmissible. The admissible network

is G f;h D .V; E f;h /, where E f;h is the set of admissible edges.

The admissible network consists of those edges through which we can push ﬂow.

The following lemma shows that this network is a directed acyclic graph (dag).

Lemma 26.26 (The admissible network is acyclic)

If G D .V; E/ is a ﬂow network, f is a preﬂow in G, and h is a height function

on G, then the admissible network G f;h D .V; E f;h / is acyclic.

Proof The proof is by contradiction. Suppose that G f;h contains a cycle p D

h 0 ; 1 ; : : : ; k i, where 0 D k and k > 0. Since each edge in p is admissible, we

have h. i1 / D h. i / C 1 for i D 1; 2; : : : ; k. Summing around the cycle gives

k X

iD1

h. i1 / D

k X

iD1

.h. i / C 1/

D

k X

iD1

h. i / C k :

Because each vertex in cycle p appears once in each of the summations, we derive

the contradiction that 0 D k.

The next two lemmas show how push and relabel operations change the admis-

sible network.

Lemma 26.27

Let G D .V; E/ be a ﬂow network, let f be a preﬂow in G, and suppose that the

attribute h is a height function. If a vertex u is overﬂowing and .u; / is an ad-

missible edge, then PUSH.u; / applies. The operation does not create any new

admissible edges, but it may cause .u; / to become inadmissible.

750 Chapter 26 Maximum Flow

Proof By the deﬁnition of an admissible edge, we can push ﬂow from u to .

Since u is overﬂowing, the operation PUSH.u; / applies. The only new residual

edge that pushing ﬂow from u to can create is .; u/. Since :h D u:h 1,

edge .; u/ cannot become admissible. If the operation is a saturating push, then

c f .u; / D 0 afterward and .u; / becomes inadmissible.

Lemma 26.28

Let G D .V; E/ be a ﬂow network, let f be a preﬂow in G, and suppose that

the attribute h is a height function. If a vertex u is overﬂowing and there are no

admissible edges leaving u, then RELABEL.u/ applies. After the relabel operation,

there is at least one admissible edge leaving u, but there are no admissible edges

entering u.

Proof If u is overﬂowing, then by Lemma 26.14, either a push or a relabel op-

eration applies to it. If there are no admissible edges leaving u, then no ﬂow

can be pushed from u and so RELABEL.u/ applies. After the relabel operation,

u:h D 1 C min f:h W .u; / 2 E f g. Thus, if is a vertex that realizes the mini-

mum in this set, the edge .u; / becomes admissible. Hence, after the relabel, there

is at least one admissible edge leaving u.

To show that no admissible edges enter u after a relabel operation, suppose that

there is a vertex such that .; u/ is admissible. Then, :h D u:h C 1 after the

relabel, and so :h > u:h C 1 just before the relabel. But by Lemma 26.12, no

residual edges exist between vertices whose heights differ by more than 1. More-

over, relabeling a vertex does not change the residual network. Thus, .; u/ is not

in the residual network, and hence it cannot be in the admissible network.

Neighbor lists

Edges in the relabel-to-front algorithm are organized into “neighbor lists.” Given

a ﬂow network G D .V; E/, the neighbor list u:N for a vertex u 2 V is a singly

linked list of the neighbors of u in G. Thus, vertex appears in the list u:N if

.u; / 2 E or .; u/ 2 E. The neighbor list u:N contains exactly those vertices

for which there may be a residual edge .u; /. The attribute u:N:head points to

the ﬁrst vertex in u:N, and :next-neighbor points to the vertex following in a

neighbor list; this pointer is NIL if is the last vertex in the neighbor list.

The relabel-to-front algorithm cycles through each neighbor list in an arbitrary

order that is ﬁxed throughout the execution of the algorithm. For each vertex u,

the attribute u:current points to the vertex currently under consideration in u:N.

Initially, u:current is set to u:N:head.

26.5 The relabel-to-front algorithm 751

Discharging an overﬂowing vertex

An overﬂowing vertex u is discharged by pushing all of its excess ﬂow through

admissible edges to neighboring vertices, relabeling u as necessary to cause edges

leaving u to become admissible. The pseudocode goes as follows.

DISCHARGE.u/

1 while u:e > 0

2 D u:current

3 if == NIL

4 RELABEL.u/

5 u:current D u:N:head

6 elseif c f .u; / > 0 and u:h == :h C 1

7 PUSH.u; /

8 else u:current D :next-neighbor

Figure 26.9 steps through several iterations of the while loop of lines 1–8, which

executes as long as vertex u has positive excess. Each iteration performs exactly

one of three actions, depending on the current vertex in the neighbor list u:N.

1. If is NIL, then we have run off the end of u:N. Line 4 relabels vertex u,

and then line 5 resets the current neighbor of u to be the ﬁrst one in u:N.

(Lemma 26.29 below states that the relabel operation applies in this situation.)

2. If is non-NIL and .u; / is an admissible edge (determined by the test in

line 6), then line 7 pushes some (or possibly all) of u’s excess to vertex .

3. If is non-NIL but .u; / is inadmissible, then line 8 advances u:current one

position further in the neighbor list u:N.

Observe that if DISCHARGE is called on an overﬂowing vertex u, then the last

action performed by DISCHARGE must be a push from u. Why? The procedure

terminates only when u:e becomes zero, and neither the relabel operation nor ad-

vancing the pointer u:current affects the value of u:e.

We must be sure that when PUSH or RELABEL is called by DISCHARGE, the

operation applies. The next lemma proves this fact.

Lemma 26.29

If DISCHARGE calls PUSH.u; / in line 7, then a push operation applies to .u; /.

If DISCHARGE calls RELABEL.u/ in line 4, then a relabel operation applies to u.

Proof The tests in lines 1 and 6 ensure that a push operation occurs only if the

operation applies, which proves the ﬁrst statement in the lemma.

752 Chapter 26 Maximum Flow

s

–26

5

4

3

2

1

0

6

x

0

y

19

z

05/5

8

14/14

s

x

z

s

–26

5

4

3

2

1

0

6

x

0

y

19

z

0

8

14/14

s

x

z

5/5

s

–26

5

4

3

2

1

0

6

x

0

y

11

z

8

8/8

14/14

5/5

s

x

z

s

x

z

1 2 3

s

x

z

4

5

s

x

z

6

s

x

z

7

s

x

z

8

s

x

z

9

(a)

(b)

(c)

Figure 26.9 Discharging a vertex y. It takes 15 iterations of the while loop of DISCHARGE to push

all the excess ﬂow from y. Only the neighbors of y and edges of the ﬂow network that enter or leave y

are shown. In each part of the ﬁgure, the number inside each vertex is its excess at the beginning of

the ﬁrst iteration shown in the part, and each vertex is shown at its height throughout the part. The

neighbor list y:N at the beginning of each iteration appears on the right, with the iteration number

on top. The shaded neighbor is y:current. (a) Initially, there are 19 units of excess to push from y,

and y:current D s. Iterations 1, 2, and 3 just advance y:current, since there are no admissible edges

leaving y. In iteration 4, y:current D NIL (shown by the shading being below the neighbor list),

and so y is relabeled and y:current is reset to the head of the neighbor list. (b) After relabeling,

vertex y has height 1. In iterations 5 and 6, edges .y; s/ and .y; x/ are found to be inadmissible, but

iteration 7 pushes 8 units of excess ﬂow from y to ´. Because of the push, y:current does not advance

in this iteration. (c) Because the push in iteration 7 saturated edge .y; ´/, it is found inadmissible in

iteration 8. In iteration 9, y:current D NIL, and so vertex y is again relabeled and y:current is reset.

26.5 The relabel-to-front algorithm 753

s

–26

5

4

3

2

1

0

6

x

5

y

6

z

8

5

8/8

14/14

s

–26

5

4

3

2

1

0

6

x

0

y

11

z

8

8/8

14/14

5/5

s

–26

5

4

3

2

1

0

6

x

5

y

6

z

8

8/8

14/14

5

s

–20

5

4

3

2

1

0

6

x

5

y

0

z

8

5

8/8

8/14

s

x

z

10

s

x

z

11

s

x

z

12

s

x

z

13

s

x

z

14

s

x

z

15

(f)

(d)

(e)

(g)

Figure 26.9, continued (d) In iteration 10, .y; s/ is inadmissible, but iteration 11 pushes 5 units

of excess ﬂow from y to x. (e) Because y:current did not advance in iteration 11, iteration 12

ﬁnds .y; x/ to be inadmissible. Iteration 13 ﬁnds .y; ´/ inadmissible, and iteration 14 relabels ver-

tex y and resets y:current. (f) Iteration 15 pushes 6 units of excess ﬂow from y to s. (g) Vertex y

now has no excess ﬂow, and DISCHARGE terminates. In this example, DISCHARGE both starts and

ﬁnishes with the current pointer at the head of the neighbor list, but in general this need not be the

case.

754 Chapter 26 Maximum Flow

To prove the second statement, according to the test in line 1 and Lemma 26.28,

we need only show that all edges leaving u are inadmissible. If a call to

DISCHARGE.u/ starts with the pointer u:current at the head of u’s neighbor list

and ﬁnishes with it off the end of the list, then all of u’s outgoing edges are in-

admissible and a relabel operation applies. It is possible, however, that during a

call to DISCHARGE.u/, the pointer u:current traverses only part of the list be-

fore the procedure returns. Calls to DISCHARGE on other vertices may then oc-

cur, but u:current will continue moving through the list during the next call to

DISCHARGE.u/. We now consider what happens during a complete pass through

the list, which begins at the head of u:N and ﬁnishes with u:current D NIL. Once

u:current reaches the end of the list, the procedure relabels u and begins a new

pass. For the u:current pointer to advance past a vertex 2 u:N during a pass, the

edge .u; / must be deemed inadmissible by the test in line 6. Thus, by the time

the pass completes, every edge leaving u has been determined to be inadmissible

at some time during the pass. The key observation is that at the end of the pass,

every edge leaving u is still inadmissible. Why? By Lemma 26.27, pushes cannot

create any admissible edges, regardless of which vertex the ﬂow is pushed from.

Thus, any admissible edge must be created by a relabel operation. But the vertex u

is not relabeled during the pass, and by Lemma 26.28, any other vertex that is

relabeled during the pass (resulting from a call of DISCHARGE./) has no entering

admissible edges after relabeling. Thus, at the end of the pass, all edges leaving u

remain inadmissible, which completes the proof.

The relabel-to-front algorithm

In the relabel-to-front algorithm, we maintain a linked list L consisting of all ver-

tices in V fs; tg. A key property is that the vertices in L are topologically sorted

according to the admissible network, as we shall see in the loop invariant that fol-

lows. (Recall from Lemma 26.26 that the admissible network is a dag.)

The pseudocode for the relabel-to-front algorithm assumes that the neighbor

lists u:N have already been created for each vertex u. It also assumes that u:next

points to the vertex that follows u in list L and that, as usual, u:next D NIL if u is

the last vertex in the list.

26.5 The relabel-to-front algorithm 755

RELABEL-TO-FRONT.G; s; t/

1 INITIALIZE-PREFLOW.G; s/

2 L D G:V fs; tg, in any order

3 for each vertex u 2 G:V fs; tg

4 u:current D u:N:head

5 u D L:head

6 while u ¤ NIL

7 old-height D u:h

8 DISCHARGE.u/

9 if u:h > old-height

10 move u to the front of list L

11 u D u:next

The relabel-to-front algorithm works as follows. Line 1 initializes the preﬂow

and heights to the same values as in the generic push-relabel algorithm. Line 2

initializes the list L to contain all potentially overﬂowing vertices, in any order.

Lines 3–4 initialize the current pointer of each vertex u to the ﬁrst vertex in u’s

neighbor list.

As Figure 26.10 illustrates, the while loop of lines 6–11 runs through the list L,

discharging vertices. Line 5 makes it start with the ﬁrst vertex in the list. Each

time through the loop, line 8 discharges a vertex u. If u was relabeled by the

DISCHARGE procedure, line 10 moves it to the front of list L. We can determine

whether u was relabeled by comparing its height before the discharge operation,

saved into the variable old-height in line 7, with its height afterward, in line 9.

Line 11 makes the next iteration of the while loop use the vertex following u in

list L. If line 10 moved u to the front of the list, the vertex used in the next iteration

is the one following u in its new position in the list.

To show that RELABEL-TO-FRONT computes a maximum ﬂow, we shall show

that it is an implementation of the generic push-relabel algorithm. First, ob-

serve that it performs push and relabel operations only when they apply, since

Lemma 26.29 guarantees that DISCHARGE performs them only when they apply.

It remains to show that when RELABEL-TO-FRONT terminates, no basic opera-

tions apply. The remainder of the correctness argument relies on the following

loop invariant:

At each test in line 6 of RELABEL-TO-FRONT, list L is a topological sort

of the vertices in the admissible network G f;h D .V; E f;h /, and no vertex

before u in the list has excess ﬂow.

Initialization: Immediately after INITIALIZE-PREFLOW has been run, s:h D jV j

and :h D 0 for all 2 V fsg. Since jV j 2 (because V contains at

756 Chapter 26 Maximum Flow

s

–26

5

4

3

2

1

0

6

x

12

y

14

z

0

t

0

5 8 10

7

16

14/14

12/12

L: x y z

N: s

y

z

t

s

x

z

x

y

t

(a)

s

–26

5

4

3

2

1

0

6

x

0

y

19

z

0

t

75/5

8 10

14/14

12/12

L: x y z

N: s

y

z

t

s

x

z

x

y

t

(b)

7

7/16

s

–20

5

4

3

2

1

0

6

x

5

y

0

z

8

t

7

5

8/8

10

8/14

12/12

L: x y z

N: s

y

z

t

s

x

z

x

y

t

(c)

7

7/16

Figure 26.10 The action of RELABEL-TO-FRONT. (a) A ﬂow network just before the ﬁrst iteration

of the while loop. Initially, 26 units of ﬂow leave source s. On the right is shown the initial list

L D hx; y; ´i, where initially u D x. Under each vertex in list L is its neighbor list, with the current

neighbor shaded. Vertex x is discharged. It is relabeled to height 1, 5 units of excess ﬂow are pushed

to y, and the 7 remaining units of excess are pushed to the sink t. Because x is relabeled, it moves

to the head of L, which in this case does not change the structure of L. (b) After x, the next vertex

in L that is discharged is y. Figure 26.9 shows the detailed action of discharging y in this situation.

Because y is relabeled, it is moved to the head of L. (c) Vertex x now follows y in L, and so it is

again discharged, pushing all 5 units of excess ﬂow to t. Because vertex x is not relabeled in this

discharge operation, it remains in place in list L.

26.5 The relabel-to-front algorithm 757

s

–20

5

4

3

2

1

0

6

x

0

y

0

z

8

t

12

5

8/8

10

8/14

12/12

L: x y z

N: s

y

z

t

s

x

z

x

y

t

(d)

7

12/16

s

–20

5

4

3

2

1

0

6

x

0

y

0

z

0

t

20

5

8/8

8/10

8/14

12/12

L: x y z

N: s

y

z

t

s

x

z

x

y

t

(e)

12/16

7

Figure 26.10, continued (d) Since vertex ´ follows vertex x in L, it is discharged. It is relabeled

to height 1 and all 8 units of excess ﬂow are pushed to t. Because ´ is relabeled, it moves to the

front of L. (e) Vertex y now follows vertex ´ in L and is therefore discharged. But because y has no

excess, DISCHARGE immediately returns, and y remains in place in L. Vertex x is then discharged.

Because it, too, has no excess, DISCHARGE again returns, and x remains in place in L. RELABEL-

TO-FRONT has reached the end of list L and terminates. There are no overﬂowing vertices, and the

preﬂow is a maximum ﬂow.

least s and t), no edge can be admissible. Thus, E f;h D ;, and any ordering of

V fs; tg is a topological sort of G f;h .

Because u is initially the head of the list L, there are no vertices before it and

so there are none before it with excess ﬂow.

Maintenance: To see that each iteration of the while loop maintains the topolog-

ical sort, we start by observing that the admissible network is changed only by

push and relabel operations. By Lemma 26.27, push operations do not cause

edges to become admissible. Thus, only relabel operations can create admissi-

ble edges. After a vertex u is relabeled, however, Lemma 26.28 states that there

are no admissible edges entering u but there may be admissible edges leaving u.

Thus, by moving u to the front of L, the algorithm ensures that any admissible

edges leaving u satisfy the topological sort ordering.

758 Chapter 26 Maximum Flow

To see that no vertex preceding u in L has excess ﬂow, we denote the vertex

that will be u in the next iteration by u 0

. The vertices that will precede u 0

in the

next iteration include the current u (due to line 11) and either no other vertices

(if u is relabeled) or the same vertices as before (if u is not relabeled). When u

is discharged, it has no excess ﬂow afterward. Thus, if u is relabeled during

the discharge, no vertices preceding u 0

have excess ﬂow. If u is not relabeled

during the discharge, no vertices before it on the list acquired excess ﬂow during

this discharge, because L remained topologically sorted at all times during the

discharge (as just pointed out, admissible edges are created only by relabeling,

not pushing), and so each push operation causes excess ﬂow to move only to

vertices further down the list (or to s or t). Again, no vertices preceding u 0

have

excess ﬂow.

Termination: When the loop terminates, u is just past the end of L, and so the

loop invariant ensures that the excess of every vertex is 0. Thus, no basic oper-

ations apply.

Analysis

We shall now show that RELABEL-TO-FRONT runs in O.V 3 / time on any ﬂow

network G D .V; E/. Since the algorithm is an implementation of the generic

push-relabel algorithm, we shall take advantage of Corollary 26.21, which pro-

vides an O.V / bound on the number of relabel operations executed per vertex and

an O.V 2 / bound on the total number of relabel operations overall. In addition, Ex-

ercise 26.4-3 provides an O.VE/ bound on the total time spent performing relabel

operations, and Lemma 26.22 provides an O.VE/ bound on the total number of

saturating push operations.

Theorem 26.30

The running time of RELABEL-TO-FRONT on any ﬂow network G D .V; E/

is O.V 3 /.

Proof Let us consider a “phase” of the relabel-to-front algorithm to be the time

between two consecutive relabel operations. There are O.V 2 / phases, since there

are O.V 2 / relabel operations. Each phase consists of at most jV j calls to DIS-

CHARGE, which we can see as follows. If DISCHARGE does not perform a re-

label operation, then the next call to DISCHARGE is further down the list L, and

the length of L is less than jV j. If DISCHARGE does perform a relabel, the next

call to DISCHARGE belongs to a different phase. Since each phase contains at

most jV j calls to DISCHARGE and there are O.V 2 / phases, the number of times

DISCHARGE is called in line 8 of RELABEL-TO-FRONT is O.V 3 /. Thus, the total

26.5 The relabel-to-front algorithm 759

work performed by the while loop in RELABEL-TO-FRONT, excluding the work

performed within DISCHARGE, is at most O.V 3 /.

We must now bound the work performed within DISCHARGE during the ex-

ecution of the algorithm. Each iteration of the while loop within DISCHARGE

performs one of three actions. We shall analyze the total amount of work involved

in performing each of these actions.

We start with relabel operations (lines 4–5). Exercise 26.4-3 provides an O.VE/

time bound on all the O.V 2 / relabels that are performed.

Now, suppose that the action updates the u:current pointer in line 8. This action

occurs O.degree.u// times each time a vertex u is relabeled, and O.V degree.u//

times overall for the vertex. For all vertices, therefore, the total amount of work

done in advancing pointers in neighbor lists is O.VE/ by the handshaking lemma

(Exercise B.4-1).

The third type of action performed by DISCHARGE is a push operation (line 7).

We already know that the total number of saturating push operations is O.VE/.

Observe that if a nonsaturating push is executed, DISCHARGE immediately returns,

since the push reduces the excess to 0. Thus, there can be at most one nonsaturating

push per call to DISCHARGE. As we have observed, DISCHARGE is called O.V 3 /

times, and thus the total time spent performing nonsaturating pushes is O.V 3 /.

The running time of RELABEL-TO-FRONT is therefore O.V 3 C VE/, which

is O.V 3 /.

Exercises

26.5-1

Illustrate the execution of RELABEL-TO-FRONT in the manner of Figure 26.10 for

the ﬂow network in Figure 26.1(a). Assume that the initial ordering of vertices in L

is h 1 ; 2 ; 3 ; 4 i and that the neighbor lists are

1 :N D hs; 2 ; 3 i ;

2 :N D hs; 1 ; 3 ; 4 i ;

3 :N D h 1 ; 2 ; 4 ; ti ;

4 :N D h 2 ; 3 ; ti :

26.5-2 ?

We would like to implement a push-relabel algorithm in which we maintain a ﬁrst-

in, ﬁrst-out queue of overﬂowing vertices. The algorithm repeatedly discharges the

vertex at the head of the queue, and any vertices that were not overﬂowing before

the discharge but are overﬂowing afterward are placed at the end of the queue.

After the vertex at the head of the queue is discharged, it is removed. When the

760 Chapter 26 Maximum Flow

queue is empty, the algorithm terminates. Show how to implement this algorithm

to compute a maximum ﬂow in O.V 3 / time.

26.5-3

Show that the generic algorithm still works if RELABEL updates u:h by sim-

ply computing u:h D u:h C 1. How would this change affect the analysis of

RELABEL-TO-FRONT?

26.5-4 ?

Show that if we always discharge a highest overﬂowing vertex, we can make the

push-relabel method run in O.V 3 / time.

26.5-5

Suppose that at some point in the execution of a push-relabel algorithm, there exists

an integer 0 < k jV j 1 for which no vertex has :h D k. Show that all

vertices with :h > k are on the source side of a minimum cut. If such a k exists,

the gap heuristic updates every vertex 2 V fsg for which :h > k, to set

:h D max.:h; jV j C 1/. Show that the resulting attribute h is a height function.

(The gap heuristic is crucial in making implementations of the push-relabel method

perform well in practice.)

Problems

26-1 Escape problem

An n n grid is an undirected graph consisting of n rows and n columns of vertices,

as shown in Figure 26.11. We denote the vertex in the ith row and the j th column

by .i; j /. All vertices in a grid have exactly four neighbors, except for the boundary

vertices, which are the points .i; j / for which i D 1, i D n, j D 1, or j D n.

Given m n 2

starting points .x 1 ; y 1 /; .x 2 ; y 2 /; : : : ; .x m ; y m / in the grid, the

escape problem is to determine whether or not there are m vertex-disjoint paths

from the starting points to any m different points on the boundary. For example,

the grid in Figure 26.11(a) has an escape, but the grid in Figure 26.11(b) does not.

a. Consider a ﬂow network in which vertices, as well as edges, have capacities.

That is, the total positive ﬂow entering any given vertex is subject to a capacity

constraint. Show that determining the maximum ﬂow in a network with edge

and vertex capacities can be reduced to an ordinary maximum-ﬂow problem on

a ﬂow network of comparable size.

Problems for Chapter 26 761

(a) (b)

Figure 26.11 Grids for the escape problem. Starting points are black, and other grid vertices are

white. (a) A grid with an escape, shown by shaded paths. (b) A grid with no escape.

b. Describe an efﬁcient algorithm to solve the escape problem, and analyze its

running time.

26-2 Minimum path cover

A path cover of a directed graph G D .V; E/ is a set P of vertex-disjoint paths

such that every vertex in V is included in exactly one path in P . Paths may start

and end anywhere, and they may be of any length, including 0. A minimum path

cover of G is a path cover containing the fewest possible paths.

a. Give an efﬁcient algorithm to ﬁnd a minimum path cover of a directed acyclic

graph G D .V; E/. (Hint: Assuming that V D f1; 2; : : : ; ng, construct the

graph G 0 D .V 0 ; E 0 /, where

V

0

D fx 0 ; x 1 ; : : : ; x n g [ fy 0 ; y 1 ; : : : ; y n g ;

E

0

D f.x 0 ; x i / W i 2 V g [ f.y i ; y 0 / W i 2 V g [ f.x i ; y j / W .i; j / 2 Eg ;

and run a maximum-ﬂow algorithm.)

b. Does your algorithm work for directed graphs that contain cycles? Explain.

26-3 Algorithmic consulting

Professor Gore wants to open up an algorithmic consulting company. He has iden-

tiﬁed n important subareas of algorithms (roughly corresponding to different por-

tions of this textbook), which he represents by the set A D fA 1 ; A 2 ; : : : ; A n g. In

each subarea A k , he can hire an expert in that area for c k dollars. The consulting

company has lined up a set J D fJ 1 ; J 2 ; : : : ; J m g of potential jobs. In order to

perform job J i , the company needs to have hired experts in a subset R i A of

762 Chapter 26 Maximum Flow

subareas. Each expert can work on multiple jobs simultaneously. If the company

chooses to accept job J i , it must have hired experts in all subareas in R i , and it will

take in revenue of p i dollars.

Professor Gore’s job is to determine which subareas to hire experts in and which

jobs to accept in order to maximize the net revenue, which is the total income from

jobs accepted minus the total cost of employing the experts.

Consider the following ﬂow network G. It contains a source vertex s, vertices

A 1 ; A 2 ; : : : ; A n , vertices J 1 ; J 2 ; : : : ; J m , and a sink vertex t. For k D 1; 2 : : : ; n,

the ﬂow network contains an edge .s; A k / with capacity c.s; A k / D c k , and

for i D 1; 2; : : : ; m, the ﬂow network contains an edge .J i ; t/ with capacity

c.J i ; t/ D p i . For k D 1; 2; : : : ; n and i D 1; 2; : : : ; m, if A k 2 R i , then G

contains an edge .A k ; J i / with capacity c.A k ; J i / D 1.

a. Show that if J i 2 T for a ﬁnite-capacity cut .S; T / of G, then A k 2 T for each

A k 2 R i .

b. Show how to determine the maximum net revenue from the capacity of a mini-

mum cut of G and the given p i values.

c. Give an efﬁcient algorithm to determine which jobs to accept and which experts

to hire. Analyze the running time of your algorithm in terms of m, n, and

r D

P m

iD1

jR i j.

26-4 Updating maximum ﬂow

Let G D .V; E/ be a ﬂow network with source s, sink t, and integer capacities.

Suppose that we are given a maximum ﬂow in G.

a. Suppose that we increase the capacity of a single edge .u; / 2 E by 1. Give

an O.V C E/-time algorithm to update the maximum ﬂow.

b. Suppose that we decrease the capacity of a single edge .u; / 2 E by 1. Give

an O.V C E/-time algorithm to update the maximum ﬂow.

26-5 Maximum ﬂow by scaling

Let G D .V; E/ be a ﬂow network with source s, sink t, and an integer capac-

ity c.u; / on each edge .u; / 2 E. Let C D max .u;/2E c.u; /.

a. Argue that a minimum cut of G has capacity at most C jEj.

b. For a given number K, show how to ﬁnd an augmenting path of capacity at

least K in O.E/ time, if such a path exists.

Problems for Chapter 26 763

We can use the following modiﬁcation of FORD-FULKERSON-METHOD to com-

pute a maximum ﬂow in G:

MAX-FLOW-BY-SCALING.G; s; t/

1 C D max .u;/2E c.u; /

2 initialize ﬂow f to 0

3 K D 2 blg C c

4 while K 1

5 while there exists an augmenting path p of capacity at least K

6 augment ﬂow f along p

7 K D K=2

8 return f

c. Argue that MAX-FLOW-BY-SCALING returns a maximum ﬂow.

d. Show that the capacity of a minimum cut of the residual network G f is at most

2K jEj each time line 4 is executed.

e. Argue that the inner while loop of lines 5–6 executes O.E/ times for each value

of K.

f. Conclude that MAX-FLOW-BY-SCALING can be implemented so that it runs

in O.E 2

lg C / time.

26-6 The Hopcroft-Karp bipartite matching algorithm

In this problem, we describe a faster algorithm, due to Hopcroft and Karp, for

ﬁnding a maximum matching in a bipartite graph. The algorithm runs in O.

p

V E/

time. Given an undirected, bipartite graph G D .V; E/, where V D L [ R and

all edges have exactly one endpoint in L, let M be a matching in G. We say that

a simple path P in G is an augmenting path with respect to M if it starts at an

unmatched vertex in L, ends at an unmatched vertex in R, and its edges belong

alternately to M and E M . (This deﬁnition of an augmenting path is related

to, but different from, an augmenting path in a ﬂow network.) In this problem,

we treat a path as a sequence of edges, rather than as a sequence of vertices. A

shortest augmenting path with respect to a matching M is an augmenting path

with a minimum number of edges.

Given two sets A and B, the symmetric difference A˚B is deﬁned as .AB/[

.B A/, that is, the elements that are in exactly one of the two sets.

764 Chapter 26 Maximum Flow

a. Show that if M is a matching and P is an augmenting path with respect to M ,

then the symmetric difference M ˚ P is a matching and jM ˚ P j D jM j C 1.

Show that if P 1 ; P 2 ; : : : ; P k are vertex-disjoint augmenting paths with respect

to M , then the symmetric difference M ˚ .P 1 [ P 2 [ [ P k / is a matching

with cardinality jM j C k.

The general structure of our algorithm is the following:

HOPCROFT-KARP.G/

1 M D ;

2 repeat

3 let P D fP 1 ; P 2 ; : : : ; P k g be a maximal set of vertex-disjoint

shortest augmenting paths with respect to M

4 M D M ˚ .P 1 [ P 2 [ [ P k /

5 until P == ;

6 return M

The remainder of this problem asks you to analyze the number of iterations in

the algorithm (that is, the number of iterations in the repeat loop) and to describe

an implementation of line 3.

b. Given two matchings M and M

in G, show that every vertex in the graph

G 0 D .V; M ˚ M / has degree at most 2. Conclude that G 0

is a disjoint

union of simple paths or cycles. Argue that edges in each such simple path

or cycle belong alternately to M or M

. Prove that if jM j jM j, then

M ˚ M

contains at least jM j jM j vertex-disjoint augmenting paths with

respect to M .

Let l be the length of a shortest augmenting path with respect to a matching M , and

let P 1 ; P 2 ; : : : ; P k be a maximal set of vertex-disjoint augmenting paths of length l

with respect to M . Let M 0 D M ˚.P 1 [ [P k /, and suppose that P is a shortest

augmenting path with respect to M 0

.

c. Show that if P is vertex-disjoint from P 1 ; P 2 ; : : : ; P k , then P has more than l

edges.

d. Now suppose that P is not vertex-disjoint from P 1 ; P 2 ; : : : ; P k . Let A be the

set of edges .M ˚ M 0 / ˚ P . Show that A D .P 1 [ P 2 [ [ P k / ˚ P and

that jAj .k C 1/l. Conclude that P has more than l edges.

e. Prove that if a shortest augmenting path with respect to M has l edges, the size

of the maximum matching is at most jM j C jV j =.l C 1/.

Notes for Chapter 26 765

f. Show that the number of repeat loop iterations in the algorithm is at

most 2

p

jV j. (Hint: By how much can M grow after iteration number

p

jV j?)

g. Give an algorithm that runs in O.E/ time to ﬁnd a maximal set of vertex-

disjoint shortest augmenting paths P 1 ; P 2 ; : : : ; P k for a given matching M .

Conclude that the total running time of HOPCROFT-KARP is O.

p

V E/.

Chapter notes

Ahuja, Magnanti, and Orlin [7], Even [103], Lawler [224], Papadimitriou and Stei-

glitz [271], and Tarjan [330] are good references for network ﬂow and related algo-

rithms. Goldberg, Tardos, and Tarjan [139] also provide a nice survey of algorithms

for network-ﬂow problems, and Schrijver [304] has written an interesting review

of historical developments in the ﬁeld of network ﬂows.

The Ford-Fulkerson method is due to Ford and Fulkerson [109], who originated

the formal study of many of the problems in the area of network ﬂow, including

the maximum-ﬂow and bipartite-matching problems. Many early implementations

of the Ford-Fulkerson method found augmenting paths using breadth-ﬁrst search;

Edmonds and Karp [102], and independently Dinic [89], proved that this strategy

yields a polynomial-time algorithm. A related idea, that of using “blocking ﬂows,”

was also ﬁrst developed by Dinic [89]. Karzanov [202] ﬁrst developed the idea of

preﬂows. The push-relabel method is due to Goldberg [136] and Goldberg and Tar-

jan [140]. Goldberg and Tarjan gave an O.V 3 /-time algorithm that uses a queue to

maintain the set of overﬂowing vertices, as well as an algorithm that uses dynamic

trees to achieve a running time of O.VE lg.V 2 =E C 2//. Several other researchers

have developed push-relabel maximum-ﬂow algorithms. Ahuja and Orlin [9] and

Ahuja, Orlin, and Tarjan [10] gave algorithms that used scaling. Cheriyan and

Maheshwari [62] proposed pushing ﬂow from the overﬂowing vertex of maximum

height. Cheriyan and Hagerup [61] suggested randomly permuting the neighbor

lists, and several researchers [14, 204, 276] developed clever derandomizations of

this idea, leading to a sequence of faster algorithms. The algorithm of King, Rao,

and Tarjan [204] is the fastest such algorithm and runs in O.VE log

E=.V lg V /

V /

time.

The asymptotically fastest algorithm to date for the maximum-ﬂow problem, by

Goldberg and Rao [138], runs in time O.min.V 2=3 ; E 1=2 /E lg.V 2 =E C 2/ lg C /,

where C D max .u;/2E c.u; /. This algorithm does not use the push-relabel

method but instead is based on ﬁnding blocking ﬂows. All previous maximum-

ﬂow algorithms, including the ones in this chapter, use some notion of distance

(the push-relabel algorithms use the analogous notion of height), with a length of 1

766 Chapter 26 Maximum Flow

assigned implicitly to each edge. This new algorithm takes a different approach and

assigns a length of 0 to high-capacity edges and a length of 1 to low-capacity edges.

Informally, with respect to these lengths, shortest paths from the source to the sink

tend have high capacity, which means that fewer iterations need be performed.

In practice, push-relabel algorithms currently dominate augmenting-path or

linear-programming based algorithms for the maximum-ﬂow problem. A study

by Cherkassky and Goldberg [63] underscores the importance of using two heuris-

tics when implementing a push-relabel algorithm. The ﬁrst heuristic is to peri-

odically perform a breadth-ﬁrst search of the residual network in order to obtain

more accurate height values. The second heuristic is the gap heuristic, described in

Exercise 26.5-5. Cherkassky and Goldberg conclude that the best choice of push-

relabel variants is the one that chooses to discharge the overﬂowing vertex with the

maximum height.

The best algorithm to date for maximum bipartite matching, discovered by

Hopcroft and Karp [176], runs in O.

p

V E/ time and is described in Problem 26-6.

The book by Lov´asz and Plummer [239] is an excellent reference on matching

problems.

VII Selected Topics

Introduction

This part contains a selection of algorithmic topics that extend and complement

earlier material in this book. Some chapters introduce new models of computation

such as circuits or parallel computers. Others cover specialized domains such as

computational geometry or number theory. The last two chapters discuss some of

the known limitations to the design of efﬁcient algorithms and introduce techniques

for coping with those limitations.

Chapter 27 presents an algorithmic model for parallel computing based on dy-

namic multithreading. The chapter introduces the basics of the model, showing

how to quantify parallelism in terms of the measures of work and span. It then

investigates several interesting multithreaded algorithms, including algorithms for

matrix multiplication and merge sorting.

Chapter 28 studies efﬁcient algorithms for operating on matrices. It presents

two general methods—LU decomposition and LUP decomposition—for solving

linear equations by Gaussian elimination in O.n 3 / time. It also shows that matrix

inversion and matrix multiplication can be performed equally fast. The chapter

concludes by showing how to compute a least-squares approximate solution when

a set of linear equations has no exact solution.

Chapter 29 studies linear programming, in which we wish to maximize or mini-

mize an objective, given limited resources and competing constraints. Linear pro-

gramming arises in a variety of practical application areas. This chapter covers how

to formulate and solve linear programs. The solution method covered is the sim-

plex algorithm, which is the oldest algorithm for linear programming. In contrast

to many algorithms in this book, the simplex algorithm does not run in polynomial

time in the worst case, but it is fairly efﬁcient and widely used in practice.

770 Part VII Selected Topics

Chapter 30 studies operations on polynomials and shows how to use a well-

known signal-processing technique—the fast Fourier transform (FFT)—to multi-

ply two degree-n polynomials in O.n lg n/ time. It also investigates efﬁcient im-

plementations of the FFT, including a parallel circuit.

Chapter 31 presents number-theoretic algorithms. After reviewing elementary

number theory, it presents Euclid’s algorithm for computing greatest common di-

visors. Next, it studies algorithms for solving modular linear equations and for

raising one number to a power modulo another number. Then, it explores an impor-

tant application of number-theoretic algorithms: the RSA public-key cryptosystem.

This cryptosystem can be used not only to encrypt messages so that an adversary

cannot read them, but also to provide digital signatures. The chapter then presents

the Miller-Rabin randomized primality test, with which we can ﬁnd large primes

efﬁciently—an essential requirement for the RSA system. Finally, the chapter cov-

ers Pollard’s “rho” heuristic for factoring integers and discusses the state of the art

of integer factorization.

Chapter 32 studies the problem of ﬁnding all occurrences of a given pattern

string in a given text string, a problem that arises frequently in text-editing pro-

grams. After examining the naive approach, the chapter presents an elegant ap-

proach due to Rabin and Karp. Then, after showing an efﬁcient solution based

on ﬁnite automata, the chapter presents the Knuth-Morris-Pratt algorithm, which

modiﬁes the automaton-based algorithm to save space by cleverly preprocessing

the pattern.

Chapter 33 considers a few problems in computational geometry. After dis-

cussing basic primitives of computational geometry, the chapter shows how to use

a “sweeping” method to efﬁciently determine whether a set of line segments con-

tains any intersections. Two clever algorithms for ﬁnding the convex hull of a set of

points—Graham’s scan and Jarvis’s march—also illustrate the power of sweeping

methods. The chapter closes with an efﬁcient algorithm for ﬁnding the closest pair

from among a given set of points in the plane.

Chapter 34 concerns NP-complete problems. Many interesting computational

problems are NP-complete, but no polynomial-time algorithm is known for solving

any of them. This chapter presents techniques for determining when a problem is

NP-complete. Several classic problems are proved to be NP-complete: determining

whether a graph has a hamiltonian cycle, determining whether a boolean formula

is satisﬁable, and determining whether a given set of numbers has a subset that

adds up to a given target value. The chapter also proves that the famous traveling-

salesman problem is NP-complete.

Chapter 35 shows how to ﬁnd approximate solutions to NP-complete problems

efﬁciently by using approximation algorithms. For some NP-complete problems,

approximate solutions that are near optimal are quite easy to produce, but for others

even the best approximation algorithms known work progressively more poorly as

Part VII Selected Topics 771

the problem size increases. Then, there are some problems for which we can invest

increasing amounts of computation time in return for increasingly better approx-

imate solutions. This chapter illustrates these possibilities with the vertex-cover

problem (unweighted and weighted versions), an optimization version of 3-CNF

satisﬁability, the traveling-salesman problem, the set-covering problem, and the

subset-sum problem.

27 Multithreaded Algorithms

The vast majority of algorithms in this book are serial algorithms suitable for

running on a uniprocessor computer in which only one instruction executes at a

time. In this chapter, we shall extend our algorithmic model to encompass parallel

algorithms, which can run on a multiprocessor computer that permits multiple

instructions to execute concurrently. In particular, we shall explore the elegant

model of dynamic multithreaded algorithms, which are amenable to algorithmic

design and analysis, as well as to efﬁcient implementation in practice.

Parallel computers—computers with multiple processing units—have become

increasingly common, and they span a wide range of prices and performance. Rela-

tively inexpensive desktop and laptop chip multiprocessors contain a single multi-

core integrated-circuit chip that houses multiple processing “cores,” each of which

is a full-ﬂedged processor that can access a common memory. At an intermedi-

ate price/performance point are clusters built from individual computers—often

simple PC-class machines—with a dedicated network interconnecting them. The

highest-priced machines are supercomputers, which often use a combination of

custom architectures and custom networks to deliver the highest performance in

terms of instructions executed per second.

Multiprocessor computers have been around, in one form or another, for

decades. Although the computing community settled on the random-access ma-

chine model for serial computing early on in the history of computer science, no

single model for parallel computing has gained as wide acceptance. A major rea-

son is that vendors have not agreed on a single architectural model for parallel

computers. For example, some parallel computers feature shared memory, where

each processor can directly access any location of memory. Other parallel com-

puters employ distributed memory, where each processor’s memory is private, and

an explicit message must be sent between processors in order for one processor to

access the memory of another. With the advent of multicore technology, however,

every new laptop and desktop machine is now a shared-memory parallel computer,

Chapter 27 Multithreaded Algorithms 773

and the trend appears to be toward shared-memory multiprocessing. Although time

will tell, that is the approach we shall take in this chapter.

One common means of programming chip multiprocessors and other shared-

memory parallel computers is by using static threading, which provides a software

abstraction of “virtual processors,” or threads, sharing a common memory. Each

thread maintains an associated program counter and can execute code indepen-

dently of the other threads. The operating system loads a thread onto a processor

for execution and switches it out when another thread needs to run. Although the

operating system allows programmers to create and destroy threads, these opera-

tions are comparatively slow. Thus, for most applications, threads persist for the

duration of a computation, which is why we call them “static.”

Unfortunately, programming a shared-memory parallel computer directly using

static threads is difﬁcult and error-prone. One reason is that dynamically parti-

tioning the work among the threads so that each thread receives approximately

the same load turns out to be a complicated undertaking. For any but the sim-

plest of applications, the programmer must use complex communication protocols

to implement a scheduler to load-balance the work. This state of affairs has led

toward the creation of concurrency platforms, which provide a layer of software

that coordinates, schedules, and manages the parallel-computing resources. Some

concurrency platforms are built as runtime libraries, but others provide full-ﬂedged

parallel languages with compiler and runtime support.

Dynamic multithreaded programming

One important class of concurrency platform is dynamic multithreading, which is

the model we shall adopt in this chapter. Dynamic multithreading allows program-

mers to specify parallelism in applications without worrying about communication

protocols, load balancing, and other vagaries of static-thread programming. The

concurrency platform contains a scheduler, which load-balances the computation

automatically, thereby greatly simplifying the programmer’s chore. Although the

functionality of dynamic-multithreading environments is still evolving, almost all

support two features: nested parallelism and parallel loops. Nested parallelism

allows a subroutine to be “spawned,” allowing the caller to proceed while the

spawned subroutine is computing its result. A parallel loop is like an ordinary

for loop, except that the iterations of the loop can execute concurrently.

These two features form the basis of the model for dynamic multithreading that

we shall study in this chapter. A key aspect of this model is that the programmer

needs to specify only the logical parallelism within a computation, and the threads

within the underlying concurrency platform schedule and load-balance the compu-

tation among themselves. We shall investigate multithreaded algorithms written for

774 Chapter 27 Multithreaded Algorithms

this model, as well how the underlying concurrency platform can schedule compu-

tations efﬁciently.

Our model for dynamic multithreading offers several important advantages:

It is a simple extension of our serial programming model. We can describe a

multithreaded algorithm by adding to our pseudocode just three “concurrency”

keywords: parallel, spawn, and sync. Moreover, if we delete these concur-

rency keywords from the multithreaded pseudocode, the resulting text is serial

pseudocode for the same problem, which we call the “serialization” of the mul-

tithreaded algorithm.

It provides a theoretically clean way to quantify parallelism based on the no-

tions of “work” and “span.”

Many multithreaded algorithms involving nested parallelism follow naturally

from the divide-and-conquer paradigm. Moreover, just as serial divide-and-

conquer algorithms lend themselves to analysis by solving recurrences, so do

multithreaded algorithms.

The model is faithful to how parallel-computing practice is evolving. A grow-

ing number of concurrency platforms support one variant or another of dynamic

multithreading, including Cilk [51, 118], Cilk++ [71], OpenMP [59], Task Par-

allel Library [230], and Threading Building Blocks [292].

Section 27.1 introduces the dynamic multithreading model and presents the met-

rics of work, span, and parallelism, which we shall use to analyze multithreaded

algorithms. Section 27.2 investigates how to multiply matrices with multithread-

ing, and Section 27.3 tackles the tougher problem of multithreading merge sort.

27.1 The basics of dynamic multithreading

We shall begin our exploration of dynamic multithreading using the example of

computing Fibonacci numbers recursively. Recall that the Fibonacci numbers are

deﬁned by recurrence (3.22):

F 0 D 0 ;

F 1 D 1 ;

F i D F i1 C F i2 for i 2 :

Here is a simple, recursive, serial algorithm to compute the nth Fibonacci number:

27.1 The basics of dynamic multithreading 775

FIB.0/

FIB.0/ FIB.0/ FIB.0/

FIB.0/

FIB.1/ FIB.1/

FIB.1/

FIB.1/

FIB.1/ FIB.1/ FIB.1/

FIB.1/

FIB.2/

FIB.2/ FIB.2/ FIB.2/

FIB.2/

FIB.3/ FIB.3/

FIB.3/

FIB.4/

FIB.4/

FIB.5/

FIB.6/

Figure 27.1 The tree of recursive procedure instances when computing FIB.6/. Each instance of

FIB with the same argument does the same work to produce the same result, providing an inefﬁcient

but interesting way to compute Fibonacci numbers.

FIB.n/

1 if n 1

2 return n

3 else x D FIB.n 1/

4 y D FIB.n 2/

5 return x C y

You would not really want to compute large Fibonacci numbers this way, be-

cause this computation does much repeated work. Figure 27.1 shows the tree of

recursive procedure instances that are created when computing F 6 . For example,

a call to FIB.6/ recursively calls FIB.5/ and then FIB.4/. But, the call to FIB.5/

also results in a call to FIB.4/. Both instances of FIB.4/ return the same result

(F 4 D 3). Since the FIB procedure does not memoize, the second call to FIB.4/

replicates the work that the ﬁrst call performs.

Let T .n/ denote the running time of FIB.n/. Since FIB.n/ contains two recur-

sive calls plus a constant amount of extra work, we obtain the recurrence

T .n/ D T .n 1/ C T .n 2/ C ‚.1/ :

This recurrence has solution T .n/ D ‚.F n /, which we can show using the substi-

tution method. For an inductive hypothesis, assume that T .n/ aF n b, where

a > 1 and b > 0 are constants. Substituting, we obtain

776 Chapter 27 Multithreaded Algorithms

T .n/ .aF n1 b/ C .aF n2 b/ C ‚.1/

D a.F n1 C F n2 / 2b C ‚.1/

D aF n b .b ‚.1//

aF n b

if we choose b large enough to dominate the constant in the ‚.1/. We can then

choose a large enough to satisfy the initial condition. The analytical bound

T .n/ D ‚.

n

/ ; (27.1)

where D .1 C

p

5/=2 is the golden ratio, now follows from equation (3.25).

Since F n grows exponentially in n, this procedure is a particularly slow way to

compute Fibonacci numbers. (See Problem 31-3 for much faster ways.)

Although the FIB procedure is a poor way to compute Fibonacci numbers, it

makes a good example for illustrating key concepts in the analysis of multithreaded

algorithms. Observe that within FIB.n/, the two recursive calls in lines 3 and 4 to

FIB.n 1/ and FIB.n 2/, respectively, are independent of each other: they could

be called in either order, and the computation performed by one in no way affects

the other. Therefore, the two recursive calls can run in parallel.

We augment our pseudocode to indicate parallelism by adding the concurrency

keywords spawn and sync. Here is how we can rewrite the FIB procedure to use

dynamic multithreading:

P-FIB.n/

1 if n 1

2 return n

3 else x D spawn P-FIB.n 1/

4 y D P-FIB.n 2/

5 sync

6 return x C y

Notice that if we delete the concurrency keywords spawn and sync from P-FIB,

the resulting pseudocode text is identical to FIB (other than renaming the procedure

in the header and in the two recursive calls). We deﬁne the serialization of a mul-

tithreaded algorithm to be the serial algorithm that results from deleting the multi-

threaded keywords: spawn, sync, and when we examine parallel loops, parallel.

Indeed, our multithreaded pseudocode has the nice property that a serialization is

always ordinary serial pseudocode to solve the same problem.

Nested parallelism occurs when the keyword spawn precedes a procedure call,

as in line 3. The semantics of a spawn differs from an ordinary procedure call in

that the procedure instance that executes the spawn—the parent—may continue

to execute in parallel with the spawned subroutine—its child—instead of waiting

27.1 The basics of dynamic multithreading 777

for the child to complete, as would normally happen in a serial execution. In this

case, while the spawned child is computing P-FIB.n 1/, the parent may go on

to compute P-FIB.n 2/ in line 4 in parallel with the spawned child. Since the

P-FIB procedure is recursive, these two subroutine calls themselves create nested

parallelism, as do their children, thereby creating a potentially vast tree of subcom-

putations, all executing in parallel.

The keyword spawn does not say, however, that a procedure must execute con-

currently with its spawned children, only that it may. The concurrency keywords

express the logical parallelism of the computation, indicating which parts of the

computation may proceed in parallel. At runtime, it is up to a scheduler to deter-

mine which subcomputations actually run concurrently by assigning them to avail-

able processors as the computation unfolds. We shall discuss the theory behind

schedulers shortly.

A procedure cannot safely use the values returned by its spawned children until

after it executes a sync statement, as in line 5. The keyword sync indicates that

the procedure must wait as necessary for all its spawned children to complete be-

fore proceeding to the statement after the sync. In the P-FIB procedure, a sync

is required before the return statement in line 6 to avoid the anomaly that would

occur if x and y were summed before x was computed. In addition to explicit

synchronization provided by the sync statement, every procedure executes a sync

implicitly before it returns, thus ensuring that all its children terminate before it

does.

A model for multithreaded execution

It helps to think of a multithreaded computation—the set of runtime instruc-

tions executed by a processor on behalf of a multithreaded program—as a directed

acyclic graph G D .V; E/, called a computation dag. As an example, Figure 27.2

shows the computation dag that results from computing P-FIB.4/. Conceptually,

the vertices in V are instructions, and the edges in E represent dependencies be-

tween instructions, where .u; / 2 E means that instruction u must execute before

instruction . For convenience, however, if a chain of instructions contains no

parallel control (no spawn, sync, or return from a spawn—via either an explicit

return statement or the return that happens implicitly upon reaching the end of

a procedure), we may group them into a single strand, each of which represents

one or more instructions. Instructions involving parallel control are not included

in strands, but are represented in the structure of the dag. For example, if a strand

has two successors, one of them must have been spawned, and a strand with mul-

tiple predecessors indicates the predecessors joined because of a sync statement.

Thus, in the general case, the set V forms the set of strands, and the set E of di-

rected edges represents dependencies between strands induced by parallel control.

778 Chapter 27 Multithreaded Algorithms

P-FIB(1) P-FIB(0)

P-FIB(3)

P-FIB(4)

P-FIB(1)

P-FIB(1)

P-FIB(0)

P-FIB(2)

P-FIB(2)

Figure 27.2 A directed acyclic graph representing the computation of P-FIB.4/. Each circle rep-

resents one strand, with black circles representing either base cases or the part of the procedure

(instance) up to the spawn of P-FIB.n 1/ in line 3, shaded circles representing the part of the pro-

cedure that calls P-FIB.n 2/ in line 4 up to the sync in line 5, where it suspends until the spawn of

P-FIB.n 1/ returns, and white circles representing the part of the procedure after the sync where

it sums x and y up to the point where it returns the result. Each group of strands belonging to the

same procedure is surrounded by a rounded rectangle, lightly shaded for spawned procedures and

heavily shaded for called procedures. Spawn edges and call edges point downward, continuation

edges point horizontally to the right, and return edges point upward. Assuming that each strand takes

unit time, the work equals 17 time units, since there are 17 strands, and the span is 8 time units, since

the critical path—shown with shaded edges—contains 8 strands.

If G has a directed path from strand u to strand , we say that the two strands are

(logically) in series. Otherwise, strands u and are (logically) in parallel.

We can picture a multithreaded computation as a dag of strands embedded in a

tree of procedure instances. For example, Figure 27.1 shows the tree of procedure

instances for P-FIB.6/ without the detailed structure showing strands. Figure 27.2

zooms in on a section of that tree, showing the strands that constitute each proce-

dure. All directed edges connecting strands run either within a procedure or along

undirected edges in the procedure tree.

We can classify the edges of a computation dag to indicate the kind of dependen-

cies between the various strands. A continuation edge .u; u 0 /, drawn horizontally

in Figure 27.2, connects a strand u to its successor u 0

within the same procedure

instance. When a strand u spawns a strand , the dag contains a spawn edge .u; /,

which points downward in the ﬁgure. Call edges, representing normal procedure

calls, also point downward. Strand u spawning strand differs from u calling

in that a spawn induces a horizontal continuation edge from u to the strand u 0

fol-

27.1 The basics of dynamic multithreading 779

lowing u in its procedure, indicating that u 0

is free to execute at the same time

as , whereas a call induces no such edge. When a strand u returns to its calling

procedure and x is the strand immediately following the next sync in the calling

procedure, the computation dag contains return edge .u; x/, which points upward.

A computation starts with a single initial strand—the black vertex in the procedure

labeled P-FIB.4/ in Figure 27.2—and ends with a single ﬁnal strand—the white

vertex in the procedure labeled P-FIB.4/.

We shall study the execution of multithreaded algorithms on an ideal paral-

lel computer, which consists of a set of processors and a sequentially consistent

shared memory. Sequential consistency means that the shared memory, which may

in reality be performing many loads and stores from the processors at the same

time, produces the same results as if at each step, exactly one instruction from one

of the processors is executed. That is, the memory behaves as if the instructions

were executed sequentially according to some global linear order that preserves the

individual orders in which each processor issues its own instructions. For dynamic

multithreaded computations, which are scheduled onto processors automatically

by the concurrency platform, the shared memory behaves as if the multithreaded

computation’s instructions were interleaved to produce a linear order that preserves

the partial order of the computation dag. Depending on scheduling, the ordering

could differ from one run of the program to another, but the behavior of any exe-

cution can be understood by assuming that the instructions are executed in some

linear order consistent with the computation dag.

In addition to making assumptions about semantics, the ideal-parallel-computer

model makes some performance assumptions. Speciﬁcally, it assumes that each

processor in the machine has equal computing power, and it ignores the cost of

scheduling. Although this last assumption may sound optimistic, it turns out that

for algorithms with sufﬁcient “parallelism” (a term we shall deﬁne precisely in a

moment), the overhead of scheduling is generally minimal in practice.

Performance measures

We can gauge the theoretical efﬁciency of a multithreaded algorithm by using two

metrics: “work” and “span.” The work of a multithreaded computation is the total

time to execute the entire computation on one processor. In other words, the work

is the sum of the times taken by each of the strands. For a computation dag in

which each strand takes unit time, the work is just the number of vertices in the

dag. The span is the longest time to execute the strands along any path in the dag.

Again, for a dag in which each strand takes unit time, the span equals the number of

vertices on a longest or critical path in the dag. (Recall from Section 24.2 that we

can ﬁnd a critical path in a dag G D .V; E/ in ‚.V C E/ time.) For example, the

computation dag of Figure 27.2 has 17 vertices in all and 8 vertices on its critical

780 Chapter 27 Multithreaded Algorithms

path, so that if each strand takes unit time, its work is 17 time units and its span

is 8 time units.

The actual running time of a multithreaded computation depends not only on

its work and its span, but also on how many processors are available and how

the scheduler allocates strands to processors. To denote the running time of a

multithreaded computation on P processors, we shall subscript by P . For example,

we might denote the running time of an algorithm on P processors by T P . The

work is the running time on a single processor, or T 1 . The span is the running time

if we could run each strand on its own processor—in other words, if we had an

unlimited number of processors—and so we denote the span by T 1 .

The work and span provide lower bounds on the running time T P of a multi-

threaded computation on P processors:

In one step, an ideal parallel computer with P processors can do at most P

units of work, and thus in T P time, it can perform at most P T P work. Since the

total work to do is T 1 , we have P T P T 1 . Dividing by P yields the work law:

T P T 1 =P : (27.2)

A P -processor ideal parallel computer cannot run any faster than a machine

with an unlimited number of processors. Looked at another way, a machine

with an unlimited number of processors can emulate a P -processor machine by

using just P of its processors. Thus, the span law follows:

T P T 1 : (27.3)

We deﬁne the speedup of a computation on P processors by the ratio T 1 =T P ,

which says how many times faster the computation is on P processors than

on 1 processor. By the work law, we have T P T 1 =P , which implies that

T 1 =T P P . Thus, the speedup on P processors can be at most P . When the

speedup is linear in the number of processors, that is, when T 1 =T P D ‚.P /, the

computation exhibits linear speedup, and when T 1 =T P D P , we have perfect

linear speedup.

The ratio T 1 =T 1 of the work to the span gives the parallelism of the multi-

threaded computation. We can view the parallelism from three perspectives. As a

ratio, the parallelism denotes the average amount of work that can be performed in

parallel for each step along the critical path. As an upper bound, the parallelism

gives the maximum possible speedup that can be achieved on any number of pro-

cessors. Finally, and perhaps most important, the parallelism provides a limit on

the possibility of attaining perfect linear speedup. Speciﬁcally, once the number of

processors exceeds the parallelism, the computation cannot possibly achieve per-

fect linear speedup. To see this last point, suppose that P > T 1 =T 1 , in which case

27.1 The basics of dynamic multithreading 781

the span law implies that the speedup satisﬁes T 1 =T P T 1 =T 1 < P . Moreover,

if the number P of processors in the ideal parallel computer greatly exceeds the

parallelism—that is, if P   
 T 1 =T 1 —then T 1 =T P P , so that the speedup is

much less than the number of processors. In other words, the more processors we

use beyond the parallelism, the less perfect the speedup.

As an example, consider the computation P-FIB.4/ in Figure 27.2, and assume

that each strand takes unit time. Since the work is T 1 D 17 and the span is T 1 D 8,

the parallelism is T 1 =T 1 D 17=8 D 2:125. Consequently, achieving much more

than double the speedup is impossible, no matter how many processors we em-

ploy to execute the computation. For larger input sizes, however, we shall see that

P-FIB.n/ exhibits substantial parallelism.

We deﬁne the (parallel) slackness of a multithreaded computation executed

on an ideal parallel computer with P processors to be the ratio .T 1 =T 1 /=P D

T 1 =.P T 1 /, which is the factor by which the parallelism of the computation ex-

ceeds the number of processors in the machine. Thus, if the slackness is less than 1,

we cannot hope to achieve perfect linear speedup, because T 1 =.P T 1 / < 1 and the

span law imply that the speedup on P processors satisﬁes T 1 =T P T 1 =T 1 < P .

Indeed, as the slackness decreases from 1 toward 0, the speedup of the computation

diverges further and further from perfect linear speedup. If the slackness is greater

than 1, however, the work per processor is the limiting constraint. As we shall see,

as the slackness increases from 1, a good scheduler can achieve closer and closer

to perfect linear speedup.

Scheduling

Good performance depends on more than just minimizing the work and span. The

strands must also be scheduled efﬁciently onto the processors of the parallel ma-

chine. Our multithreaded programming model provides no way to specify which

strands to execute on which processors. Instead, we rely on the concurrency plat-

form’s scheduler to map the dynamically unfolding computation to individual pro-

cessors. In practice, the scheduler maps the strands to static threads, and the op-

erating system schedules the threads on the processors themselves, but this extra

level of indirection is unnecessary for our understanding of scheduling. We can

just imagine that the concurrency platform’s scheduler maps strands to processors

directly.

A multithreaded scheduler must schedule the computation with no advance

knowledge of when strands will be spawned or when they will complete—it must

operate on-line. Moreover, a good scheduler operates in a distributed fashion,

where the threads implementing the scheduler cooperate to load-balance the com-

putation. Provably good on-line, distributed schedulers exist, but analyzing them

is complicated.

782 Chapter 27 Multithreaded Algorithms

Instead, to keep our analysis simple, we shall investigate an on-line centralized

scheduler, which knows the global state of the computation at any given time. In

particular, we shall analyze greedy schedulers, which assign as many strands to

processors as possible in each time step. If at least P strands are ready to execute

during a time step, we say that the step is a complete step, and a greedy scheduler

assigns any P of the ready strands to processors. Otherwise, fewer than P strands

are ready to execute, in which case we say that the step is an incomplete step, and

the scheduler assigns each ready strand to its own processor.

From the work law, the best running time we can hope for on P processors

is T P D T 1 =P , and from the span law the best we can hope for is T P D T 1 .

The following theorem shows that greedy scheduling is provably good in that it

achieves the sum of these two lower bounds as an upper bound.

Theorem 27.1

On an ideal parallel computer with P processors, a greedy scheduler executes a

multithreaded computation with work T 1 and span T 1 in time

T P T 1 =P C T 1 : (27.4)

Proof We start by considering the complete steps. In each complete step, the

P processors together perform a total of P work. Suppose for the purpose of

contradiction that the number of complete steps is strictly greater than bT 1 =P c.

Then, the total work of the complete steps is at least

P .bT 1 =P c C 1/ D P bT 1 =P c C P

D T 1 .T 1 mod P / C P (by equation (3.8))

> T 1 (by inequality (3.9)) .

Thus, we obtain the contradiction that the P processors would perform more work

than the computation requires, which allows us to conclude that the number of

complete steps is at most bT 1 =P c.

Now, consider an incomplete step. Let G be the dag representing the entire

computation, and without loss of generality, assume that each strand takes unit

time. (We can replace each longer strand by a chain of unit-time strands.) Let G 0

be the subgraph of G that has yet to be executed at the start of the incomplete step,

and let G 00

be the subgraph remaining to be executed after the incomplete step. A

longest path in a dag must necessarily start at a vertex with in-degree 0. Since an

incomplete step of a greedy scheduler executes all strands with in-degree 0 in G 0

,

the length of a longest path in G 00

must be 1 less than the length of a longest path

in G 0

. In other words, an incomplete step decreases the span of the unexecuted dag

by 1. Hence, the number of incomplete steps is at most T 1 .

Since each step is either complete or incomplete, the theorem follows.

27.1 The basics of dynamic multithreading 783

The following corollary to Theorem 27.1 shows that a greedy scheduler always

performs well.

Corollary 27.2

The running time T P of any multithreaded computation scheduled by a greedy

scheduler on an ideal parallel computer with P processors is within a factor of 2

of optimal.

Proof Let T

P

be the running time produced by an optimal scheduler on a machine

with P processors, and let T 1 and T 1 be the work and span of the computation,

respectively. Since the work and span laws—inequalities (27.2) and (27.3)—give

us T

P

max.T 1 =P; T 1 /, Theorem 27.1 implies that

T P T 1 =P C T 1

2 max.T 1 =P; T 1 /

2T

P

:

The next corollary shows that, in fact, a greedy scheduler achieves near-perfect

linear speedup on any multithreaded computation as the slackness grows.

Corollary 27.3

Let T P be the running time of a multithreaded computation produced by a greedy

scheduler on an ideal parallel computer with P processors, and let T 1 and T 1 be

the work and span of the computation, respectively. Then, if P T 1 =T 1 , we

have T P T 1 =P , or equivalently, a speedup of approximately P .

Proof If we suppose that P T 1 =T 1 , then we also have T 1 T 1 =P , and

hence Theorem 27.1 gives us T P T 1 =P C T 1 T 1 =P . Since the work

law (27.2) dictates that T P T 1 =P , we conclude that T P T 1 =P , or equiva-

lently, that the speedup is T 1 =T P P .

The symbol denotes “much less,” but how much is “much less”? As a rule

of thumb, a slackness of at least 10—that is, 10 times more parallelism than pro-

cessors—generally sufﬁces to achieve good speedup. Then, the span term in the

greedy bound, inequality (27.4), is less than 10% of the work-per-processor term,

which is good enough for most engineering situations. For example, if a computa-

tion runs on only 10 or 100 processors, it doesn’t make sense to value parallelism

of, say 1,000,000 over parallelism of 10,000, even with the factor of 100 differ-

ence. As Problem 27-2 shows, sometimes by reducing extreme parallelism, we

can obtain algorithms that are better with respect to other concerns and which still

scale up well on reasonable numbers of processors.

784 Chapter 27 Multithreaded Algorithms

A

(a) (b)

B

A

B

Work: T 1 .A [ B/ D T 1 .A/ C T 1 .B/

Span: T 1 .A [ B/ D T 1 .A/ C T 1 .B/

Work: T 1 .A [ B/ D T 1 .A/ C T 1 .B/

Span: T 1 .A [ B/ D max.T 1 .A/; T 1 .B/)

Figure 27.3 The work and span of composed subcomputations. (a) When two subcomputations

are joined in series, the work of the composition is the sum of their work, and the span of the

composition is the sum of their spans. (b) When two subcomputations are joined in parallel, the

work of the composition remains the sum of their work, but the span of the composition is only the

maximum of their spans.

Analyzing multithreaded algorithms

We now have all the tools we need to analyze multithreaded algorithms and provide

good bounds on their running times on various numbers of processors. Analyzing

the work is relatively straightforward, since it amounts to nothing more than ana-

lyzing the running time of an ordinary serial algorithm—namely, the serialization

of the multithreaded algorithm—which you should already be familiar with, since

that is what most of this textbook is about! Analyzing the span is more interesting,

but generally no harder once you get the hang of it. We shall investigate the basic

ideas using the P-FIB program.

Analyzing the work T 1 .n/ of P-FIB.n/ poses no hurdles, because we’ve already

done it. The original FIB procedure is essentially the serialization of P-FIB, and

hence T 1 .n/ D T .n/ D ‚. n / from equation (27.1).

Figure 27.3 illustrates how to analyze the span. If two subcomputations are

joined in series, their spans add to form the span of their composition, whereas

if they are joined in parallel, the span of their composition is the maximum of the

spans of the two subcomputations. For P-FIB.n/, the spawned call to P-FIB.n1/

in line 3 runs in parallel with the call to P-FIB.n 2/ in line 4. Hence, we can

express the span of P-FIB.n/ as the recurrence

T 1 .n/ D max.T 1 .n 1/; T 1 .n 2// C ‚.1/

D T 1 .n 1/ C ‚.1/ ;

which has solution T 1 .n/ D ‚.n/.

The parallelism of P-FIB.n/ is T 1 .n/=T 1 .n/ D ‚. n =n/, which grows dra-

matically as n gets large. Thus, on even the largest parallel computers, a modest

27.1 The basics of dynamic multithreading 785

value for n sufﬁces to achieve near perfect linear speedup for P-FIB.n/, because

this procedure exhibits considerable parallel slackness.

Parallel loops

Many algorithms contain loops all of whose iterations can operate in parallel. As

we shall see, we can parallelize such loops using the spawn and sync keywords,

but it is much more convenient to specify directly that the iterations of such loops

can run concurrently. Our pseudocode provides this functionality via the parallel

concurrency keyword, which precedes the for keyword in a for loop statement.

As an example, consider the problem of multiplying an n n matrix A D .a ij /

by an n-vector x D .x j /. The resulting n-vector y D .y i / is given by the equation

y i D

n X

j D1

a ij x j ;

for i D 1; 2; : : : ; n. We can perform matrix-vector multiplication by computing all

the entries of y in parallel as follows:

MAT-VEC.A; x/

1 n D A:rows

2 let y be a new vector of length n

3 parallel for i D 1 to n

4 y i D 0

5 parallel for i D 1 to n

6 for j D 1 to n

7 y i D y i C a ij x j

8 return y

In this code, the parallel for keywords in lines 3 and 5 indicate that the itera-

tions of the respective loops may be run concurrently. A compiler can implement

each parallel for loop as a divide-and-conquer subroutine using nested parallelism.

For example, the parallel for loop in lines 5–7 can be implemented with the call

MAT-VEC-MAIN-LOOP.A; x; y; n; 1; n/, where the compiler produces the auxil-

iary subroutine MAT-VEC-MAIN-LOOP as follows:

786 Chapter 27 Multithreaded Algorithms

1,1 2,2 3,3 4,4 5,5 6,6 7,7 8,8

1,2 3,4 5,6 7,8

1,4 5,8

1,8

Figure 27.4 A dag representing the computation of MAT-VEC-MAIN-LOOP.A; x; y; 8; 1; 8/. The

two numbers within each rounded rectangle give the values of the last two parameters (i and i

0

in

the procedure header) in the invocation (spawn or call) of the procedure. The black circles repre-

sent strands corresponding to either the base case or the part of the procedure up to the spawn of

MAT-VEC-MAIN-LOOP in line 5; the shaded circles represent strands corresponding to the part of

the procedure that calls MAT-VEC-MAIN-LOOP in line 6 up to the sync in line 7, where it suspends

until the spawned subroutine in line 5 returns; and the white circles represent strands corresponding

to the (negligible) part of the procedure after the sync up to the point where it returns.

MAT-VEC-MAIN-LOOP.A; x; y; n; i; i 0 /

1 if i == i 0

2 for j D 1 to n

3 y i D y i C a ij x j

4 else mid D b.i C i 0 /=2c

5 spawn MAT-VEC-MAIN-LOOP.A; x; y; n; i; mid/

6 MAT-VEC-MAIN-LOOP.A; x; y; n; mid C 1; i 0 /

7 sync

This code recursively spawns the ﬁrst half of the iterations of the loop to execute

in parallel with the second half of the iterations and then executes a sync, thereby

creating a binary tree of execution where the leaves are individual loop iterations,

as shown in Figure 27.4.

To calculate the work T 1 .n/ of MAT-VEC on an n n matrix, we simply compute

the running time of its serialization, which we obtain by replacing the parallel for

loops with ordinary for loops. Thus, we have T 1 .n/ D ‚.n 2 /, because the qua-

dratic running time of the doubly nested loops in lines 5–7 dominates. This analysis

27.1 The basics of dynamic multithreading 787

seems to ignore the overhead for recursive spawning in implementing the parallel

loops, however. In fact, the overhead of recursive spawning does increase the work

of a parallel loop compared with that of its serialization, but not asymptotically.

To see why, observe that since the tree of recursive procedure instances is a full

binary tree, the number of internal nodes is 1 fewer than the number of leaves (see

Exercise B.5-3). Each internal node performs constant work to divide the iteration

range, and each leaf corresponds to an iteration of the loop, which takes at least

constant time (‚.n/ time in this case). Thus, we can amortize the overhead of re-

cursive spawning against the work of the iterations, contributing at most a constant

factor to the overall work.

As a practical matter, dynamic-multithreading concurrency platforms sometimes

coarsen the leaves of the recursion by executing several iterations in a single leaf,

either automatically or under programmer control, thereby reducing the overhead

of recursive spawning. This reduced overhead comes at the expense of also reduc-

ing the parallelism, however, but if the computation has sufﬁcient parallel slack-

ness, near-perfect linear speedup need not be sacriﬁced.

We must also account for the overhead of recursive spawning when analyzing the

span of a parallel-loop construct. Since the depth of recursive calling is logarithmic

in the number of iterations, for a parallel loop with n iterations in which the ith

iteration has span iter 1 .i/, the span is

T 1 .n/ D ‚.lg n/ C max

1in

iter 1 .i/ :

For example, for MAT-VEC on an n n matrix, the parallel initialization loop in

lines 3–4 has span ‚.lg n/, because the recursive spawning dominates the constant-

time work of each iteration. The span of the doubly nested loops in lines 5–7

is ‚.n/, because each iteration of the outer parallel for loop contains n iterations

of the inner (serial) for loop. The span of the remaining code in the procedure

is constant, and thus the span is dominated by the doubly nested loops, yielding

an overall span of ‚.n/ for the whole procedure. Since the work is ‚.n 2 /, the

parallelism is ‚.n 2 /=‚.n/ D ‚.n/. (Exercise 27.1-6 asks you to provide an

implementation with even more parallelism.)

Race conditions

A multithreaded algorithm is deterministic if it always does the same thing on the

same input, no matter how the instructions are scheduled on the multicore com-

puter. It is nondeterministic if its behavior might vary from run to run. Often, a

multithreaded algorithm that is intended to be deterministic fails to be, because it

contains a “determinacy race.”

Race conditions are the bane of concurrency. Famous race bugs include the

Therac-25 radiation therapy machine, which killed three people and injured sev-

788 Chapter 27 Multithreaded Algorithms

eral others, and the North American Blackout of 2003, which left over 50 million

people without power. These pernicious bugs are notoriously hard to ﬁnd. You can

run tests in the lab for days without a failure only to discover that your software

sporadically crashes in the ﬁeld.

A determinacy race occurs when two logically parallel instructions access the

same memory location and at least one of the instructions performs a write. The

following procedure illustrates a race condition:

RACE-EXAMPLE. /

1 x D 0

2 parallel for i D 1 to 2

3 x D x C 1

4 print x

After initializing x to 0 in line 1, RACE-EXAMPLE creates two parallel strands,

each of which increments x in line 3. Although it might seem that RACE-

EXAMPLE should always print the value 2 (its serialization certainly does), it could

instead print the value 1. Let’s see how this anomaly might occur.

When a processor increments x, the operation is not indivisible, but is composed

of a sequence of instructions:

1. Read x from memory into one of the processor’s registers.

2. Increment the value in the register.

3. Write the value in the register back into x in memory.

Figure 27.5(a) illustrates a computation dag representing the execution of RACE-

EXAMPLE, with the strands broken down to individual instructions. Recall that

since an ideal parallel computer supports sequential consistency, we can view the

parallel execution of a multithreaded algorithm as an interleaving of instructions

that respects the dependencies in the dag. Part (b) of the ﬁgure shows the values

in an execution of the computation that elicits the anomaly. The value x is stored

in memory, and r 1 and r 2 are processor registers. In step 1, one of the processors

sets x to 0. In steps 2 and 3, processor 1 reads x from memory into its register r 1

and increments it, producing the value 1 in r 1 . At that point, processor 2 comes

into the picture, executing instructions 4–6. Processor 2 reads x from memory into

register r 2 ; increments it, producing the value 1 in r 2 ; and then stores this value

into x, setting x to 1. Now, processor 1 resumes with step 7, storing the value 1

in r 1 into x, which leaves the value of x unchanged. Therefore, step 8 prints the

value 1, rather than 2, as the serialization would print.

We can see what has happened. If the effect of the parallel execution were that

processor 1 executed all its instructions before processor 2, the value 2 would be

27.1 The basics of dynamic multithreading 789

incr r

1

3

r

1

= x 2

x = r

1

7

incr r

2

5

r

2

= x 4

x = r

2

6

x = 0 1

print x 8

(a)

step x r

1

r

2

1

2

3

4

5

6

7

0

0

0

0

0

1

1

–

0

1

1

1

1

1

–

–

–

0

1

1

1

(b)

Figure 27.5 Illustration of the determinacy race in RACE-EXAMPLE. (a) A computation dag show-

ing the dependencies among individual instructions. The processor registers are r1 and r2. Instruc-

tions unrelated to the race, such as the implementation of loop control, are omitted. (b) An execution

sequence that elicits the bug, showing the values of x in memory and registers r1 and r2 for each

step in the execution sequence.

printed. Conversely, if the effect were that processor 2 executed all its instructions

before processor 1, the value 2 would still be printed. When the instructions of the

two processors execute at the same time, however, it is possible, as in this example

execution, that one of the updates to x is lost.

Of course, many executions do not elicit the bug. For example, if the execution

order were h1; 2; 3; 7; 4; 5; 6; 8i or h1; 4; 5; 6; 2; 3; 7; 8i, we would get the cor-

rect result. That’s the problem with determinacy races. Generally, most orderings

produce correct results—such as any in which the instructions on the left execute

before the instructions on the right, or vice versa. But some orderings generate

improper results when the instructions interleave. Consequently, races can be ex-

tremely hard to test for. You can run tests for days and never see the bug, only to

experience a catastrophic system crash in the ﬁeld when the outcome is critical.

Although we can cope with races in a variety of ways, including using mutual-

exclusion locks and other methods of synchronization, for our purposes, we shall

simply ensure that strands that operate in parallel are independent: they have no

determinacy races among them. Thus, in a parallel for construct, all the iterations

should be independent. Between a spawn and the corresponding sync, the code

of the spawned child should be independent of the code of the parent, including

code executed by additional spawned or called children. Note that arguments to a

spawned child are evaluated in the parent before the actual spawn occurs, and thus

the evaluation of arguments to a spawned subroutine is in series with any accesses

to those arguments after the spawn.

790 Chapter 27 Multithreaded Algorithms

As an example of how easy it is to generate code with races, here is a faulty

implementation of multithreaded matrix-vector multiplication that achieves a span

of ‚.lg n/ by parallelizing the inner for loop:

MAT-VEC-WRONG.A; x/

1 n D A:rows

2 let y be a new vector of length n

3 parallel for i D 1 to n

4 y i D 0

5 parallel for i D 1 to n

6 parallel for j D 1 to n

7 y i D y i C a ij x j

8 return y

This procedure is, unfortunately, incorrect due to races on updating y i in line 7,

which executes concurrently for all n values of j . Exercise 27.1-6 asks you to give

a correct implementation with ‚.lg n/ span.

A multithreaded algorithm with races can sometimes be correct. As an exam-

ple, two parallel threads might store the same value into a shared variable, and it

wouldn’t matter which stored the value ﬁrst. Generally, however, we shall consider

code with races to be illegal.

A chess lesson

We close this section with a true story that occurred during the development of

the world-class multithreaded chess-playing program ?Socrates [80], although the

timings below have been simpliﬁed for exposition. The program was prototyped

on a 32-processor computer but was ultimately to run on a supercomputer with 512

processors. At one point, the developers incorporated an optimization into the pro-

gram that reduced its running time on an important benchmark on the 32-processor

machine from T 32 D 65 seconds to T 0

32

D 40 seconds. Yet, the developers used

the work and span performance measures to conclude that the optimized version,

which was faster on 32 processors, would actually be slower than the original ver-

sion on 512 processsors. As a result, they abandoned the “optimization.”

Here is their analysis. The original version of the program had work T 1 D 2048

seconds and span T 1 D 1 second. If we treat inequality (27.4) as an equation,

T P D T 1 =P C T 1 , and use it as an approximation to the running time on P pro-

cessors, we see that indeed T 32 D 2048=32 C 1 D 65. With the optimization, the

work became T 0

1

D 1024 seconds and the span became T 0

1

D 8 seconds. Again

using our approximation, we get T 0

32

D 1024=32 C 8 D 40.

The relative speeds of the two versions switch when we calculate the running

times on 512 processors, however. In particular, we have T 512 D 2048=512C1 D 5

27.1 The basics of dynamic multithreading 791

seconds, and T 0

512

D 1024=512 C 8 D 10 seconds. The optimization that sped up

the program on 32 processors would have made the program twice as slow on 512

processors! The optimized version’s span of 8, which was not the dominant term in

the running time on 32 processors, became the dominant term on 512 processors,

nullifying the advantage from using more processors.

The moral of the story is that work and span can provide a better means of

extrapolating performance than can measured running times.

Exercises

27.1-1

Suppose that we spawn P-FIB.n 2/ in line 4 of P-FIB, rather than calling it

as is done in the code. What is the impact on the asymptotic work, span, and

parallelism?

27.1-2

Draw the computation dag that results from executing P-FIB.5/. Assuming that

each strand in the computation takes unit time, what are the work, span, and par-

allelism of the computation? Show how to schedule the dag on 3 processors using

greedy scheduling by labeling each strand with the time step in which it is executed.

27.1-3

Prove that a greedy scheduler achieves the following time bound, which is slightly

stronger than the bound proven in Theorem 27.1:

T P

T 1 T 1

P

C T 1 : (27.5)

27.1-4

Construct a computation dag for which one execution of a greedy scheduler can

take nearly twice the time of another execution of a greedy scheduler on the same

number of processors. Describe how the two executions would proceed.

27.1-5

Professor Karan measures her deterministic multithreaded algorithm on 4, 10,

and 64 processors of an ideal parallel computer using a greedy scheduler. She

claims that the three runs yielded T 4 D 80 seconds, T 10 D 42 seconds, and

T 64 D 10 seconds. Argue that the professor is either lying or incompetent. (Hint:

Use the work law (27.2), the span law (27.3), and inequality (27.5) from Exer-

cise 27.1-3.)

792 Chapter 27 Multithreaded Algorithms

27.1-6

Give a multithreaded algorithm to multiply an n n matrix by an n-vector that

achieves ‚.n 2 = lg n/ parallelism while maintaining ‚.n 2 / work.

27.1-7

Consider the following multithreaded pseudocode for transposing an n n matrix A

in place:

P-TRANSPOSE.A/

1 n D A:rows

2 parallel for j D 2 to n

3 parallel for i D 1 to j 1

4 exchange a ij with a ji

Analyze the work, span, and parallelism of this algorithm.

27.1-8

Suppose that we replace the parallel for loop in line 3 of P-TRANSPOSE (see Ex-

ercise 27.1-7) with an ordinary for loop. Analyze the work, span, and parallelism

of the resulting algorithm.

27.1-9

For how many processors do the two versions of the chess programs run equally

fast, assuming that T P D T 1 =P C T 1 ?

27.2 Multithreaded matrix multiplication

In this section, we examine how to multithread matrix multiplication, a problem

whose serial running time we studied in Section 4.2. We’ll look at multithreaded

algorithms based on the standard triply nested loop, as well as divide-and-conquer

algorithms.

Multithreaded matrix multiplication

The ﬁrst algorithm we study is the straighforward algorithm based on parallelizing

the loops in the procedure SQUARE-MATRIX-MULTIPLY on page 75:

27.2 Multithreaded matrix multiplication 793

P-SQUARE-MATRIX-MULTIPLY.A; B/

1 n D A:rows

2 let C be a new n n matrix

3 parallel for i D 1 to n

4 parallel for j D 1 to n

5 c ij D 0

6 for k D 1 to n

7 c ij D c ij C a ik b kj

8 return C

To analyze this algorithm, observe that since the serialization of the algorithm is

just SQUARE-MATRIX-MULTIPLY, the work is therefore simply T 1 .n/ D ‚.n 3 /,

the same as the running time of SQUARE-MATRIX-MULTIPLY. The span is

T 1 .n/ D ‚.n/, because it follows a path down the tree of recursion for the

parallel for loop starting in line 3, then down the tree of recursion for the parallel

for loop starting in line 4, and then executes all n iterations of the ordinary for loop

starting in line 6, resulting in a total span of ‚.lg n/ C ‚.lg n/ C ‚.n/ D ‚.n/.

Thus, the parallelism is ‚.n 3 /=‚.n/ D ‚.n 2 /. Exercise 27.2-3 asks you to par-

allelize the inner loop to obtain a parallelism of ‚.n 3 = lg n/, which you cannot do

straightforwardly using parallel for, because you would create races.

A divide-and-conquer multithreaded algorithm for matrix multiplication

As we learned in Section 4.2, we can multiply n n matrices serially in time

‚.n lg 7 / D O.n 2:81 / using Strassen’s divide-and-conquer strategy, which motivates

us to look at multithreading such an algorithm. We begin, as we did in Section 4.2,

with multithreading a simpler divide-and-conquer algorithm.

Recall from page 77 that the SQUARE-MATRIX-MULTIPLY-RECURSIVE proce-

dure, which multiplies two n n matrices A and B to produce the n n matrix C ,

relies on partitioning each of the three matrices into four n=2 n=2 submatrices:

A D

A 11 A 12

A 21 A 22

; B D

B 11 B 12

B 21 B 22

; C D

C 11 C 12

C 21 C 22

:

Then, we can write the matrix product as

C 11 C 12

C 21 C 22

D

A 11 A 12

A 21 A 22

B 11 B 12

B 21 B 22

D

A 11 B 11 A 11 B 12

A 21 B 11 A 21 B 12

C

A 12 B 21 A 12 B 22

A 22 B 21 A 22 B 22

: (27.6)

Thus, to multiply two nn matrices, we perform eight multiplications of n=2n=2

matrices and one addition of nn matrices. The following pseudocode implements

794 Chapter 27 Multithreaded Algorithms

this divide-and-conquer strategy using nested parallelism. Unlike the SQUARE-

MATRIX-MULTIPLY-RECURSIVE procedure on which it is based, P-MATRIX-

MULTIPLY-RECURSIVE takes the output matrix as a parameter to avoid allocating

matrices unnecessarily.

P-MATRIX-MULTIPLY-RECURSIVE.C; A; B/

1 n D A:rows

2 if n == 1

3 c 11 D a 11 b 11

4 else let T be a new n n matrix

5 partition A, B, C , and T into n=2 n=2 submatrices

A 11 ; A 12 ; A 21 ; A 22 ; B 11 ; B 12 ; B 21 ; B 22 ; C 11 ; C 12 ; C 21 ; C 22 ;

and T 11 ; T 12 ; T 21 ; T 22 ; respectively

6 spawn P-MATRIX-MULTIPLY-RECURSIVE.C 11 ; A 11 ; B 11 /

7 spawn P-MATRIX-MULTIPLY-RECURSIVE.C 12 ; A 11 ; B 12 /

8 spawn P-MATRIX-MULTIPLY-RECURSIVE.C 21 ; A 21 ; B 11 /

9 spawn P-MATRIX-MULTIPLY-RECURSIVE.C 22 ; A 21 ; B 12 /

10 spawn P-MATRIX-MULTIPLY-RECURSIVE.T 11 ; A 12 ; B 21 /

11 spawn P-MATRIX-MULTIPLY-RECURSIVE.T 12 ; A 12 ; B 22 /

12 spawn P-MATRIX-MULTIPLY-RECURSIVE.T 21 ; A 22 ; B 21 /

13 P-MATRIX-MULTIPLY-RECURSIVE.T 22 ; A 22 ; B 22 /

14 sync

15 parallel for i D 1 to n

16 parallel for j D 1 to n

17 c ij D c ij C t ij

Line 3 handles the base case, where we are multiplying 1 1 matrices. We handle

the recursive case in lines 4–17. We allocate a temporary matrix T in line 4, and

line 5 partitions each of the matrices A, B, C , and T into n=2 n=2 submatrices.

(As with SQUARE-MATRIX-MULTIPLY-RECURSIVE on page 77, we gloss over

the minor issue of how to use index calculations to represent submatrix sections

of a matrix.) The recursive call in line 6 sets the submatrix C 11 to the submatrix

product A 11 B 11 , so that C 11 equals the ﬁrst of the two terms that form its sum in

equation (27.6). Similarly, lines 7–9 set C 12 , C 21 , and C 22 to the ﬁrst of the two

terms that equal their sums in equation (27.6). Line 10 sets the submatrix T 11 to

the submatrix product A 12 B 21 , so that T 11 equals the second of the two terms that

form C 11 ’s sum. Lines 11–13 set T 12 , T 21 , and T 22 to the second of the two terms

that form the sums of C 12 , C 21 , and C 22 , respectively. The ﬁrst seven recursive

calls are spawned, and the last one runs in the main strand. The sync statement in

line 14 ensures that all the submatrix products in lines 6–13 have been computed,

27.2 Multithreaded matrix multiplication 795

after which we add the products from T into C in using the doubly nested parallel

for loops in lines 15–17.

We ﬁrst analyze the work M 1 .n/ of the P-MATRIX-MULTIPLY-RECURSIVE

procedure, echoing the serial running-time analysis of its progenitor SQUARE-

MATRIX-MULTIPLY-RECURSIVE. In the recursive case, we partition in ‚.1/ time,

perform eight recursive multiplications of n=2 n=2 matrices, and ﬁnish up with

the ‚.n 2 / work from adding two n n matrices. Thus, the recurrence for the

work M 1 .n/ is

M 1 .n/ D 8M 1 .n=2/ C ‚.n

2

/

D ‚.n

3

/

by case 1 of the master theorem. In other words, the work of our multithreaded al-

gorithm is asymptotically the same as the running time of the procedure SQUARE-

MATRIX-MULTIPLY in Section 4.2, with its triply nested loops.

To determine the span M 1 .n/ of P-MATRIX-MULTIPLY-RECURSIVE, we ﬁrst

observe that the span for partitioning is ‚.1/, which is dominated by the ‚.lg n/

span of the doubly nested parallel for loops in lines 15–17. Because the eight

parallel recursive calls all execute on matrices of the same size, the maximum span

for any recursive call is just the span of any one. Hence, the recurrence for the

span M 1 .n/ of P-MATRIX-MULTIPLY-RECURSIVE is

M 1 .n/ D M 1 .n=2/ C ‚.lg n/ : (27.7)

This recurrence does not fall under any of the cases of the master theorem, but

it does meet the condition of Exercise 4.6-2. By Exercise 4.6-2, therefore, the

solution to recurrence (27.7) is M 1 .n/ D ‚.lg

2

n/.

Now that we know the work and span of P-MATRIX-MULTIPLY-RECURSIVE,

we can compute its parallelism as M 1 .n/=M 1 .n/ D ‚.n 3 = lg

2

n/, which is very

high.

Multithreading Strassen’s method

To multithread Strassen’s algorithm, we follow the same general outline as on

page 79, only using nested parallelism:

1. Divide the input matrices A and B and output matrix C into n=2 n=2 sub-

matrices, as in equation (27.6). This step takes ‚.1/ work and span by index

calculation.

2. Create 10 matrices S 1 ; S 2 ; : : : ; S 10 , each of which is n=2 n=2 and is the sum

or difference of two matrices created in step 1. We can create all 10 matrices

with ‚.n 2 / work and ‚.lg n/ span by using doubly nested parallel for loops.

796 Chapter 27 Multithreaded Algorithms

3. Using the submatrices created in step 1 and the 10 matrices created in

step 2, recursively spawn the computation of seven n=2 n=2 matrix products

P 1 ; P 2 ; : : : ; P 7 .

4. Compute the desired submatrices C 11 ; C 12 ; C 21 ; C 22 of the result matrix C by

adding and subtracting various combinations of the P i matrices, once again

using doubly nested parallel for loops. We can compute all four submatrices

with ‚.n 2 / work and ‚.lg n/ span.

To analyze this algorithm, we ﬁrst observe that since the serialization is the

same as the original serial algorithm, the work is just the running time of the

serialization, namely, ‚.n lg 7 /. As for P-MATRIX-MULTIPLY-RECURSIVE, we

can devise a recurrence for the span. In this case, seven recursive calls exe-

cute in parallel, but since they all operate on matrices of the same size, we ob-

tain the same recurrence (27.7) as we did for P-MATRIX-MULTIPLY-RECURSIVE,

which has solution ‚.lg

2

n/. Thus, the parallelism of multithreaded Strassen’s

method is ‚.n lg 7 = lg

2

n/, which is high, though slightly less than the parallelism

of P-MATRIX-MULTIPLY-RECURSIVE.

Exercises

27.2-1

Draw the computation dag for computing P-SQUARE-MATRIX-MULTIPLY on 2 2

matrices, labeling how the vertices in your diagram correspond to strands in the

execution of the algorithm. Use the convention that spawn and call edges point

downward, continuation edges point horizontally to the right, and return edges

point upward. Assuming that each strand takes unit time, analyze the work, span,

and parallelism of this computation.

27.2-2

Repeat Exercise 27.2-1 for P-MATRIX-MULTIPLY-RECURSIVE.

27.2-3

Give pseudocode for a multithreaded algorithm that multiplies two n n matrices

with work ‚.n 3 / but span only ‚.lg n/. Analyze your algorithm.

27.2-4

Give pseudocode for an efﬁcient multithreaded algorithm that multiplies a p q

matrix by a q r matrix. Your algorithm should be highly parallel even if any of

p, q, and r are 1. Analyze your algorithm.

27.3 Multithreaded merge sort 797

27.2-5

Give pseudocode for an efﬁcient multithreaded algorithm that transposes an n n

matrix in place by using divide-and-conquer to divide the matrix recursively into

four n=2 n=2 submatrices. Analyze your algorithm.

27.2-6

Give pseudocode for an efﬁcient multithreaded implementation of the Floyd-

Warshall algorithm (see Section 25.2), which computes shortest paths between all

pairs of vertices in an edge-weighted graph. Analyze your algorithm.

27.3 Multithreaded merge sort

We ﬁrst saw serial merge sort in Section 2.3.1, and in Section 2.3.2 we analyzed its

running time and showed it to be ‚.n lg n/. Because merge sort already uses the

divide-and-conquer paradigm, it seems like a terriﬁc candidate for multithreading

using nested parallelism. We can easily modify the pseudocode so that the ﬁrst

recursive call is spawned:

MERGE-SORT

0 .A; p; r/

1 if p < r

2 q D b.p C r/=2c

3 spawn MERGE-SORT

0

.A; p; q/

4 MERGE-SORT

0

.A; q C 1; r/

5 sync

6 MERGE.A; p; q; r/

Like its serial counterpart, MERGE-SORT

0

sorts the subarray AŒp : : r. After the

two recursive subroutines in lines 3 and 4 have completed, which is ensured by the

sync statement in line 5, MERGE-SORT

0

calls the same MERGE procedure as on

page 31.

Let us analyze MERGE-SORT

0

. To do so, we ﬁrst need to analyze MERGE. Re-

call that its serial running time to merge n elements is ‚.n/. Because MERGE is

serial, both its work and its span are ‚.n/. Thus, the following recurrence charac-

terizes the work MS

0

1

.n/ of MERGE-SORT

0

on n elements:

MS

0

1

.n/ D 2 MS

0

1

.n=2/ C ‚.n/

D ‚.n lg n/ ;

798 Chapter 27 Multithreaded Algorithms

… … …

… …

merge merge copy

p 1 q 1 r 1 p 2 q 2 r 2

p 3 q 3 r 3

A

T

x

x

x

x < x

x

x x

Figure 27.6 The idea behind the multithreaded merging of two sorted subarrays T Œp1 : : r1

and T Œp2 : : r2 into the subarray AŒp3 : : r3. Letting x D T Œq1 be the median of T Œp1 : : r1 and q2

be the place in T Œp2 : : r2 such that x would fall between T Œq2 1 and T Œq2, every element in

subarrays T Œp1 : : q1 1 and T Œp2 : : q2 1 (lightly shaded) is less than or equal to x, and every

element in the subarrays T Œq1 C 1 : : r1 and T Œq2 C 1 : : r2 (heavily shaded) is at least x. To merge,

we compute the index q3 where x belongs in AŒp3 : : r3, copy x into AŒq3, and then recursively

merge T Œp1 : : q1 1 with T Œp2 : : q2 1 into AŒp3 : : q3 1 and T Œq1 C 1 : : r1 with T Œq2 : : r2

into AŒq3 C 1 : : r3.

which is the same as the serial running time of merge sort. Since the two recursive

calls of MERGE-SORT

0

can run in parallel, the span MS

0

1

is given by the recurrence

MS

0

1

.n/ D MS

0

1

.n=2/ C ‚.n/

D ‚.n/ :

Thus, the parallelism of MERGE-SORT

0

comes to MS

0

1

.n/=MS

0

1

.n/ D ‚.lg n/,

which is an unimpressive amount of parallelism. To sort 10 million elements, for

example, it might achieve linear speedup on a few processors, but it would not

scale up effectively to hundreds of processors.

You probably have already ﬁgured out where the parallelism bottleneck is in

this multithreaded merge sort: the serial MERGE procedure. Although merging

might initially seem to be inherently serial, we can, in fact, fashion a multithreaded

version of it by using nested parallelism.

Our divide-and-conquer strategy for multithreaded merging, which is illus-

trated in Figure 27.6, operates on subarrays of an array T . Suppose that we

are merging the two sorted subarrays T Œp 1 : : r 1 of length n 1 D r 1 p 1 C 1

and T Œp 2 : : r 2 of length n 2 D r 2 p 2 C 1 into another subarray AŒp 3 : : r 3 , of

length n 3 D r 3 p 3 C 1 D n 1 C n 2 . Without loss of generality, we make the sim-

plifying assumption that n 1 n 2 .

We ﬁrst ﬁnd the middle element x D T Œq 1 of the subarray T Œp 1 : : r 1 ,

where q 1 D b.p 1 C r 1 /=2c. Because the subarray is sorted, x is a median

of T Œp 1 : : r 1 : every element in T Œp 1 : : q 1 1 is no more than x, and every el-

ement in T Œq 1 C 1 : : r 1 is no less than x. We then use binary search to ﬁnd the

27.3 Multithreaded merge sort 799

index q 2 in the subarray T Œp 2 : : r 2 so that the subarray would still be sorted if we

inserted x between T Œq 2 1 and T Œq 2 .

We next merge the original subarrays T Œp 1 : : r 1 and T Œp 2 : : r 2 into AŒp 3 : : r 3

as follows:

1. Set q 3 D p 3 C .q 1 p 1 / C .q 2 p 2 /.

2. Copy x into AŒq 3 .

3. Recursively merge T Œp 1 : : q 1 1 with T Œp 2 : : q 2 1, and place the result into

the subarray AŒp 3 : : q 3 1.

4. Recursively merge T Œq 1 C 1 : : r 1 with T Œq 2 : : r 2 , and place the result into the

subarray AŒq 3 C 1 : : r 3 .

When we compute q 3 , the quantity q 1 p 1 is the number of elements in the subarray

T Œp 1 : : q 1 1, and the quantity q 2 p 2 is the number of elements in the subarray

T Œp 2 : : q 2 1. Thus, their sum is the number of elements that end up before x in

the subarray AŒp 3 : : r 3 .

The base case occurs when n 1 D n 2 D 0, in which case we have no work

to do to merge the two empty subarrays. Since we have assumed that the sub-

array T Œp 1 : : r 1 is at least as long as T Œp 2 : : r 2 , that is, n 1 n 2 , we can check

for the base case by just checking whether n 1 D 0. We must also ensure that the

recursion properly handles the case when only one of the two subarrays is empty,

which, by our assumption that n 1 n 2 , must be the subarray T Œp 2 : : r 2 .

Now, let’s put these ideas into pseudocode. We start with the binary search,

which we express serially. The procedure BINARY-SEARCH.x; T; p; r/ takes a

key x and a subarray T Œp : : r, and it returns one of the following:

If T Œp : : r is empty (r < p), then it returns the index p.

If x T Œp, and hence less than or equal to all the elements of T Œp : : r, then

it returns the index p.

If x > T Œp, then it returns the largest index q in the range p < q r C 1 such

that T Œq 1 < x.

Here is the pseudocode:

BINARY-SEARCH.x; T; p; r/

1 low D p

2 high D max.p; r C 1/

3 while low < high

4 mid D b.low C high/=2c

5 if x T Œmid

6 high D mid

7 else low D mid C 1

8 return high

800 Chapter 27 Multithreaded Algorithms

The call BINARY-SEARCH.x; T; p; r/ takes ‚.lg n/ serial time in the worst case,

where n D r p C 1 is the size of the subarray on which it runs. (See Exer-

cise 2.3-5.) Since BINARY-SEARCH is a serial procedure, its worst-case work and

span are both ‚.lg n/.

We are now prepared to write pseudocode for the multithreaded merging pro-

cedure itself. Like the MERGE procedure on page 31, the P-MERGE procedure

assumes that the two subarrays to be merged lie within the same array. Un-

like MERGE, however, P-MERGE does not assume that the two subarrays to

be merged are adjacent within the array. (That is, P-MERGE does not require

that p 2 D r 1 C 1.) Another difference between MERGE and P-MERGE is that

P-MERGE takes as an argument an output subarray A into which the merged val-

ues should be stored. The call P-MERGE.T; p 1 ; r 1 ; p 2 ; r 2 ; A; p 3 / merges the sorted

subarrays T Œp 1 : : r 1 and T Œp 2 : : r 2 into the subarray AŒp 3 : : r 3 , where r 3 D

p 3 C .r 1 p 1 C 1/ C .r 2 p 2 C 1/ 1 D p 3 C .r 1 p 1 / C .r 2 p 2 / C 1 and

is not provided as an input.

P-MERGE.T; p 1 ; r 1 ; p 2 ; r 2 ; A; p 3 /

1 n 1 D r 1 p 1 C 1

2 n 2 D r 2 p 2 C 1

3 if n 1 < n 2 // ensure that n 1 n 2

4 exchange p 1 with p 2

5 exchange r 1 with r 2

6 exchange n 1 with n 2

7 if n 1 == 0 // both empty?

8 return

9 else q 1 D b.p 1 C r 1 /=2c

10 q 2 D BINARY-SEARCH.T Œq 1 ; T; p 2 ; r 2 /

11 q 3 D p 3 C .q 1 p 1 / C .q 2 p 2 /

12 AŒq 3 D T Œq 1

13 spawn P-MERGE.T; p 1 ; q 1 1; p 2 ; q 2 1; A; p 3 /

14 P-MERGE.T; q 1 C 1; r 1 ; q 2 ; r 2 ; A; q 3 C 1/

15 sync

The P-MERGE procedure works as follows. Lines 1–2 compute the lengths n 1

and n 2 of the subarrays T Œp 1 : : r 1 and T Œp 2 : : r 2 , respectively. Lines 3–6 en-

force the assumption that n 1 n 2 . Line 7 tests for the base case, where the

subarray T Œp 1 : : r 1 is empty (and hence so is T Œp 2 : : r 2 ), in which case we sim-

ply return. Lines 9–15 implement the divide-and-conquer strategy. Line 9 com-

putes the midpoint of T Œp 1 : : r 1 , and line 10 ﬁnds the point q 2 in T Œp 2 : : r 2 such

that all elements in T Œp 2 : : q 2 1 are less than T Œq 1 (which corresponds to x)

and all the elements in T Œq 2 : : p 2 are at least as large as T Œq 1 . Line 11 com-

27.3 Multithreaded merge sort 801

putes the index q 3 of the element that divides the output subarray AŒp 3 : : r 3 into

AŒp 3 : : q 3 1 and AŒq 3 C1 : : r 3 , and then line 12 copies T Œq 1 directly into AŒq 3 .

Then, we recurse using nested parallelism. Line 13 spawns the ﬁrst subproblem,

while line 14 calls the second subproblem in parallel. The sync statement in line 15

ensures that the subproblems have completed before the procedure returns. (Since

every procedure implicitly executes a sync before returning, we could have omitted

the sync statement in line 15, but including it is good coding practice.) There

is some cleverness in the coding to ensure that when the subarray T Œp 2 : : r 2 is

empty, the code operates correctly. The way it works is that on each recursive call,

a median element of T Œp 1 : : r 1 is placed into the output subarray, until T Œp 1 : : r 1

itself ﬁnally becomes empty, triggering the base case.

Analysis of multithreaded merging

We ﬁrst derive a recurrence for the span PM 1 .n/ of P-MERGE, where the two

subarrays contain a total of n D n 1 Cn 2 elements. Because the spawn in line 13 and

the call in line 14 operate logically in parallel, we need examine only the costlier of

the two calls. The key is to understand that in the worst case, the maximum number

of elements in either of the recursive calls can be at most 3n=4, which we see as

follows. Because lines 3–6 ensure that n 2 n 1 , it follows that n 2 D 2n 2 =2

.n 1 C n 2 /=2 D n=2. In the worst case, one of the two recursive calls merges

bn 1 =2c elements of T Œp 1 : : r 1 with all n 2 elements of T Œp 2 : : r 2 , and hence the

number of elements involved in the call is

bn 1 =2c C n 2 n 1 =2 C n 2 =2 C n 2 =2

D .n 1 C n 2 /=2 C n 2 =2

n=2 C n=4

D 3n=4 :

Adding in the ‚.lg n/ cost of the call to BINARY-SEARCH in line 10, we obtain

the following recurrence for the worst-case span:

PM 1 .n/ D PM 1 .3n=4/ C ‚.lg n/ : (27.8)

(For the base case, the span is ‚.1/, since lines 1–8 execute in constant time.)

This recurrence does not fall under any of the cases of the master theorem, but it

meets the condition of Exercise 4.6-2. Therefore, the solution to recurrence (27.8)

is PM 1 .n/ D ‚.lg

2

n/.

We now analyze the work PM 1 .n/ of P-MERGE on n elements, which turns out

to be ‚.n/. Since each of the n elements must be copied from array T to array A,

we have PM 1 .n/ D .n/. Thus, it remains only to show that PM 1 .n/ D O.n/.

We shall ﬁrst derive a recurrence for the worst-case work. The binary search in

line 10 costs ‚.lg n/ in the worst case, which dominates the other work outside

802 Chapter 27 Multithreaded Algorithms

of the recursive calls. For the recursive calls, observe that although the recursive

calls in lines 13 and 14 might merge different numbers of elements, together the

two recursive calls merge at most n elements (actually n 1 elements, since T Œq 1

does not participate in either recursive call). Moreover, as we saw in analyzing the

span, a recursive call operates on at most 3n=4 elements. We therefore obtain the

recurrence

PM 1 .n/ D PM 1 .˛n/ C PM 1 ..1 ˛/n/ C O.lg n/ ; (27.9)

where ˛ lies in the range 1=4 ˛ 3=4, and where we understand that the actual

value of ˛ may vary for each level of recursion.

We prove that recurrence (27.9) has solution PM 1 D O.n/ via the substitution

method. Assume that PM 1 .n/ c 1 nc 2 lg n for some positive constants c 1 and c 2 .

Substituting gives us

PM 1 .n/ .c 1 ˛n c 2 lg.˛n// C .c 1 .1 ˛/n c 2 lg..1 ˛/n// C ‚.lg n/

D c 1 .˛ C .1 ˛//n c 2 .lg.˛n/ C lg..1 ˛/n// C ‚.lg n/

D c 1 n c 2 .lg ˛ C lg n C lg.1 ˛/ C lg n/ C ‚.lg n/

D c 1 n c 2 lg n .c 2 .lg n C lg.˛.1 ˛/// ‚.lg n//

c 1 n c 2 lg n ;

since we can choose c 2 large enough that c 2 .lg n C lg.˛.1 ˛/// dominates the

‚.lg n/ term. Furthermore, we can choose c 1 large enough to satisfy the base

conditions of the recurrence. Since the work PM 1 .n/ of P-MERGE is both .n/

and O.n/, we have PM 1 .n/ D ‚.n/.

The parallelism of P-MERGE is PM 1 .n/=PM 1 .n/ D ‚.n= lg

2

n/.

Multithreaded merge sort

Now that we have a nicely parallelized multithreaded merging procedure, we can

incorporate it into a multithreaded merge sort. This version of merge sort is similar

to the MERGE-SORT

0

procedure we saw earlier, but unlike MERGE-SORT

0

, it takes

as an argument an output subarray B, which will hold the sorted result. In par-

ticular, the call P-MERGE-SORT.A; p; r; B; s/ sorts the elements in AŒp : : r and

stores them in BŒs : : s C r p.

27.3 Multithreaded merge sort 803

P-MERGE-SORT.A; p; r; B; s/

1 n D r p C 1

2 if n == 1

3 BŒs D AŒp

4 else let T Œ1 : : n be a new array

5 q D b.p C r/=2c

6 q 0 D q p C 1

7 spawn P-MERGE-SORT.A; p; q; T; 1/

8 P-MERGE-SORT.A; q C 1; r; T; q 0 C 1/

9 sync

10 P-MERGE.T; 1; q 0 ; q 0 C 1; n; B; s/

After line 1 computes the number n of elements in the input subarray AŒp : : r,

lines 2–3 handle the base case when the array has only 1 element. Lines 4–6 set

up for the recursive spawn in line 7 and call in line 8, which operate in parallel. In

particular, line 4 allocates a temporary array T with n elements to store the results

of the recursive merge sorting. Line 5 calculates the index q of AŒp : : r to divide

the elements into the two subarrays AŒp : : q and AŒq C 1 : : r that will be sorted

recursively, and line 6 goes on to compute the number q 0

of elements in the ﬁrst

subarray AŒp : : q, which line 8 uses to determine the starting index in T of where

to store the sorted result of AŒq C 1 : : r. At that point, the spawn and recursive

call are made, followed by the sync in line 9, which forces the procedure to wait

until the spawned procedure is done. Finally, line 10 calls P-MERGE to merge

the sorted subarrays, now in T Œ1 : : q 0 and T Œq 0 C 1 : : n, into the output subarray

BŒs : : s C r p.

Analysis of multithreaded merge sort

We start by analyzing the work PMS 1 .n/ of P-MERGE-SORT, which is consider-

ably easier than analyzing the work of P-MERGE. Indeed, the work is given by the

recurrence

PMS 1 .n/ D 2 PMS 1 .n=2/ C PM 1 .n/

D 2 PMS 1 .n=2/ C ‚.n/ :

This recurrence is the same as the recurrence (4.4) for ordinary MERGE-SORT

from Section 2.3.1 and has solution PMS 1 .n/ D ‚.n lg n/ by case 2 of the master

theorem.

We now derive and analyze a recurrence for the worst-case span PMS 1 .n/. Be-

cause the two recursive calls to P-MERGE-SORT on lines 7 and 8 operate logically

in parallel, we can ignore one of them, obtaining the recurrence

804 Chapter 27 Multithreaded Algorithms

PMS 1 .n/ D PMS 1 .n=2/ C PM 1 .n/

D PMS 1 .n=2/ C ‚.lg

2

n/ : (27.10)

As for recurrence (27.8), the master theorem does not apply to recurrence (27.10),

but Exercise 4.6-2 does. The solution is PMS 1 .n/ D ‚.lg

3

n/, and so the span of

P-MERGE-SORT is ‚.lg

3

n/.

Parallel merging gives P-MERGE-SORT a signiﬁcant parallelism advantage over

MERGE-SORT

0

. Recall that the parallelism of MERGE-SORT

0

, which calls the se-

rial MERGE procedure, is only ‚.lg n/. For P-MERGE-SORT, the parallelism is

PMS 1 .n/=PMS 1 .n/ D ‚.n lg n/=‚.lg

3

n/

D ‚.n= lg

2

n/ ;

which is much better both in theory and in practice. A good implementation in

practice would sacriﬁce some parallelism by coarsening the base case in order to

reduce the constants hidden by the asymptotic notation. The straightforward way

to coarsen the base case is to switch to an ordinary serial sort, perhaps quicksort,

when the size of the array is sufﬁciently small.

Exercises

27.3-1

Explain how to coarsen the base case of P-MERGE.

27.3-2

Instead of ﬁnding a median element in the larger subarray, as P-MERGE does, con-

sider a variant that ﬁnds a median element of all the elements in the two sorted

subarrays using the result of Exercise 9.3-8. Give pseudocode for an efﬁcient

multithreaded merging procedure that uses this median-ﬁnding procedure. Ana-

lyze your algorithm.

27.3-3

Give an efﬁcient multithreaded algorithm for partitioning an array around a pivot,

as is done by the PARTITION procedure on page 171. You need not partition the ar-

ray in place. Make your algorithm as parallel as possible. Analyze your algorithm.

(Hint: You may need an auxiliary array and may need to make more than one pass

over the input elements.)

27.3-4

Give a multithreaded version of RECURSIVE-FFT on page 911. Make your imple-

mentation as parallel as possible. Analyze your algorithm.

Problems for Chapter 27 805

27.3-5 ?

Give a multithreaded version of RANDOMIZED-SELECT on page 216. Make your

implementation as parallel as possible. Analyze your algorithm. (Hint: Use the

partitioning algorithm from Exercise 27.3-3.)

27.3-6 ?

Show how to multithread SELECT from Section 9.3. Make your implementation as

parallel as possible. Analyze your algorithm.

Problems

27-1 Implementing parallel loops using nested parallelism

Consider the following multithreaded algorithm for performing pairwise addition

on n-element arrays AŒ1 : : n and BŒ1 : : n, storing the sums in C Œ1 : : n:

SUM-ARRAYS.A; B; C /

1 parallel for i D 1 to A:length

2 C Œi D AŒi C BŒi

a. Rewrite the parallel loop in SUM-ARRAYS using nested parallelism (spawn

and sync) in the manner of MAT-VEC-MAIN-LOOP. Analyze the parallelism

of your implementation.

Consider the following alternative implementation of the parallel loop, which

contains a value grain-size to be speciﬁed:

SUM-ARRAYS

0 .A; B; C /

1 n D A:length

2 grain-size D ‹ // to be determined

3 r D dn=grain-sizee

4 for k D 0 to r 1

5 spawn ADD-SUBARRAY.A; B; C; k grain-size C 1;

min..k C 1/ grain-size; n//

6 sync

ADD-SUBARRAY.A; B; C; i; j /

1 for k D i to j

2 C Œk D AŒk C BŒk

806 Chapter 27 Multithreaded Algorithms

b. Suppose that we set grain-size D 1. What is the parallelism of this implemen-

tation?

c. Give a formula for the span of SUM-ARRAYS

0

in terms of n and grain-size.

Derive the best value for grain-size to maximize parallelism.

27-2 Saving temporary space in matrix multiplication

The P-MATRIX-MULTIPLY-RECURSIVE procedure has the disadvantage that it

must allocate a temporary matrix T of size n n, which can adversely affect the

constants hidden by the ‚-notation. The P-MATRIX-MULTIPLY-RECURSIVE pro-

cedure does have high parallelism, however. For example, ignoring the constants

in the ‚-notation, the parallelism for multiplying 1000 1000 matrices comes to

approximately 1000 3 =10 2 D 10 7

, since lg 1000 10. Most parallel computers

have far fewer than 10 million processors.

a. Describe a recursive multithreaded algorithm that eliminates the need for the

temporary matrix T at the cost of increasing the span to ‚.n/. (Hint: Com-

pute C D C C AB following the general strategy of P-MATRIX-MULTIPLY-

RECURSIVE, but initialize C in parallel and insert a sync in a judiciously cho-

sen location.)

b. Give and solve recurrences for the work and span of your implementation.

c. Analyze the parallelism of your implementation. Ignoring the constants in the

‚-notation, estimate the parallelism on 1000 1000 matrices. Compare with

the parallelism of P-MATRIX-MULTIPLY-RECURSIVE.

27-3 Multithreaded matrix algorithms

a. Parallelize the LU-DECOMPOSITION procedure on page 821 by giving pseu-

docode for a multithreaded version of this algorithm. Make your implementa-

tion as parallel as possible, and analyze its work, span, and parallelism.

b. Do the same for LUP-DECOMPOSITION on page 824.

c. Do the same for LUP-SOLVE on page 817.

d. Do the same for a multithreaded algorithm based on equation (28.13) for in-

verting a symmetric positive-deﬁnite matrix.

Problems for Chapter 27 807

27-4 Multithreading reductions and preﬁx computations

A ˝ -reduction of an array xŒ1 : : n, where ˝ is an associative operator, is the value

y D xŒ1 ˝ xŒ2 ˝ ˝ xŒn :

The following procedure computes the ˝-reduction of a subarray xŒi : : j serially.

REDUCE.x; i; j /

1 y D xŒi

2 for k D i C 1 to j

3 y D y ˝ xŒk

4 return y

a. Use nested parallelism to implement a multithreaded algorithm P-REDUCE,

which performs the same function with ‚.n/ work and ‚.lg n/ span. Analyze

your algorithm.

A related problem is that of computing a ˝ -preﬁx computation, sometimes

called a ˝ -scan, on an array xŒ1 : : n, where ˝ is once again an associative op-

erator. The ˝-scan produces the array yŒ1 : : n given by

yŒ1 D xŒ1 ;

yŒ2 D xŒ1 ˝ xŒ2 ;

yŒ3 D xŒ1 ˝ xŒ2 ˝ xŒ3 ;

:

:

:

yŒn D xŒ1 ˝ xŒ2 ˝ xŒ3 ˝ ˝ xŒn ;

that is, all preﬁxes of the array x “summed” using the ˝ operator. The following

serial procedure SCAN performs a ˝-preﬁx computation:

SCAN.x/

1 n D x:length

2 let yŒ1 : : n be a new array

3 yŒ1 D xŒ1

4 for i D 2 to n

5 yŒi D yŒi 1 ˝ xŒi

6 return y

Unfortunately, multithreading SCAN is not straightforward. For example, changing

the for loop to a parallel for loop would create races, since each iteration of the

loop body depends on the previous iteration. The following procedure P-SCAN-1

performs the ˝-preﬁx computation in parallel, albeit inefﬁciently:

808 Chapter 27 Multithreaded Algorithms

P-SCAN-1.x/

1 n D x:length

2 let yŒ1 : : n be a new array

3 P-SCAN-1-AUX.x; y; 1; n/

4 return y

P-SCAN-1-AUX.x; y; i; j /

1 parallel for l D i to j

2 yŒl D P-REDUCE.x; 1; l/

b. Analyze the work, span, and parallelism of P-SCAN-1.

By using nested parallelism, we can obtain a more efﬁcient ˝-preﬁx computa-

tion:

P-SCAN-2.x/

1 n D x:length

2 let yŒ1 : : n be a new array

3 P-SCAN-2-AUX.x; y; 1; n/

4 return y

P-SCAN-2-AUX.x; y; i; j /

1 if i == j

2 yŒi D xŒi

3 else k D b.i C j /=2c

4 spawn P-SCAN-2-AUX.x; y; i; k/

5 P-SCAN-2-AUX.x; y; k C 1; j /

6 sync

7 parallel for l D k C 1 to j

8 yŒl D yŒk ˝ yŒl

c. Argue that P-SCAN-2 is correct, and analyze its work, span, and parallelism.

We can improve on both P-SCAN-1 and P-SCAN-2 by performing the ˝-preﬁx

computation in two distinct passes over the data. On the ﬁrst pass, we gather the

terms for various contiguous subarrays of x into a temporary array t, and on the

second pass we use the terms in t to compute the ﬁnal result y. The following

pseudocode implements this strategy, but certain expressions have been omitted:

Problems for Chapter 27 809

P-SCAN-3.x/

1 n D x:length

2 let yŒ1 : : n and tŒ1 : : n be new arrays

3 yŒ1 D xŒ1

4 if n > 1

5 P-SCAN-UP.x; t; 2; n/

6 P-SCAN-DOWN.xŒ1; x; t; y; 2; n/

7 return y

P-SCAN-UP.x; t; i; j /

1 if i == j

2 return xŒi

3 else

4 k D b.i C j /=2c

5 tŒk D spawn P-SCAN-UP.x; t; i; k/

6 right D P-SCAN-UP.x; t; k C 1; j /

7 sync

8 return // ﬁll in the blank

P-SCAN-DOWN.; x; t; y; i; j /

1 if i == j

2 yŒi D ˝ xŒi

3 else

4 k D b.i C j /=2c

5 spawn P-SCAN-DOWN. ; x; t; y; i; k/ // ﬁll in the blank

6 P-SCAN-DOWN. ; x; t; y; k C 1; j / // ﬁll in the blank

7 sync

d. Fill in the three missing expressions in line 8 of P-SCAN-UP and lines 5 and 6

of P-SCAN-DOWN. Argue that with expressions you supplied, P-SCAN-3 is

correct. (Hint: Prove that the value passed to P-SCAN-DOWN.; x; t; y; i; j /

satisﬁes D xŒ1 ˝ xŒ2 ˝ ˝ xŒi 1.)

e. Analyze the work, span, and parallelism of P-SCAN-3.

27-5 Multithreading a simple stencil calculation

Computational science is replete with algorithms that require the entries of an array

to be ﬁlled in with values that depend on the values of certain already computed

neighboring entries, along with other information that does not change over the

course of the computation. The pattern of neighboring entries does not change

during the computation and is called a stencil. For example, Section 15.4 presents

810 Chapter 27 Multithreaded Algorithms

a stencil algorithm to compute a longest common subsequence, where the value in

entry cŒi; j depends only on the values in cŒi 1; j , cŒi; j 1, and cŒi 1; j 1,

as well as the elements x i and y j within the two sequences given as inputs. The

input sequences are ﬁxed, but the algorithm ﬁlls in the two-dimensional array c so

that it computes entry cŒi; j after computing all three entries cŒi 1; j , cŒi; j 1,

and cŒi 1; j 1.

In this problem, we examine how to use nested parallelism to multithread a

simple stencil calculation on an n n array A in which, of the values in A, the

value placed into entry AŒi; j depends only on values in AŒi 0 ; j 0 , where i 0 i

and j 0 j (and of course, i 0 ¤ i or j 0 ¤ j ). In other words, the value in an

entry depends only on values in entries that are above it and/or to its left, along

with static information outside of the array. Furthermore, we assume throughout

this problem that once we have ﬁlled in the entries upon which AŒi; j depends, we

can ﬁll in AŒi; j in ‚.1/ time (as in the LCS-LENGTH procedure of Section 15.4).

We can partition the n n array A into four n=2 n=2 subarrays as follows:

A D

A 11 A 12

A 21 A 22

: (27.11)

Observe now that we can ﬁll in subarray A 11 recursively, since it does not depend

on the entries of the other three subarrays. Once A 11 is complete, we can continue

to ﬁll in A 12 and A 21 recursively in parallel, because although they both depend

on A 11 , they do not depend on each other. Finally, we can ﬁll in A 22 recursively.

a. Give multithreaded pseudocode that performs this simple stencil calculation

using a divide-and-conquer algorithm SIMPLE-STENCIL based on the decom-

position (27.11) and the discussion above. (Don’t worry about the details of the

base case, which depends on the speciﬁc stencil.) Give and solve recurrences

for the work and span of this algorithm in terms of n. What is the parallelism?

b. Modify your solution to part (a) to divide an n n array into nine n=3 n=3

subarrays, again recursing with as much parallelism as possible. Analyze this

algorithm. How much more or less parallelism does this algorithm have com-

pared with the algorithm from part (a)?

c. Generalize your solutions to parts (a) and (b) as follows. Choose an integer

b 2. Divide an n n array into b 2

subarrays, each of size n=b n=b, recursing

with as much parallelism as possible. In terms of n and b, what are the work,

span, and parallelism of your algorithm? Argue that, using this approach, the

parallelism must be o.n/ for any choice of b 2. (Hint: For this last argument,

show that the exponent of n in the parallelism is strictly less than 1 for any

choice of b 2.)

Notes for Chapter 27 811

d. Give pseudocode for a multithreaded algorithm for this simple stencil calcu-

lation that achieves ‚.n= lg n/ parallelism. Argue using notions of work and

span that the problem, in fact, has ‚.n/ inherent parallelism. As it turns out,

the divide-and-conquer nature of our multithreaded pseudocode does not let us

achieve this maximal parallelism.

27-6 Randomized multithreaded algorithms

Just as with ordinary serial algorithms, we sometimes want to implement random-

ized multithreaded algorithms. This problem explores how to adapt the various

performance measures in order to handle the expected behavior of such algorithms.

It also asks you to design and analyze a multithreaded algorithm for randomized

quicksort.

a. Explain how to modify the work law (27.2), span law (27.3), and greedy sched-

uler bound (27.4) to work with expectations when T P , T 1 , and T 1 are all ran-

dom variables.

b. Consider a randomized multithreaded algorithm for which 1% of the time we

have T 1 D 10 4

and T 10;000 D 1, but for 99% of the time we have T 1 D

T 10;000 D 10 9

. Argue that the speedup of a randomized multithreaded algo-

rithm should be deﬁned as E ŒT 1 =E ŒT P , rather than E ŒT 1 =T P .

c. Argue that the parallelism of a randomized multithreaded algorithm should be

deﬁned as the ratio E ŒT 1 =E ŒT 1 .

d. Multithread the RANDOMIZED-QUICKSORT algorithm on page 179 by using

nested parallelism. (Do not parallelize RANDOMIZED-PARTITION.) Give the

pseudocode for your P-RANDOMIZED-QUICKSORT algorithm.

e. Analyze your multithreaded algorithm for randomized quicksort. (Hint: Re-

view the analysis of RANDOMIZED-SELECT on page 216.)

Chapter notes

Parallel computers, models for parallel computers, and algorithmic models for par-

allel programming have been around in various forms for years. Prior editions of

this book included material on sorting networks and the PRAM (Parallel Random-

Access Machine) model. The data-parallel model [48, 168] is another popular al-

gorithmic programming model, which features operations on vectors and matrices

as primitives.

812 Chapter 27 Multithreaded Algorithms

Graham [149] and Brent [55] showed that there exist schedulers achieving the

bound of Theorem 27.1. Eager, Zahorjan, and Lazowska [98] showed that any

greedy scheduler achieves this bound and proposed the methodology of using work

and span (although not by those names) to analyze parallel algorithms. Blelloch

[47] developed an algorithmic programming model based on work and span (which

he called the “depth” of the computation) for data-parallel programming. Blumofe

and Leiserson [52] gave a distributed scheduling algorithm for dynamic multi-

threading based on randomized “work-stealing” and showed that it achieves the

bound E ŒT P T 1 =P C O.T 1 /. Arora, Blumofe, and Plaxton [19] and Blelloch,

Gibbons, and Matias [49] also provided provably good algorithms for scheduling

dynamic multithreaded computations.

The multithreaded pseudocode and programming model were heavily inﬂuenced

by the Cilk [51, 118] project at MIT and the Cilk++ [71] extensions to C++ dis-

tributed by Cilk Arts, Inc. Many of the multithreaded algorithms in this chapter

appeared in unpublished lecture notes by C. E. Leiserson and H. Prokop and have

been implemented in Cilk or Cilk++. The multithreaded merge-sorting algorithm

was inspired by an algorithm of Akl [12].

The notion of sequential consistency is due to Lamport [223].

28 Matrix Operations

Because operations on matrices lie at the heart of scientiﬁc computing, efﬁcient al-

gorithms for working with matrices have many practical applications. This chapter

focuses on how to multiply matrices and solve sets of simultaneous linear equa-

tions. Appendix D reviews the basics of matrices.

Section 28.1 shows how to solve a set of linear equations using LUP decomposi-

tions. Then, Section 28.2 explores the close relationship between multiplying and

inverting matrices. Finally, Section 28.3 discusses the important class of symmetric

positive-deﬁnite matrices and shows how we can use them to ﬁnd a least-squares

solution to an overdetermined set of linear equations.

One important issue that arises in practice is numerical stability. Due to the

limited precision of ﬂoating-point representations in actual computers, round-off

errors in numerical computations may become ampliﬁed over the course of a com-

putation, leading to incorrect results; we call such computations numerically un-

stable. Although we shall brieﬂy consider numerical stability on occasion, we do

not focus on it in this chapter. We refer you to the excellent book by Golub and

Van Loan [144] for a thorough discussion of stability issues.

28.1 Solving systems of linear equations

Numerous applications need to solve sets of simultaneous linear equations. We

can formulate a linear system as a matrix equation in which each matrix or vector

element belongs to a ﬁeld, typically the real numbers R . This section discusses how

to solve a system of linear equations using a method called LUP decomposition.

We start with a set of linear equations in n unknowns x 1 ; x 2 ; : : : ; x n :

814 Chapter 28 Matrix Operations

a 11 x 1 C a 12 x 2 C C a 1n x n D b 1 ;

a 21 x 1 C a 22 x 2 C C a 2n x n D b 2 ;

:

:

:

a n1 x 1 C a n2 x 2 C C a nn x n D b n :

(28.1)

A solution to the equations (28.1) is a set of values for x 1 ; x 2 ; : : : ; x n that satisfy

all of the equations simultaneously. In this section, we treat only the case in which

there are exactly n equations in n unknowns.

We can conveniently rewrite equations (28.1) as the matrix-vector equation ˙

a 11 a 12 a 1n

a 21 a 22 a 2n

:

:

:

:

:

:

:

:

:

:

:

:

a n1 a n2 a nn

˙

x 1

x 2

:

:

:

x n

D

˙

b 1

b 2

:

:

:

b n

or, equivalently, letting A D .a ij /, x D .x i /, and b D .b i /, as

Ax D b : (28.2)

If A is nonsingular, it possesses an inverse A 1

, and

x D A

1

b (28.3)

is the solution vector. We can prove that x is the unique solution to equation (28.2)

as follows. If there are two solutions, x and x 0

, then Ax D Ax 0 D b and, letting I

denote an identity matrix,

x D Ix

D .A

1

A/x

D A

1

.Ax/

D A

1

.Ax

0

/

D .A

1

A/x

0

D x

0

:

In this section, we shall be concerned predominantly with the case in which A

is nonsingular or, equivalently (by Theorem D.1), the rank of A is equal to the

number n of unknowns. There are other possibilities, however, which merit a brief

discussion. If the number of equations is less than the number n of unknowns—or,

more generally, if the rank of A is less than n—then the system is underdeter-

mined. An underdetermined system typically has inﬁnitely many solutions, al-

though it may have no solutions at all if the equations are inconsistent. If the

number of equations exceeds the number n of unknowns, the system is overdeter-

mined, and there may not exist any solutions. Section 28.3 addresses the important

28.1 Solving systems of linear equations 815

problem of ﬁnding good approximate solutions to overdetermined systems of linear

equations.

Let us return to our problem of solving the system Ax D b of n equations in n

unknowns. We could compute A 1

and then, using equation (28.3), multiply b

by A 1

, yielding x D A 1 b. This approach suffers in practice from numerical

instability. Fortunately, another approach—LUP decomposition—is numerically

stable and has the further advantage of being faster in practice.

Overview of LUP decomposition

The idea behind LUP decomposition is to ﬁnd three n n matrices L, U , and P

such that

PA D LU ; (28.4)

where

L is a unit lower-triangular matrix,

U is an upper-triangular matrix, and

P is a permutation matrix.

We call matrices L, U , and P satisfying equation (28.4) an LUP decomposition

of the matrix A. We shall show that every nonsingular matrix A possesses such a

decomposition.

Computing an LUP decomposition for the matrix A has the advantage that we

can more easily solve linear systems when they are triangular, as is the case for

both matrices L and U . Once we have found an LUP decomposition for A, we

can solve equation (28.2), Ax D b, by solving only triangular linear systems, as

follows. Multiplying both sides of Ax D b by P yields the equivalent equation

PAx D P b, which, by Exercise D.1-4, amounts to permuting the equations (28.1).

Using our decomposition (28.4), we obtain

LUx D P b :

We can now solve this equation by solving two triangular linear systems. Let us

deﬁne y D Ux, where x is the desired solution vector. First, we solve the lower-

triangular system

Ly D P b (28.5)

for the unknown vector y by a method called “forward substitution.” Having solved

for y, we then solve the upper-triangular system

Ux D y (28.6)

816 Chapter 28 Matrix Operations

for the unknown x by a method called “back substitution.” Because the permu-

tation matrix P is invertible (Exercise D.2-3), multiplying both sides of equa-

tion (28.4) by P 1

gives P 1 PA D P 1 LU , so that

A D P

1

LU : (28.7)

Hence, the vector x is our solution to Ax D b:

Ax D P

1

LUx (by equation (28.7))

D P

1

Ly (by equation (28.6))

D P

1

P b (by equation (28.5))

D b :

Our next step is to show how forward and back substitution work and then attack

the problem of computing the LUP decomposition itself.

Forward and back substitution

Forward substitution can solve the lower-triangular system (28.5) in ‚.n 2 / time,

given L, P , and b. For convenience, we represent the permutation P compactly

by an array Œ1 : : n. For i D 1; 2; : : : ; n, the entry Œi indicates that P i;Œi D 1

and P ij D 0 for j ¤ Œi. Thus, PA has a Œi;j in row i and column j , and P b

has b Œi as its ith element. Since L is unit lower-triangular, we can rewrite equa-

tion (28.5) as

y 1 D b Œ1 ;

l 21 y 1 C y 2 D b Œ2 ;

l 31 y 1 C l 32 y 2 C y 3 D b Œ3 ;

:

:

:

l n1 y 1 C l n2 y 2 C l n3 y 3 C C y n D b Œn :

The ﬁrst equation tells us that y 1 D b Œ1 . Knowing the value of y 1 , we can

substitute it into the second equation, yielding

y 2 D b Œ2 l 21 y 1 :

Now, we can substitute both y 1 and y 2 into the third equation, obtaining

y 3 D b Œ3 .l 31 y 1 C l 32 y 2 / :

In general, we substitute y 1 ; y 2 ; : : : ; y i1 “forward” into the ith equation to solve

for y i :

28.1 Solving systems of linear equations 817

y i D b Œi

i1 X

j D1

l ij y j :

Having solved for y, we solve for x in equation (28.6) using back substitution,

which is similar to forward substitution. Here, we solve the nth equation ﬁrst and

work backward to the ﬁrst equation. Like forward substitution, this process runs

in ‚.n 2 / time. Since U is upper-triangular, we can rewrite the system (28.6) as

u 11 x 1 C u 12 x 2 C C u 1;n2 x n2 C u 1;n1 x n1 C u 1n x n D y 1 ;

u 22 x 2 C C u 2;n2 x n2 C u 2;n1 x n1 C u 2n x n D y 2 ;

:

:

:

u n2;n2 x n2 C u n2;n1 x n1 C u n2;n x n D y n2 ;

u n1;n1 x n1 C u n1;n x n D y n1 ;

u n;n x n D y n :

Thus, we can solve for x n ; x n1 ; : : : ; x 1 successively as follows:

x n D y n =u n;n ;

x n1 D .y n1 u n1;n x n /=u n1;n1 ;

x n2 D .y n2 .u n2;n1 x n1 C u n2;n x n //=u n2;n2 ;

:

:

:

or, in general,

x i D

y i

n X

j DiC1

u ij x j

!

=u ii :

Given P , L, U , and b, the procedure LUP-SOLVE solves for x by combining

forward and back substitution. The pseudocode assumes that the dimension n ap-

pears in the attribute L:rows and that the permutation matrix P is represented by

the array .

LUP-SOLVE.L; U; ; b/

1 n D L:rows

2 let x be a new vector of length n

3 for i D 1 to n

4 y i D b Œi

P i1

j D1

l ij y j

5 for i D n downto 1

6 x i D

y i

P n

j DiC1

u ij x j

=u ii

7 return x

818 Chapter 28 Matrix Operations

Procedure LUP-SOLVE solves for y using forward substitution in lines 3–4, and

then it solves for x using backward substitution in lines 5–6. Since the summation

within each of the for loops includes an implicit loop, the running time is ‚.n 2 /.

As an example of these methods, consider the system of linear equations deﬁned

by

1 2 0

3 4 4

5 6 3

x D

3

7

8

;

where

A D

1 2 0

3 4 4

5 6 3

;

b D

3

7

8

;

and we wish to solve for the unknown x. The LUP decomposition is

L D

1 0 0

0:2 1 0

0:6 0:5 1

;

U D

5 6 3

0 0:8 0:6

0 0 2:5

;

P D

0 0 1

1 0 0

0 1 0

:

(You might want to verify that PA D LU .) Using forward substitution, we solve

Ly D P b for y:

1 0 0

0:2 1 0

0:6 0:5 1

y 1

y 2

y 3

D

8

3

7

;

obtaining

y D

8

1:4

1:5

by computing ﬁrst y 1 , then y 2 , and ﬁnally y 3 . Using back substitution, we solve

Ux D y for x:

28.1 Solving systems of linear equations 819

5 6 3

0 0:8 0:6

0 0 2:5

x 1

x 2

x 3

D

8

1:4

1:5

;

thereby obtaining the desired answer

x D

1:4

2:2

0:6

by computing ﬁrst x 3 , then x 2 , and ﬁnally x 1 .

Computing an LU decomposition

We have now shown that if we can create an LUP decomposition for a nonsingular

matrix A, then forward and back substitution can solve the system Ax D b of

linear equations. Now we show how to efﬁciently compute an LUP decomposition

for A. We start with the case in which A is an n n nonsingular matrix and P is

absent (or, equivalently, P D I n ). In this case, we factor A D LU . We call the

two matrices L and U an LU decomposition of A.

We use a process known as Gaussian elimination to create an LU decomposi-

tion. We start by subtracting multiples of the ﬁrst equation from the other equations

in order to remove the ﬁrst variable from those equations. Then, we subtract mul-

tiples of the second equation from the third and subsequent equations so that now

the ﬁrst and second variables are removed from them. We continue this process

until the system that remains has an upper-triangular form—in fact, it is the ma-

trix U . The matrix L is made up of the row multipliers that cause variables to be

eliminated.

Our algorithm to implement this strategy is recursive. We wish to construct an

LU decomposition for an n n nonsingular matrix A. If n D 1, then we are done,

since we can choose L D I 1 and U D A. For n > 1, we break A into four parts:

A D

˙

a 11 a 12 a 1n

a 21 a 22 a 2n

:

:

:

:

:

:

:

:

:

:

:

:

a n1 a n2 a nn

D

a 11 w T

A 0

;

where is a column .n 1/-vector, w T is a row .n 1/-vector, and A 0

is an

.n 1/ .n 1/ matrix. Then, using matrix algebra (verify the equations by

820 Chapter 28 Matrix Operations

simply multiplying through), we can factor A as

A D

a 11 w T

A 0

D

1 0

=a 11 I n1

a 11 w T

0 A 0 w T =a 11

: (28.8)

The 0s in the ﬁrst and second matrices of equation (28.8) are row and col-

umn .n 1/-vectors, respectively. The term w T =a 11 , formed by taking the

outer product of and w and dividing each element of the result by a 11 , is an

.n 1/ .n 1/ matrix, which conforms in size to the matrix A 0

from which it is

subtracted. The resulting .n 1/ .n 1/ matrix

A

0

w T =a 11 (28.9)

is called the Schur complement of A with respect to a 11 .

We claim that if A is nonsingular, then the Schur complement is nonsingular,

too. Why? Suppose that the Schur complement, which is .n 1/ .n 1/, is

singular. Then by Theorem D.1, it has row rank strictly less than n 1. Because

the bottom n 1 entries in the ﬁrst column of the matrix

a 11 w T

0 A 0 w T =a 11

are all 0, the bottom n 1 rows of this matrix must have row rank strictly less

than n 1. The row rank of the entire matrix, therefore, is strictly less than n.

Applying Exercise D.2-8 to equation (28.8), A has rank strictly less than n, and

from Theorem D.1 we derive the contradiction that A is singular.

Because the Schur complement is nonsingular, we can now recursively ﬁnd an

LU decomposition for it. Let us say that

A

0

w T =a 11 D L

0

U

0

;

where L 0

is unit lower-triangular and U 0

is upper-triangular. Then, using matrix

algebra, we have

A D

1 0

=a 11 I n1

a 11 w T

0 A 0 w T =a 11

D

1 0

=a 11 I n1

a 11 w T

0 L 0 U 0

D

1 0

=a 11 L 0

a 11 w T

0 U 0

D LU ;

thereby providing our LU decomposition. (Note that because L 0

is unit lower-

triangular, so is L, and because U 0

is upper-triangular, so is U .)

28.1 Solving systems of linear equations 821

Of course, if a 11 D 0, this method doesn’t work, because it divides by 0. It also

doesn’t work if the upper leftmost entry of the Schur complement A 0 w T =a 11

is 0, since we divide by it in the next step of the recursion. The elements by

which we divide during LU decomposition are called pivots, and they occupy the

diagonal elements of the matrix U . The reason we include a permutation matrix P

during LUP decomposition is that it allows us to avoid dividing by 0. When we use

permutations to avoid division by 0 (or by small numbers, which would contribute

to numerical instability), we are pivoting.

An important class of matrices for which LU decomposition always works cor-

rectly is the class of symmetric positive-deﬁnite matrices. Such matrices require

no pivoting, and thus we can employ the recursive strategy outlined above with-

out fear of dividing by 0. We shall prove this result, as well as several others, in

Section 28.3.

Our code for LU decomposition of a matrix A follows the recursive strategy, ex-

cept that an iteration loop replaces the recursion. (This transformation is a standard

optimization for a “tail-recursive” procedure—one whose last operation is a recur-

sive call to itself. See Problem 7-4.) It assumes that the attribute A:rows gives

the dimension of A. We initialize the matrix U with 0s below the diagonal and

matrix L with 1s on its diagonal and 0s above the diagonal.

LU-DECOMPOSITION.A/

1 n D A:rows

2 let L and U be new n n matrices

3 initialize U with 0s below the diagonal

4 initialize L with 1s on the diagonal and 0s above the diagonal

5 for k D 1 to n

6 u kk D a kk

7 for i D k C 1 to n

8 l ik D a ik =u kk // l ik holds i

9 u ki D a ki // u ki holds w T

i

10 for i D k C 1 to n

11 for j D k C 1 to n

12 a ij D a ij l ik u kj

13 return L and U

The outer for loop beginning in line 5 iterates once for each recursive step. Within

this loop, line 6 determines the pivot to be u kk D a kk . The for loop in lines 7–9

(which does not execute when k D n), uses the and w T vectors to update L

and U . Line 8 determines the elements of the vector, storing i in l ik , and line 9

computes the elements of the w T vector, storing w T

i

in u ki . Finally, lines 10–12

compute the elements of the Schur complement and store them back into the ma-

822 Chapter 28 Matrix Operations

2 3 1 5

6 13 5 19

2 19 10 23

4 10 11 31

(a)

3 1 5

3 4 2 4

1 16 9 18

2 4 9 21

(b)

2 3 1 5

3 2 4

1 4 1 2

2 1 7 17

(c)

2 3 1 5

3 4 2 4

1 4 2

2 1 7 3

(d)

(e)

2

4

1

2 3 1 5

6 13 5 19

2 19 10 23

4 10 11 31

˘

D

1 0 0 0

3 1 0 0

1 4 1 0

2 1 7 1

˘

2 3 1 5

0 4 2 4

0 0 1 2

0 0 0 3

˘

A L U

Figure 28.1 The operation of LU-DECOMPOSITION. (a) The matrix A. (b) The element a11 D 2

in the black circle is the pivot, the shaded column is =a11, and the shaded row is w

T

. The elements

of U computed thus far are above the horizontal line, and the elements of L are to the left of the

vertical line. The Schur complement matrix A

0

w

T

=a11 occupies the lower right. (c) We now

operate on the Schur complement matrix produced from part (b). The element a22 D 4 in the black

circle is the pivot, and the shaded column and row are =a22 and w

T

(in the partitioning of the Schur

complement), respectively. Lines divide the matrix into the elements of U computed so far (above),

the elements of L computed so far (left), and the new Schur complement (lower right). (d) After the

next step, the matrix A is factored. (The element 3 in the new Schur complement becomes part of U

when the recursion terminates.) (e) The factorization A D LU .

trix A. (We don’t need to divide by a kk in line 12 because we already did so when

we computed l ik in line 8.) Because line 12 is triply nested, LU-DECOMPOSITION

runs in time ‚.n 3 /.

Figure 28.1 illustrates the operation of LU-DECOMPOSITION. It shows a stan-

dard optimization of the procedure in which we store the signiﬁcant elements of L

and U in place in the matrix A. That is, we can set up a correspondence between

each element a ij and either l ij (if i > j ) or u ij (if i j ) and update the ma-

trix A so that it holds both L and U when the procedure terminates. To obtain

the pseudocode for this optimization from the above pseudocode, just replace each

reference to l or u by a; you can easily verify that this transformation preserves

correctness.

Computing an LUP decomposition

Generally, in solving a system of linear equations Ax D b, we must pivot on off-

diagonal elements of A to avoid dividing by 0. Dividing by 0 would, of course,

be disastrous. But we also want to avoid dividing by a small value—even if A is

28.1 Solving systems of linear equations 823

nonsingular—because numerical instabilities can result. We therefore try to pivot

on a large value.

The mathematics behind LUP decomposition is similar to that of LU decom-

position. Recall that we are given an n n nonsingular matrix A, and we wish

to ﬁnd a permutation matrix P , a unit lower-triangular matrix L, and an upper-

triangular matrix U such that PA D LU . Before we partition the matrix A, as we

did for LU decomposition, we move a nonzero element, say a k1 , from somewhere

in the ﬁrst column to the .1; 1/ position of the matrix. For numerical stability, we

choose a k1 as the element in the ﬁrst column with the greatest absolute value. (The

ﬁrst column cannot contain only 0s, for then A would be singular, because its de-

terminant would be 0, by Theorems D.4 and D.5.) In order to preserve the set of

equations, we exchange row 1 with row k, which is equivalent to multiplying A by

a permutation matrix Q on the left (Exercise D.1-4). Thus, we can write QA as

QA D

a k1 w T

A 0

;

where D .a 21 ; a 31 ; : : : ; a n1 / T , except that a 11 replaces a k1 ; w T D .a k2 ; a k3 ;

: : : ; a kn /; and A 0

is an .n1/ .n1/ matrix. Since a k1 ¤ 0, we can now perform

much the same linear algebra as for LU decomposition, but now guaranteeing that

we do not divide by 0:

QA D

a k1 w T

A 0

D

1 0

=a k1 I n1

a k1 w T

0 A 0 w T =a k1

:

As we saw for LU decomposition, if A is nonsingular, then the Schur comple-

ment A 0 w T =a k1 is nonsingular, too. Therefore, we can recursively ﬁnd an

LUP decomposition for it, with unit lower-triangular matrix L 0

, upper-triangular

matrix U 0

, and permutation matrix P 0

, such that

P

0

.A

0

w T =a k1 / D L

0

U

0

:

Deﬁne

P D

1 0

0 P 0

Q ;

which is a permutation matrix, since it is the product of two permutation matrices

(Exercise D.1-4). We now have

824 Chapter 28 Matrix Operations

PA D

1 0

0 P 0

QA

D

1 0

0 P 0

1 0

=a k1 I n1

a k1 w T

0 A 0 w T =a k1

D

1 0

P 0 =a k1 P 0

a k1 w T

0 A 0 w T =a k1

D

1 0

P 0 =a k1 I n1

a k1 w T

0 P 0 .A 0 w T =a k1 /

D

1 0

P 0 =a k1 I n1

a k1 w T

0 L 0 U 0

D

1 0

P 0 =a k1 L 0

a k1 w T

0 U 0

D LU ;

yielding the LUP decomposition. Because L 0

is unit lower-triangular, so is L, and

because U 0

is upper-triangular, so is U .

Notice that in this derivation, unlike the one for LU decomposition, we must

multiply both the column vector =a k1 and the Schur complement A 0 w T =a k1

by the permutation matrix P 0

. Here is the pseudocode for LUP decomposition:

LUP-DECOMPOSITION.A/

1 n D A:rows

2 let Œ1 : : n be a new array

3 for i D 1 to n

4 Œi D i

5 for k D 1 to n

6 p D 0

7 for i D k to n

8 if ja ik j > p

9 p D ja ik j

10 k 0 D i

11 if p == 0

12 error “singular matrix”

13 exchange Œk with Œk 0

14 for i D 1 to n

15 exchange a ki with a k 0 i

16 for i D k C 1 to n

17 a ik D a ik =a kk

18 for j D k C 1 to n

19 a ij D a ij a ik a kj

28.1 Solving systems of linear equations 825

Like LU-DECOMPOSITION, our LUP-DECOMPOSITION procedure replaces

the recursion with an iteration loop. As an improvement over a direct implemen-

tation of the recursion, we dynamically maintain the permutation matrix P as an

array , where Œi D j means that the ith row of P contains a 1 in column j .

We also implement the code to compute L and U “in place” in the matrix A. Thus,

when the procedure terminates,

a ij D

(

l ij if i > j ;

u ij if i j :

Figure 28.2 illustrates how LUP-DECOMPOSITION factors a matrix. Lines 3–4

initialize the array to represent the identity permutation. The outer for loop

beginning in line 5 implements the recursion. Each time through the outer loop,

lines 6–10 determine the element a k 0 k with largest absolute value of those in the

current ﬁrst column (column k) of the .n k C 1/ .n k C 1/ matrix whose

LUP decomposition we are ﬁnding. If all elements in the current ﬁrst column are

zero, lines 11–12 report that the matrix is singular. To pivot, we exchange Œk 0

with Œk in line 13 and exchange the kth and k 0

th rows of A in lines 14–15,

thereby making the pivot element a kk . (The entire rows are swapped because in

the derivation of the method above, not only is A 0 w T =a k1 multiplied by P 0

, but

so is =a k1 .) Finally, the Schur complement is computed by lines 16–19 in much

the same way as it is computed by lines 7–12 of LU-DECOMPOSITION, except that

here the operation is written to work in place.

Because of its triply nested loop structure, LUP-DECOMPOSITION has a run-

ning time of ‚.n 3 /, which is the same as that of LU-DECOMPOSITION. Thus,

pivoting costs us at most a constant factor in time.

Exercises

28.1-1

Solve the equation

1 0 0

4 1 0

6 5 1

x 1

x 2

x 3

D

3

14

7

by using forward substitution.

28.1-2

Find an LU decomposition of the matrix

4 5 6

8 6 7

12 7 12

:

826 Chapter 28 Matrix Operations

2 0 2 0.6

3 3 4 –2

5 5 4 2

–1 –2 3.4 –1

(a)

1

2

3

4

2 0 2 0.6

3 3 4 –2

5 5 4 2

–1 –2 3.4 –1

(b)

3

2

1

4

0.4 –2 0.4 –.2

0.6 0 1.6 –3.2

5 5 4 2

–0.2 –1 4.2 –0.6

(c)

3

2

1

4

0.4 –2 0.4 –0.2

0.6 0 1.6 –3.2

5 5 4 2

–0.2 –1 4.2 –0.6

(d)

3

2

1

4

0.4 –2 0.4 –0.2

0.6 0 1.6 –3.2

5 5 4 2

–0.2 –1 4.2 –0.6

(e)

3

2

1

4

0.4 –2 0.4 –0.2

0.6 0 1.6 –3.2

5 5 4 2

–0.2 0.5 4 –0.5

(f)

3

2

1

4

0.4 –2 0.4 –0.2

0.6 0 1.6 –3.2

5 5 4 2

–0.2 0.5 4 –0.5

(g)

3

2

1

4

0.4 –2 0.4 –0.2

0.6 0 1.6 –3.2

5 5 4 2

–0.2 0.5 4 –0.5

(h)

3

2

1

4

0.4 –2 0.4 –0.2

0.6 0 0.4 –3

5 5 4 2

–0.2 0.5 4 –0.5

(i)

3

2

1

4

(j)

0 0 1 0

1 0 0 0

0 0 0 1

0 1 0 0

˘

2 0 2 0:6

3 3 4 2

5 5 4 2

1 2 3:4 1

˘

D

1 0 0 0

0:4 1 0 0

0:2 0:5 1 0

0:6 0 0:4 1

˘

5 5 4 2

0 2 0:4 0:2

0 0 4 0:5

0 0 0 3

˘

P A L U

Figure 28.2 The operation of LUP-DECOMPOSITION. (a) The input matrix A with the identity

permutation of the rows on the left. The ﬁrst step of the algorithm determines that the element 5

in the black circle in the third row is the pivot for the ﬁrst column. (b) Rows 1 and 3 are swapped

and the permutation is updated. The shaded column and row represent and w

T

. (c) The vector

is replaced by =5, and the lower right of the matrix is updated with the Schur complement. Lines

divide the matrix into three regions: elements of U (above), elements of L (left), and elements of the

Schur complement (lower right). (d)–(f) The second step. (g)–(i) The third step. No further changes

occur on the fourth (ﬁnal) step. (j) The LUP decomposition PA D LU .

28.2 Inverting matrices 827

28.1-3

Solve the equation

1 5 4

2 0 3

5 8 2

x 1

x 2

x 3

D

12

9

5

by using an LUP decomposition.

28.1-4

Describe the LUP decomposition of a diagonal matrix.

28.1-5

Describe the LUP decomposition of a permutation matrix A, and prove that it is

unique.

28.1-6

Show that for all n 1, there exists a singular n n matrix that has an LU decom-

position.

28.1-7

In LU-DECOMPOSITION, is it necessary to perform the outermost for loop itera-

tion when k D n? How about in LUP-DECOMPOSITION?

28.2 Inverting matrices

Although in practice we do not generally use matrix inverses to solve systems of

linear equations, preferring instead to use more numerically stable techniques such

as LUP decomposition, sometimes we need to compute a matrix inverse. In this

section, we show how to use LUP decomposition to compute a matrix inverse.

We also prove that matrix multiplication and computing the inverse of a matrix

are equivalently hard problems, in that (subject to technical conditions) we can

use an algorithm for one to solve the other in the same asymptotic running time.

Thus, we can use Strassen’s algorithm (see Section 4.2) for matrix multiplication

to invert a matrix. Indeed, Strassen’s original paper was motivated by the problem

of showing that a set of a linear equations could be solved more quickly than by

the usual method.

828 Chapter 28 Matrix Operations

Computing a matrix inverse from an LUP decomposition

Suppose that we have an LUP decomposition of a matrix A in the form of three

matrices L, U , and P such that PA D LU . Using LUP-SOLVE, we can solve

an equation of the form Ax D b in time ‚.n 2 /. Since the LUP decomposition

depends on A but not b, we can run LUP-SOLVE on a second set of equations of

the form Ax D b 0

in additional time ‚.n 2 /. In general, once we have the LUP

decomposition of A, we can solve, in time ‚.kn 2 /, k versions of the equation

Ax D b that differ only in b.

We can think of the equation

AX D I n ; (28.10)

which deﬁnes the matrix X, the inverse of A, as a set of n distinct equations of the

form Ax D b. To be precise, let X i denote the ith column of X, and recall that the

unit vector e i is the ith column of I n . We can then solve equation (28.10) for X by

using the LUP decomposition for A to solve each equation

AX i D e i

separately for X i . Once we have the LUP decomposition, we can compute each of

the n columns X i in time ‚.n 2 /, and so we can compute X from the LUP decom-

position of A in time ‚.n 3 /. Since we can determine the LUP decomposition of A

in time ‚.n 3 /, we can compute the inverse A 1

of a matrix A in time ‚.n 3 /.

Matrix multiplication and matrix inversion

We now show that the theoretical speedups obtained for matrix multiplication

translate to speedups for matrix inversion. In fact, we prove something stronger:

matrix inversion is equivalent to matrix multiplication, in the following sense.

If M.n/ denotes the time to multiply two n n matrices, then we can invert a

nonsingular n n matrix in time O.M.n//. Moreover, if I.n/ denotes the time

to invert a nonsingular n n matrix, then we can multiply two n n matrices in

time O.I.n//. We prove these results as two separate theorems.

Theorem 28.1 (Multiplication is no harder than inversion)

If we can invert an n n matrix in time I.n/, where I.n/ D .n 2 / and I.n/

satisﬁes the regularity condition I.3n/ D O.I.n//, then we can multiply two n n

matrices in time O.I.n//.

Proof Let A and B be n n matrices whose matrix product C we wish to com-

pute. We deﬁne the 3n 3n matrix D by

28.2 Inverting matrices 829

D D

I n A 0

0 I n B

0 0 I n

:

The inverse of D is

D

1

D

I n A AB

0 I n B

0 0 I n

;

and thus we can compute the product AB by taking the upper right n n submatrix

of D 1

.

We can construct matrix D in ‚.n 2 / time, which is O.I.n// because we assume

that I.n/ D .n 2 /, and we can invert D in O.I.3n// D O.I.n// time, by the

regularity condition on I.n/. We thus have M.n/ D O.I.n//.

Note that I.n/ satisﬁes the regularity condition whenever I.n/ D ‚.n c

lg

d

n/

for any constants c > 0 and d 0.

The proof that matrix inversion is no harder than matrix multiplication relies

on some properties of symmetric positive-deﬁnite matrices that we will prove in

Section 28.3.

Theorem 28.2 (Inversion is no harder than multiplication)

Suppose we can multiply two n n real matrices in time M.n/, where M.n/ D

.n 2 / and M.n/ satisﬁes the two regularity conditions M.n C k/ D O.M.n// for

any k in the range 0 k n and M.n=2/ cM.n/ for some constant c < 1=2.

Then we can compute the inverse of any real nonsingular n n matrix in time

O.M.n//.

Proof We prove the theorem here for real matrices. Exercise 28.2-6 asks you to

generalize the proof for matrices whose entries are complex numbers.

We can assume that n is an exact power of 2, since we have

A 0

0 I k

1

D

A 1 0

0 I k

for any k > 0. Thus, by choosing k such that n C k is a power of 2, we enlarge

the matrix to a size that is the next power of 2 and obtain the desired answer A 1

from the answer to the enlarged problem. The ﬁrst regularity condition on M.n/

ensures that this enlargement does not cause the running time to increase by more

than a constant factor.

For the moment, let us assume that the n n matrix A is symmetric and positive-

deﬁnite. We partition each of A and its inverse A 1

into four n=2 n=2 submatri-

ces:

830 Chapter 28 Matrix Operations

A D

B C T

C D

and A

1

D

R T

U V

: (28.11)

Then, if we let

S D D CB

1

C T

(28.12)

be the Schur complement of A with respect to B (we shall see more about this form

of Schur complement in Section 28.3), we have

A

1

D

R T

U V

D

B 1 C B 1 C T S 1 CB 1 B 1 C T S 1

S 1 CB 1 S 1

; (28.13)

since AA 1 D I n , as you can verify by performing the matrix multiplication. Be-

cause A is symmetric and positive-deﬁnite, Lemmas 28.4 and 28.5 in Section 28.3

imply that B and S are both symmetric and positive-deﬁnite. By Lemma 28.3 in

Section 28.3, therefore, the inverses B 1

and S 1

exist, and by Exercise D.2-6,

B 1

and S 1

are symmetric, so that .B 1 / T D B 1

and .S 1 / T D S 1

. There-

fore, we can compute the submatrices R, T , U , and V of A 1

as follows, where

all matrices mentioned are n=2 n=2:

1. Form the submatrices B, C , C T , and D of A.

2. Recursively compute the inverse B 1

of B.

3. Compute the matrix product W D CB 1

, and then compute its transpose W T ,

which equals B 1 C T (by Exercise D.1-2 and .B 1 / T D B 1

).

4. Compute the matrix product X D W C T , which equals CB 1 C T , and then

compute the matrix S D D X D D CB 1 C T .

5. Recursively compute the inverse S 1

of S, and set V to S 1

.

6. Compute the matrix product Y D S 1 W , which equals S 1 CB 1

, and

then compute its transpose Y T , which equals B 1 C T S 1

(by Exercise D.1-2,

.B 1 / T D B 1

, and .S 1 / T D S 1

). Set T to Y T and U to Y .

7. Compute the matrix product Z D W T Y , which equals B 1 C T S 1 CB 1

, and

set R to B 1 C Z.

Thus, we can invert an n n symmetric positive-deﬁnite matrix by inverting two

n=2 n=2 matrices in steps 2 and 5; performing four multiplications of n=2 n=2

matrices in steps 3, 4, 6, and 7; plus an additional cost of O.n 2 / for extracting

submatrices from A, inserting submatrices into A 1

, and performing a constant

number of additions, subtractions, and transposes on n=2 n=2 matrices. We get

the recurrence

I.n/ 2I.n=2/ C 4M.n=2/ C O.n

2

/

D 2I.n=2/ C ‚.M.n//

D O.M.n// :

28.2 Inverting matrices 831

The second line holds because the second regularity condition in the statement

of the theorem implies that 4M.n=2/ < 2M.n/ and because we assume that

M.n/ D .n 2 /. The third line follows because the second regularity condition

allows us to apply case 3 of the master theorem (Theorem 4.1).

It remains to prove that we can obtain the same asymptotic running time for ma-

trix multiplication as for matrix inversion when A is invertible but not symmetric

and positive-deﬁnite. The basic idea is that for any nonsingular matrix A, the ma-

trix A T A is symmetric (by Exercise D.1-2) and positive-deﬁnite (by Theorem D.6).

The trick, then, is to reduce the problem of inverting A to the problem of invert-

ing A T A.

The reduction is based on the observation that when A is an n n nonsingular

matrix, we have

A

1

D .A T A/

1

A T ;

since ..A T A/ 1 A T /A D .A T A/ 1 .A T A/ D I n and a matrix inverse is unique.

Therefore, we can compute A 1

by ﬁrst multiplying A T by A to obtain A T A, then

inverting the symmetric positive-deﬁnite matrix A T A using the above divide-and-

conquer algorithm, and ﬁnally multiplying the result by A T . Each of these three

steps takes O.M.n// time, and thus we can invert any nonsingular matrix with real

entries in O.M.n// time.

The proof of Theorem 28.2 suggests a means of solving the equation Ax D b

by using LU decomposition without pivoting, so long as A is nonsingular. We

multiply both sides of the equation by A T , yielding .A T A/x D A T b. This trans-

formation doesn’t affect the solution x, since A T is invertible, and so we can fac-

tor the symmetric positive-deﬁnite matrix A T A by computing an LU decomposi-

tion. We then use forward and back substitution to solve for x with the right-hand

side A T b. Although this method is theoretically correct, in practice the procedure

LUP-DECOMPOSITION works much better. LUP decomposition requires fewer

arithmetic operations by a constant factor, and it has somewhat better numerical

properties.

Exercises

28.2-1

Let M.n/ be the time to multiply two n n matrices, and let S.n/ denote the time

required to square an n n matrix. Show that multiplying and squaring matri-

ces have essentially the same difﬁculty: an M.n/-time matrix-multiplication al-

gorithm implies an O.M.n//-time squaring algorithm, and an S.n/-time squaring

algorithm implies an O.S.n//-time matrix-multiplication algorithm.

832 Chapter 28 Matrix Operations

28.2-2

Let M.n/ be the time to multiply two n n matrices, and let L.n/ be the time to

compute the LUP decomposition of an n n matrix. Show that multiplying matri-

ces and computing LUP decompositions of matrices have essentially the same dif-

ﬁculty: an M.n/-time matrix-multiplication algorithm implies an O.M.n//-time

LUP-decomposition algorithm, and an L.n/-time LUP-decomposition algorithm

implies an O.L.n//-time matrix-multiplication algorithm.

28.2-3

Let M.n/ be the time to multiply two n n matrices, and let D.n/ denote the

time required to ﬁnd the determinant of an n n matrix. Show that multiply-

ing matrices and computing the determinant have essentially the same difﬁculty:

an M.n/-time matrix-multiplication algorithm implies an O.M.n//-time determi-

nant algorithm, and a D.n/-time determinant algorithm implies an O.D.n//-time

matrix-multiplication algorithm.

28.2-4

Let M.n/ be the time to multiply two n n boolean matrices, and let T .n/ be the

time to ﬁnd the transitive closure of an n n boolean matrix. (See Section 25.2.)

Show that an M.n/-time boolean matrix-multiplication algorithm implies an

O.M.n/ lg n/-time transitive-closure algorithm, and a T .n/-time transitive-closure

algorithm implies an O.T .n//-time boolean matrix-multiplication algorithm.

28.2-5

Does the matrix-inversion algorithm based on Theorem 28.2 work when matrix

elements are drawn from the ﬁeld of integers modulo 2? Explain.

28.2-6 ?

Generalize the matrix-inversion algorithm of Theorem 28.2 to handle matrices of

complex numbers, and prove that your generalization works correctly. (Hint: In-

stead of the transpose of A, use the conjugate transpose A

, which you obtain from

the transpose of A by replacing every entry with its complex conjugate. Instead of

symmetric matrices, consider Hermitian matrices, which are matrices A such that

A D A

.)

28.3 Symmetric positive-deﬁnite matrices and least-squares approximation

Symmetric positive-deﬁnite matrices have many interesting and desirable proper-

ties. For example, they are nonsingular, and we can perform LU decomposition

on them without having to worry about dividing by 0. In this section, we shall

28.3 Symmetric positive-deﬁnite matrices and least-squares approximation 833

prove several other important properties of symmetric positive-deﬁnite matrices

and show an interesting application to curve ﬁtting by a least-squares approxima-

tion.

The ﬁrst property we prove is perhaps the most basic.

Lemma 28.3

Any positive-deﬁnite matrix is nonsingular.

Proof Suppose that a matrix A is singular. Then by Corollary D.3, there exists a

nonzero vector x such that Ax D 0. Hence, x T Ax D 0, and A cannot be positive-

deﬁnite.

The proof that we can perform LU decomposition on a symmetric positive-

deﬁnite matrix A without dividing by 0 is more involved. We begin by proving

properties about certain submatrices of A. Deﬁne the kth leading submatrix of A

to be the matrix A k consisting of the intersection of the ﬁrst k rows and ﬁrst k

columns of A.

Lemma 28.4

If A is a symmetric positive-deﬁnite matrix, then every leading submatrix of A is

symmetric and positive-deﬁnite.

Proof That each leading submatrix A k is symmetric is obvious. To prove that A k

is positive-deﬁnite, we assume that it is not and derive a contradiction. If A k is not

positive-deﬁnite, then there exists a k-vector x k ¤ 0 such that x T

k

A k x k 0. Let A

be n n, and

A D

A k B T

B C

(28.14)

for submatrices B (which is .nk/ k) and C (which is .nk/ .nk/). Deﬁne

the n-vector x D . x T

k

0 / T , where n k 0s follow x k . Then we have

x T Ax D . x T

k

0 /

A k B T

B C

x k

0

D . x T

k

0 /

A k x k

Bx k

D x T

k

A k x k

0 ;

which contradicts A being positive-deﬁnite.

834 Chapter 28 Matrix Operations

We now turn to some essential properties of the Schur complement. Let A be

a symmetric positive-deﬁnite matrix, and let A k be a leading k k submatrix

of A. Partition A once again according to equation (28.14). We generalize equa-

tion (28.9) to deﬁne the Schur complement S of A with respect to A k as

S D C BA

1

k

B T : (28.15)

(By Lemma 28.4, A k is symmetric and positive-deﬁnite; therefore, A 1

k

exists by

Lemma 28.3, and S is well deﬁned.) Note that our earlier deﬁnition (28.9) of the

Schur complement is consistent with equation (28.15), by letting k D 1.

The next lemma shows that the Schur-complement matrices of symmetric posi-

tive-deﬁnite matrices are themselves symmetric and positive-deﬁnite. We used this

result in Theorem 28.2, and we need its corollary to prove the correctness of LU

decomposition for symmetric positive-deﬁnite matrices.

Lemma 28.5 (Schur complement lemma)

If A is a symmetric positive-deﬁnite matrix and A k is a leading k k submatrix

of A, then the Schur complement S of A with respect to A k is symmetric and

positive-deﬁnite.

Proof Because A is symmetric, so is the submatrix C . By Exercise D.2-6, the

product BA 1

k

B T is symmetric, and by Exercise D.1-1, S is symmetric.

It remains to show that S is positive-deﬁnite. Consider the partition of A given in

equation (28.14). For any nonzero vector x, we have x T Ax > 0 by the assumption

that A is positive-deﬁnite. Let us break x into two subvectors y and ´ compatible

with A k and C , respectively. Because A 1

k

exists, we have

x T Ax D . y T ´ T /

A k B T

B C

y

´

D . y T ´ T /

A k y C B T ´

By C C ´

D y T A k y C y T B T ´ C ´ T By C ´ T C ´

D .y C A

1

k

B T ´/ T A k .y C A

1

k

B T ´/ C ´ T .C BA

1

k

B T /´ ; (28.16)

by matrix magic. (Verify by multiplying through.) This last equation amounts to

“completing the square” of the quadratic form. (See Exercise 28.3-2.)

Since x T Ax > 0 holds for any nonzero x, let us pick any nonzero ´ and then

choose y D A 1

k

B T ´, which causes the ﬁrst term in equation (28.16) to vanish,

leaving

´ T .C BA

1

k

B T /´ D ´ T S´

as the value of the expression. For any ´ ¤ 0, we therefore have ´ T S´ D

x T Ax > 0, and thus S is positive-deﬁnite.

28.3 Symmetric positive-deﬁnite matrices and least-squares approximation 835

Corollary 28.6

LU decomposition of a symmetric positive-deﬁnite matrix never causes a division

by 0.

Proof Let A be a symmetric positive-deﬁnite matrix. We shall prove something

stronger than the statement of the corollary: every pivot is strictly positive. The ﬁrst

pivot is a 11 . Let e 1 be the ﬁrst unit vector, from which we obtain a 11 D e T

1

Ae 1 > 0.

Since the ﬁrst step of LU decomposition produces the Schur complement of A

with respect to A 1 D .a 11 /, Lemma 28.5 implies by induction that all pivots are

positive.

Least-squares approximation

One important application of symmetric positive-deﬁnite matrices arises in ﬁtting

curves to given sets of data points. Suppose that we are given a set of m data points

.x 1 ; y 1 /; .x 2 ; y 2 /; : : : ; .x m ; y m / ;

where we know that the y i are subject to measurement errors. We would like to

determine a function F.x/ such that the approximation errors

i D F.x i / y i (28.17)

are small for i D 1; 2; : : : ; m. The form of the function F depends on the problem

at hand. Here, we assume that it has the form of a linearly weighted sum,

F.x/ D

n X

j D1

c j f j .x/ ;

where the number of summands n and the speciﬁc basis functions f j are chosen

based on knowledge of the problem at hand. A common choice is f j .x/ D x j 1

,

which means that

F.x/ D c 1 C c 2 x C c 3 x

2

C C c n x

n1

is a polynomial of degree n 1 in x. Thus, given m data points .x 1 ; y 1 /; .x 2 ; y 2 /;

: : : ; .x m ; y m /, we wish to calculate n coefﬁcients c 1 ; c 2 ; : : : ; c n that minimize the

approximation errors 1 ; 2 ; : : : ; m .

By choosing n D m, we can calculate each y i exactly in equation (28.17). Such

a high-degree F “ﬁts the noise” as well as the data, however, and generally gives

poor results when used to predict y for previously unseen values of x. It is usu-

ally better to choose n signiﬁcantly smaller than m and hope that by choosing the

coefﬁcients c j well, we can obtain a function F that ﬁnds the signiﬁcant patterns

in the data points without paying undue attention to the noise. Some theoretical

836 Chapter 28 Matrix Operations

principles exist for choosing n, but they are beyond the scope of this text. In any

case, once we choose a value of n that is less than m, we end up with an overde-

termined set of equations whose solution we wish to approximate. We now show

how to do so.

Let

A D

˙

f 1 .x 1 / f 2 .x 1 / : : : f n .x 1 /

f 1 .x 2 / f 2 .x 2 / : : : f n .x 2 /

:

:

:

:

:

:

:

:

:

:

:

:

f 1 .x m / f 2 .x m / : : : f n .x m /

denote the matrix of values of the basis functions at the given points; that is,

a ij D f j .x i /. Let c D .c k / denote the desired n-vector of coefﬁcients. Then,

Ac D

˙

f 1 .x 1 / f 2 .x 1 / : : : f n .x 1 /

f 1 .x 2 / f 2 .x 2 / : : : f n .x 2 /

:

:

:

:

:

:

:

:

:

:

:

:

f 1 .x m / f 2 .x m / : : : f n .x m /

˙

c 1

c 2

:

:

:

c n

D

˙

F.x 1 /

F.x 2 /

:

:

:

F.x m /

is the m-vector of “predicted values” for y. Thus,

D Ac y

is the m-vector of approximation errors.

To minimize approximation errors, we choose to minimize the norm of the error

vector , which gives us a least-squares solution, since

kk D

m X

iD1

2

i

! 1=2

:

Because

kk

2

D kAc yk

2

D

m X

iD1

n X

j D1

a ij c j y i

! 2

;

we can minimize kk by differentiating kk

2

with respect to each c k and then

setting the result to 0:

28.3 Symmetric positive-deﬁnite matrices and least-squares approximation 837

d kk

2

dc k

D

m X

iD1

2

n X

j D1

a ij c j y i

!

a ik D 0 : (28.18)

The n equations (28.18) for k D 1; 2; : : : ; n are equivalent to the single matrix

equation

.Ac y/ T A D 0

or, equivalently (using Exercise D.1-2), to

A T .Ac y/ D 0 ;

which implies

A T Ac D A T y : (28.19)

In statistics, this is called the normal equation. The matrix A T A is symmetric

by Exercise D.1-2, and if A has full column rank, then by Theorem D.6, A T A

is positive-deﬁnite as well. Hence, .A T A/ 1

exists, and the solution to equa-

tion (28.19) is

c D

.A T A/

1

A T

y

D A

C

y ; (28.20)

where the matrix A C D ..A T A/ 1 A T / is the pseudoinverse of the matrix A. The

pseudoinverse naturally generalizes the notion of a matrix inverse to the case in

which A is not square. (Compare equation (28.20) as the approximate solution to

Ac D y with the solution A 1 b as the exact solution to Ax D b.)

As an example of producing a least-squares ﬁt, suppose that we have ﬁve data

points

.x 1 ; y 1 / D .1; 2/ ;

.x 2 ; y 2 / D .1; 1/ ;

.x 3 ; y 3 / D .2; 1/ ;

.x 4 ; y 4 / D .3; 0/ ;

.x 5 ; y 5 / D .5; 3/ ;

shown as black dots in Figure 28.3. We wish to ﬁt these points with a quadratic

polynomial

F.x/ D c 1 C c 2 x C c 3 x

2

:

We start with the matrix of basis-function values

838 Chapter 28 Matrix Operations

0.5

1.0

1.5

2.0

2.5

3.0

0.0

1 2 3 4 5 0 –1 –2

x

y

F(x) = 1.2 – 0.757x + 0.214x2

Figure 28.3 The least-squares ﬁt of a quadratic polynomial to the set of ﬁve data points

f.1; 2/; .1; 1/; .2; 1/; .3; 0/; .5; 3/g. The black dots are the data points, and the white dots are their

estimated values predicted by the polynomial F.x/ D 1:2 0:757x C 0:214x

2

, the quadratic poly-

nomial that minimizes the sum of the squared errors. Each shaded line shows the error for one data

point.

A D

1 x 1 x 2

1

1 x 2 x 2

2

1 x 3 x 2

3

1 x 4 x 2

4

1 x 5 x 2

5

D

1 1 1

1 1 1

1 2 4

1 3 9

1 5 25

;

whose pseudoinverse is

A

C

D

0:500 0:300 0:200 0:100 0:100

0:388 0:093 0:190 0:193 0:088

0:060 0:036 0:048 0:036 0:060

:

Multiplying y by A C

, we obtain the coefﬁcient vector

c D

1:200

0:757

0:214

;

which corresponds to the quadratic polynomial

28.3 Symmetric positive-deﬁnite matrices and least-squares approximation 839

F.x/ D 1:200 0:757x C 0:214x

2

as the closest-ﬁtting quadratic to the given data, in a least-squares sense.

As a practical matter, we solve the normal equation (28.19) by multiplying y

by A T and then ﬁnding an LU decomposition of A T A. If A has full rank, the

matrix A T A is guaranteed to be nonsingular, because it is symmetric and positive-

deﬁnite. (See Exercise D.1-2 and Theorem D.6.)

Exercises

28.3-1

Prove that every diagonal element of a symmetric positive-deﬁnite matrix is posi-

tive.

28.3-2

Let A D

a b

b c

be a 2 2 symmetric positive-deﬁnite matrix. Prove that its

determinant ac b 2

is positive by “completing the square” in a manner similar to

that used in the proof of Lemma 28.5.

28.3-3

Prove that the maximum element in a symmetric positive-deﬁnite matrix lies on

the diagonal.

28.3-4

Prove that the determinant of each leading submatrix of a symmetric positive-

deﬁnite matrix is positive.

28.3-5

Let A k denote the kth leading submatrix of a symmetric positive-deﬁnite matrix A.

Prove that det.A k /= det.A k1 / is the kth pivot during LU decomposition, where,

by convention, det.A 0 / D 1.

28.3-6

Find the function of the form

F.x/ D c 1 C c 2 x lg x C c 3 e

x

that is the best least-squares ﬁt to the data points

.1; 1/; .2; 1/; .3; 3/; .4; 8/ :

840 Chapter 28 Matrix Operations

28.3-7

Show that the pseudoinverse A C

satisﬁes the following four equations:

AA

C

A D A ;

A

C

AA

C

D A

C

;

.AA

C

/ T D AA

C

;

.A

C

A/ T D A

C

A :

Problems

28-1 Tridiagonal systems of linear equations

Consider the tridiagonal matrix

A D

ˇ

1 1 0 0 0

1 2 1 0 0

0 1 2 1 0

0 0 1 2 1

0 0 0 1 2

:

a. Find an LU decomposition of A.

b. Solve the equation Ax D

1 1 1 1 1

T

by using forward and back sub-

stitution.

c. Find the inverse of A.

d. Show how, for any n n symmetric positive-deﬁnite, tridiagonal matrix A and

any n-vector b, to solve the equation Ax D b in O.n/ time by performing an

LU decomposition. Argue that any method based on forming A 1

is asymptot-

ically more expensive in the worst case.

e. Show how, for any n n nonsingular, tridiagonal matrix A and any n-vector b, to

solve the equation Ax D b in O.n/ time by performing an LUP decomposition.

28-2 Splines

A practical method for interpolating a set of points with a curve is to use cu-

bic splines. We are given a set f.x i ; y i / W i D 0; 1; : : : ; ng of n C 1 point-value

pairs, where x 0 < x 1 < < x n . We wish to ﬁt a piecewise-cubic curve

(spline) f .x/ to the points. That is, the curve f .x/ is made up of n cubic polyno-

mials f i .x/ D a i C b i x C c i x 2 C d i x 3

for i D 0; 1; : : : ; n 1, where if x falls in

Problems for Chapter 28 841

the range x i x x iC1 , then the value of the curve is given by f .x/ D f i .xx i /.

The points x i at which the cubic polynomials are “pasted” together are called knots.

For simplicity, we shall assume that x i D i for i D 0; 1; : : : ; n.

To ensure continuity of f .x/, we require that

f .x i / D f i .0/ D y i ;

f .x iC1 / D f i .1/ D y iC1

for i D 0; 1; : : : ; n 1. To ensure that f .x/ is sufﬁciently smooth, we also insist

that the ﬁrst derivative be continuous at each knot:

f

0

.x iC1 / D f

0

i

.1/ D f

0

iC1

.0/

for i D 0; 1; : : : ; n 2.

a. Suppose that for i D 0; 1; : : : ; n, we are given not only the point-value pairs

f.x i ; y i /g but also the ﬁrst derivatives D i D f 0 .x i / at each knot. Express each

coefﬁcient a i , b i , c i , and d i in terms of the values y i , y iC1 , D i , and D iC1 .

(Remember that x i D i.) How quickly can we compute the 4n coefﬁcients

from the point-value pairs and ﬁrst derivatives?

The question remains of how to choose the ﬁrst derivatives of f .x/ at the knots.

One method is to require the second derivatives to be continuous at the knots:

f

00

.x iC1 / D f

00

i

.1/ D f

00

iC1

.0/

for i D 0; 1; : : : ; n 2. At the ﬁrst and last knots, we assume that f 00 .x 0 / D

f 00

0

.0/ D 0 and f 00 .x n / D f 00

n1

.1/ D 0; these assumptions make f .x/ a natural

cubic spline.

b. Use the continuity constraints on the second derivative to show that for i D

1; 2; : : : ; n 1,

D i1 C 4D i C D iC1 D 3.y iC1 y i1 / : (28.21)

c. Show that

2D 0 C D 1 D 3.y 1 y 0 / ; (28.22)

D n1 C 2D n D 3.y n y n1 / : (28.23)

d. Rewrite equations (28.21)–(28.23) as a matrix equation involving the vector

D D hD 0 ; D 1 ; : : : ; D n i of unknowns. What attributes does the matrix in your

equation have?

e. Argue that a natural cubic spline can interpolate a set of n C 1 point-value pairs

in O.n/ time (see Problem 28-1).

842 Chapter 28 Matrix Operations

f. Show how to determine a natural cubic spline that interpolates a set of n C 1

points .x i ; y i / satisfying x 0 < x 1 < < x n , even when x i is not necessarily

equal to i. What matrix equation must your method solve, and how quickly

does your algorithm run?

Chapter notes

Many excellent texts describe numerical and scientiﬁc computation in much greater

detail than we have room for here. The following are especially readable: George

and Liu [132], Golub and Van Loan [144], Press, Teukolsky, Vetterling, and Flan-

nery [283, 284], and Strang [323, 324].

Golub and Van Loan [144] discuss numerical stability. They show why det.A/

is not necessarily a good indicator of the stability of a matrix A, proposing instead

to use kAk 1 kA 1 k 1

, where kAk 1

D max 1in

P n

j D1

ja ij j. They also address

the question of how to compute this value without actually computing A 1

.

Gaussian elimination, upon which the LU and LUP decompositions are based,

was the ﬁrst systematic method for solving linear systems of equations. It was also

one of the earliest numerical algorithms. Although it was known earlier, its dis-

covery is commonly attributed to C. F. Gauss (1777–1855). In his famous paper

[325], Strassen showed that an n n matrix can be inverted in O.n lg 7 / time. Wino-

grad [358] originally proved that matrix multiplication is no harder than matrix

inversion, and the converse is due to Aho, Hopcroft, and Ullman [5].

Another important matrix decomposition is the singular value decomposition,

or SVD. The SVD factors an m n matrix A into A D Q 1 †Q T

2

, where † is an

m n matrix with nonzero values only on the diagonal, Q 1 is m m with mutually

orthonormal columns, and Q 2 is n n, also with mutually orthonormal columns.

Two vectors are orthonormal if their inner product is 0 and each vector has a norm

of 1. The books by Strang [323, 324] and Golub and Van Loan [144] contain good

treatments of the SVD.

Strang [324] has an excellent presentation of symmetric positive-deﬁnite matri-

ces and of linear algebra in general.

29 Linear Programming

Many problems take the form of maximizing or minimizing an objective, given

limited resources and competing constraints. If we can specify the objective as

a linear function of certain variables, and if we can specify the constraints on

resources as equalities or inequalities on those variables, then we have a linear-

programming problem. Linear programs arise in a variety of practical applica-

tions. We begin by studying an application in electoral politics.

A political problem

Suppose that you are a politician trying to win an election. Your district has three

different types of areas—urban, suburban, and rural. These areas have, respec-

tively, 100,000, 200,000, and 50,000 registered voters. Although not all the reg-

istered voters actually go to the polls, you decide that to govern effectively, you

would like at least half the registered voters in each of the three regions to vote for

you. You are honorable and would never consider supporting policies in which you

do not believe. You realize, however, that certain issues may be more effective in

winning votes in certain places. Your primary issues are building more roads, gun

control, farm subsidies, and a gasoline tax dedicated to improved public transit.

According to your campaign staff’s research, you can estimate how many votes

you win or lose from each population segment by spending $1,000 on advertising

on each issue. This information appears in the table of Figure 29.1. In this table,

each entry indicates the number of thousands of either urban, suburban, or rural

voters who would be won over by spending $1,000 on advertising in support of a

particular issue. Negative entries denote votes that would be lost. Your task is to

ﬁgure out the minimum amount of money that you need to spend in order to win

50,000 urban votes, 100,000 suburban votes, and 25,000 rural votes.

You could, by trial and error, devise a strategy that wins the required number

of votes, but the strategy you come up with might not be the least expensive one.

For example, you could devote $20,000 of advertising to building roads, $0 to gun

control, $4,000 to farm subsidies, and $9,000 to a gasoline tax. In this case, you

844 Chapter 29 Linear Programming

policy urban suburban rural

build roads 2 5 3

gun control 8 2 5

farm subsidies 0 0 10

gasoline tax 10 0 2

Figure 29.1 The effects of policies on voters. Each entry describes the number of thousands of

urban, suburban, or rural voters who could be won over by spending $1,000 on advertising support

of a policy on a particular issue. Negative entries denote votes that would be lost.

would win 20.2/C0.8/C4.0/C9.10/ D 50 thousand urban votes, 20.5/C0.2/C

4.0/C9.0/ D 100 thousand suburban votes, and 20.3/C0.5/C4.10/C9.2/ D

82 thousand rural votes. You would win the exact number of votes desired in the

urban and suburban areas and more than enough votes in the rural area. (In fact,

in the rural area, you would receive more votes than there are voters.) In order to

garner these votes, you would have paid for 20 C 0 C 4 C 9 D 33 thousand dollars

of advertising.

Naturally, you may wonder whether this strategy is the best possible. That is,

could you achieve your goals while spending less on advertising? Additional trial

and error might help you to answer this question, but wouldn’t you rather have a

systematic method for answering such questions? In order to develop one, we shall

formulate this question mathematically. We introduce 4 variables:

x 1 is the number of thousands of dollars spent on advertising on building roads,

x 2 is the number of thousands of dollars spent on advertising on gun control,

x 3 is the number of thousands of dollars spent on advertising on farm subsidies,

and

x 4 is the number of thousands of dollars spent on advertising on a gasoline tax.

We can write the requirement that we win at least 50,000 urban votes as

2x 1 C 8x 2 C 0x 3 C 10x 4 50 : (29.1)

Similarly, we can write the requirements that we win at least 100,000 suburban

votes and 25,000 rural votes as

5x 1 C 2x 2 C 0x 3 C 0x 4 100 (29.2)

and

3x 1 5x 2 C 10x 3 2x 4 25 : (29.3)

Any setting of the variables x 1 ; x 2 ; x 3 ; x 4 that satisﬁes inequalities (29.1)–(29.3)

yields a strategy that wins a sufﬁcient number of each type of vote. In order to

Chapter 29 Linear Programming 845

keep costs as small as possible, you would like to minimize the amount spent on

advertising. That is, you want to minimize the expression

x 1 C x 2 C x 3 C x 4 : (29.4)

Although negative advertising often occurs in political campaigns, there is no such

thing as negative-cost advertising. Consequently, we require that

x 1 0; x 2 0; x 3 0; and x 4 0 : (29.5)

Combining inequalities (29.1)–(29.3) and (29.5) with the objective of minimiz-

ing (29.4), we obtain what is known as a “linear program.” We format this problem

as

minimize x 1 C x 2 C x 3 C x 4 (29.6)

subject to

2x 1 C 8x 2 C 0x 3 C 10x 4 50 (29.7)

5x 1 C 2x 2 C 0x 3 C 0x 4 100 (29.8)

3x 1 5x 2 C 10x 3 2x 4 25 (29.9)

x 1 ; x 2 ; x 3 ; x 4 0 : (29.10)

The solution of this linear program yields your optimal strategy.

General linear programs

In the general linear-programming problem, we wish to optimize a linear function

subject to a set of linear inequalities. Given a set of real numbers a 1 ; a 2 ; : : : ; a n and

a set of variables x 1 ; x 2 ; : : : ; x n , we deﬁne a linear function f on those variables

by

f .x 1 ; x 2 ; : : : ; x n / D a 1 x 1 C a 2 x 2 C C a n x n D

n X

j D1

a j x j :

If b is a real number and f is a linear function, then the equation

f .x 1 ; x 2 ; : : : ; x n / D b

is a linear equality and the inequalities

f .x 1 ; x 2 ; : : : ; x n / b

and

f .x 1 ; x 2 ; : : : ; x n / b

846 Chapter 29 Linear Programming

are linear inequalities. We use the general term linear constraints to denote either

linear equalities or linear inequalities. In linear programming, we do not allow

strict inequalities. Formally, a linear-programming problem is the problem of

either minimizing or maximizing a linear function subject to a ﬁnite set of linear

constraints. If we are to minimize, then we call the linear program a minimization

linear program, and if we are to maximize, then we call the linear program a

maximization linear program.

The remainder of this chapter covers how to formulate and solve linear pro-

grams. Although several polynomial-time algorithms for linear programming have

been developed, we will not study them in this chapter. Instead, we shall study the

simplex algorithm, which is the oldest linear-programming algorithm. The simplex

algorithm does not run in polynomial time in the worst case, but it is fairly efﬁcient

and widely used in practice.

An overview of linear programming

In order to describe properties of and algorithms for linear programs, we ﬁnd it

convenient to express them in canonical forms. We shall use two forms, standard

and slack, in this chapter. We will deﬁne them precisely in Section 29.1. Infor-

mally, a linear program in standard form is the maximization of a linear function

subject to linear inequalities, whereas a linear program in slack form is the max-

imization of a linear function subject to linear equalities. We shall typically use

standard form for expressing linear programs, but we ﬁnd it more convenient to

use slack form when we describe the details of the simplex algorithm. For now, we

restrict our attention to maximizing a linear function on n variables subject to a set

of m linear inequalities.

Let us ﬁrst consider the following linear program with two variables:

maximize x 1 C x 2 (29.11)

subject to

4x 1 x 2 8 (29.12)

2x 1 C x 2 10 (29.13)

5x 1 2x 2 2 (29.14)

x 1 ; x 2 0 : (29.15)

We call any setting of the variables x 1 and x 2 that satisﬁes all the constraints

(29.12)–(29.15) a feasible solution to the linear program. If we graph the con-

straints in the .x 1 ; x 2 /-Cartesian coordinate system, as in Figure 29.2(a), we see

Chapter 29 Linear Programming 847

4x

1

– x

2 ≤ 8

2x

1

+ x

2 ≤ 10

x

2

x

1

x

2 ≥ 0

x

1 ≥ 0

5x

1

– 2x

2 ≥ –2

(a)

x

2

x

1

(b)

x

1

+ x

2

= 0

x

1

+ x

2

= 4

x

1

+ x

2

= 8

Figure 29.2 (a) The linear program given in (29.12)–(29.15). Each constraint is represented by

a line and a direction. The intersection of the constraints, which is the feasible region, is shaded.

(b) The dotted lines show, respectively, the points for which the objective value is 0, 4, and 8. The

optimal solution to the linear program is x1 D 2 and x2 D 6 with objective value 8.

that the set of feasible solutions (shaded in the ﬁgure) forms a convex region 1 in

the two-dimensional space. We call this convex region the feasible region and the

function we wish to maximize the objective function. Conceptually, we could eval-

uate the objective function x 1 C x 2 at each point in the feasible region; we call the

value of the objective function at a particular point the objective value. We could

then identify a point that has the maximum objective value as an optimal solution.

For this example (and for most linear programs), the feasible region contains an

inﬁnite number of points, and so we need to determine an efﬁcient way to ﬁnd a

point that achieves the maximum objective value without explicitly evaluating the

objective function at every point in the feasible region.

In two dimensions, we can optimize via a graphical procedure. The set of points

for which x 1 Cx 2 D ´, for any ´, is a line with a slope of 1. If we plot x 1 Cx 2 D 0,

we obtain the line with slope 1 through the origin, as in Figure 29.2(b). The

intersection of this line and the feasible region is the set of feasible solutions that

have an objective value of 0. In this case, that intersection of the line with the

feasible region is the single point .0; 0/. More generally, for any ´, the intersection

1

An intuitive deﬁnition of a convex region is that it fulﬁlls the requirement that for any two points in

the region, all points on a line segment between them are also in the region.

848 Chapter 29 Linear Programming

of the line x 1 C x 2 D ´ and the feasible region is the set of feasible solutions that

have objective value ´. Figure 29.2(b) shows the lines x 1 C x 2 D 0, x 1 C x 2 D 4,

and x 1 C x 2 D 8. Because the feasible region in Figure 29.2 is bounded, there

must be some maximum value ´ for which the intersection of the line x 1 C x 2 D ´

and the feasible region is nonempty. Any point at which this occurs is an optimal

solution to the linear program, which in this case is the point x 1 D 2 and x 2 D 6

with objective value 8.

It is no accident that an optimal solution to the linear program occurs at a vertex

of the feasible region. The maximum value of ´ for which the line x 1 C x 2 D ´

intersects the feasible region must be on the boundary of the feasible region, and

thus the intersection of this line with the boundary of the feasible region is either a

single vertex or a line segment. If the intersection is a single vertex, then there is

just one optimal solution, and it is that vertex. If the intersection is a line segment,

every point on that line segment must have the same objective value; in particular,

both endpoints of the line segment are optimal solutions. Since each endpoint of a

line segment is a vertex, there is an optimal solution at a vertex in this case as well.

Although we cannot easily graph linear programs with more than two variables,

the same intuition holds. If we have three variables, then each constraint corre-

sponds to a half-space in three-dimensional space. The intersection of these half-

spaces forms the feasible region. The set of points for which the objective function

obtains a given value ´ is now a plane (assuming no degenerate conditions). If all

coefﬁcients of the objective function are nonnegative, and if the origin is a feasible

solution to the linear program, then as we move this plane away from the origin, in

a direction normal to the objective function, we ﬁnd points of increasing objective

value. (If the origin is not feasible or if some coefﬁcients in the objective function

are negative, the intuitive picture becomes slightly more complicated.) As in two

dimensions, because the feasible region is convex, the set of points that achieve

the optimal objective value must include a vertex of the feasible region. Simi-

larly, if we have n variables, each constraint deﬁnes a half-space in n-dimensional

space. We call the feasible region formed by the intersection of these half-spaces a

simplex. The objective function is now a hyperplane and, because of convexity, an

optimal solution still occurs at a vertex of the simplex.

The simplex algorithm takes as input a linear program and returns an optimal

solution. It starts at some vertex of the simplex and performs a sequence of itera-

tions. In each iteration, it moves along an edge of the simplex from a current vertex

to a neighboring vertex whose objective value is no smaller than that of the current

vertex (and usually is larger.) The simplex algorithm terminates when it reaches

a local maximum, which is a vertex from which all neighboring vertices have a

smaller objective value. Because the feasible region is convex and the objective

function is linear, this local optimum is actually a global optimum. In Section 29.4,

Chapter 29 Linear Programming 849

we shall use a concept called “duality” to show that the solution returned by the

simplex algorithm is indeed optimal.

Although the geometric view gives a good intuitive view of the operations of the

simplex algorithm, we shall not refer to it explicitly when developing the details

of the simplex algorithm in Section 29.3. Instead, we take an algebraic view. We

ﬁrst write the given linear program in slack form, which is a set of linear equalities.

These linear equalities express some of the variables, called “basic variables,” in

terms of other variables, called “nonbasic variables.” We move from one vertex

to another by making a basic variable become nonbasic and making a nonbasic

variable become basic. We call this operation a “pivot” and, viewed algebraically,

it is nothing more than rewriting the linear program in an equivalent slack form.

The two-variable example described above was particularly simple. We shall

need to address several more details in this chapter. These issues include iden-

tifying linear programs that have no solutions, linear programs that have no ﬁnite

optimal solution, and linear programs for which the origin is not a feasible solution.

Applications of linear programming

Linear programming has a large number of applications. Any textbook on opera-

tions research is ﬁlled with examples of linear programming, and linear program-

ming has become a standard tool taught to students in most business schools. The

election scenario is one typical example. Two more examples of linear program-

ming are the following:

An airline wishes to schedule its ﬂight crews. The Federal Aviation Adminis-

tration imposes many constraints, such as limiting the number of consecutive

hours that each crew member can work and insisting that a particular crew work

only on one model of aircraft during each month. The airline wants to schedule

crews on all of its ﬂights using as few crew members as possible.

An oil company wants to decide where to drill for oil. Siting a drill at a particu-

lar location has an associated cost and, based on geological surveys, an expected

payoff of some number of barrels of oil. The company has a limited budget for

locating new drills and wants to maximize the amount of oil it expects to ﬁnd,

given this budget.

With linear programs, we also model and solve graph and combinatorial prob-

lems, such as those appearing in this textbook. We have already seen a special

case of linear programming used to solve systems of difference constraints in Sec-

tion 24.4. In Section 29.2, we shall study how to formulate several graph and

network-ﬂow problems as linear programs. In Section 35.4, we shall use linear

programming as a tool to ﬁnd an approximate solution to another graph problem.

850 Chapter 29 Linear Programming

Algorithms for linear programming

This chapter studies the simplex algorithm. This algorithm, when implemented

carefully, often solves general linear programs quickly in practice. With some

carefully contrived inputs, however, the simplex algorithm can require exponential

time. The ﬁrst polynomial-time algorithm for linear programming was the ellipsoid

algorithm, which runs slowly in practice. A second class of polynomial-time algo-

rithms are known as interior-point methods. In contrast to the simplex algorithm,

which moves along the exterior of the feasible region and maintains a feasible solu-

tion that is a vertex of the simplex at each iteration, these algorithms move through

the interior of the feasible region. The intermediate solutions, while feasible, are

not necessarily vertices of the simplex, but the ﬁnal solution is a vertex. For large

inputs, interior-point algorithms can run as fast as, and sometimes faster than, the

simplex algorithm. The chapter notes point you to more information about these

algorithms.

If we add to a linear program the additional requirement that all variables take

on integer values, we have an integer linear program. Exercise 34.5-3 asks you

to show that just ﬁnding a feasible solution to this problem is NP-hard; since

no polynomial-time algorithms are known for any NP-hard problems, there is no

known polynomial-time algorithm for integer linear programming. In contrast, we

can solve a general linear-programming problem in polynomial time.

In this chapter, if we have a linear program with variables x D .x 1 ; x 2 ; : : : ; x n /

and wish to refer to a particular setting of the variables, we shall use the notation

Nx D . Nx 1 ; Nx 2 ; : : : ; Nx n /.

29.1 Standard and slack forms

This section describes two formats, standard form and slack form, that are use-

ful when we specify and work with linear programs. In standard form, all the

constraints are inequalities, whereas in slack form, all constraints are equalities

(except for those that require the variables to be nonnegative).

Standard form

In standard form, we are given n real numbers c 1 ; c 2 ; : : : ; c n ; m real numbers

b 1 ; b 2 ; : : : ; b m ; and mn real numbers a ij for i D 1; 2; : : : ; m and j D 1; 2; : : : ; n.

We wish to ﬁnd n real numbers x 1 ; x 2 ; : : : ; x n that

29.1 Standard and slack forms 851

maximize

n X

j D1

c j x j (29.16)

subject to

n X

j D1

a ij x j b i for i D 1; 2; : : : ; m (29.17)

x j 0 for j D 1; 2; : : : ; n : (29.18)

Generalizing the terminology we introduced for the two-variable linear program,

we call expression (29.16) the objective function and the n C m inequalities in

lines (29.17) and (29.18) the constraints. The n constraints in line (29.18) are the

nonnegativity constraints. An arbitrary linear program need not have nonnegativ-

ity constraints, but standard form requires them. Sometimes we ﬁnd it convenient

to express a linear program in a more compact form. If we create an m n matrix

A D .a ij /, an m-vector b D .b i /, an n-vector c D .c j /, and an n-vector x D .x j /,

then we can rewrite the linear program deﬁned in (29.16)–(29.18) as

maximize c T x (29.19)

subject to

Ax b (29.20)

x 0 : (29.21)

In line (29.19), c T x is the inner product of two vectors. In inequality (29.20), Ax

is a matrix-vector product, and in inequality (29.21), x 0 means that each entry

of the vector x must be nonnegative. We see that we can specify a linear program

in standard form by a tuple .A; b; c/, and we shall adopt the convention that A, b,

and c always have the dimensions given above.

We now introduce terminology to describe solutions to linear programs. We used

some of this terminology in the earlier example of a two-variable linear program.

We call a setting of the variables Nx that satisﬁes all the constraints a feasible solu-

tion, whereas a setting of the variables Nx that fails to satisfy at least one constraint

is an infeasible solution. We say that a solution Nx has objective value c T Nx. A fea-

sible solution Nx whose objective value is maximum over all feasible solutions is an

optimal solution, and we call its objective value c T Nx the optimal objective value.

If a linear program has no feasible solutions, we say that the linear program is in-

feasible; otherwise it is feasible. If a linear program has some feasible solutions

but does not have a ﬁnite optimal objective value, we say that the linear program

is unbounded. Exercise 29.1-9 asks you to show that a linear program can have a

ﬁnite optimal objective value even if the feasible region is not bounded.

852 Chapter 29 Linear Programming

Converting linear programs into standard form

It is always possible to convert a linear program, given as minimizing or maxi-

mizing a linear function subject to linear constraints, into standard form. A linear

program might not be in standard form for any of four possible reasons:

1. The objective function might be a minimization rather than a maximization.

2. There might be variables without nonnegativity constraints.

3. There might be equality constraints, which have an equal sign rather than a

less-than-or-equal-to sign.

4. There might be inequality constraints, but instead of having a less-than-or-

equal-to sign, they have a greater-than-or-equal-to sign.

When converting one linear program L into another linear program L 0

, we would

like the property that an optimal solution to L 0

yields an optimal solution to L. To

capture this idea, we say that two maximization linear programs L and L 0

are

equivalent if for each feasible solution Nx to L with objective value ´, there is

a corresponding feasible solution Nx 0

to L 0

with objective value ´, and for each

feasible solution Nx 0

to L 0

with objective value ´, there is a corresponding feasible

solution Nx to L with objective value ´. (This deﬁnition does not imply a one-to-

one correspondence between feasible solutions.) A minimization linear program L

and a maximization linear program L 0

are equivalent if for each feasible solution Nx

to L with objective value ´, there is a corresponding feasible solution Nx 0

to L 0

with

objective value ´, and for each feasible solution Nx 0

to L 0

with objective value ´,

there is a corresponding feasible solution Nx to L with objective value ´.

We now show how to remove, one by one, each of the possible problems in the

list above. After removing each one, we shall argue that the new linear program is

equivalent to the old one.

To convert a minimization linear program L into an equivalent maximization lin-

ear program L 0

, we simply negate the coefﬁcients in the objective function. Since

L and L 0

have identical sets of feasible solutions and, for any feasible solution, the

objective value in L is the negative of the objective value in L 0

, these two linear

programs are equivalent. For example, if we have the linear program

minimize 2x 1 C 3x 2

subject to

x 1 C x 2 D 7

x 1 2x 2 4

x 1 0 ;

and we negate the coefﬁcients of the objective function, we obtain

29.1 Standard and slack forms 853

maximize 2x 1 3x 2

subject to

x 1 C x 2 D 7

x 1 2x 2 4

x 1 0 :

Next, we show how to convert a linear program in which some of the variables

do not have nonnegativity constraints into one in which each variable has a non-

negativity constraint. Suppose that some variable x j does not have a nonnegativity

constraint. Then, we replace each occurrence of x j by x 0

j

x 00

j

, and add the non-

negativity constraints x 0

j

0 and x 00

j

0. Thus, if the objective function has a

term c j x j , we replace it by c j x 0

j

c j x 00

j

, and if constraint i has a term a ij x j , we

replace it by a ij x 0

j

a ij x 00

j

. Any feasible solution yx to the new linear program cor-

responds to a feasible solution Nx to the original linear program with Nx j D yx 0

j

yx 00

j

and with the same objective value. Also, any feasible solution Nx to the original

linear program corresponds to a feasible solution yx to the new linear program with

yx 0

j

D Nx j and yx 00

j

D 0 if Nx j 0, or with yx 00

j

D Nx j and yx 0

j

D 0 if Nx j < 0. The two

linear programs have the same objective value regardless of the sign of Nx j . Thus,

the two linear programs are equivalent. We apply this conversion scheme to each

variable that does not have a nonnegativity constraint to yield an equivalent linear

program in which all variables have nonnegativity constraints.

Continuing the example, we want to ensure that each variable has a correspond-

ing nonnegativity constraint. Variable x 1 has such a constraint, but variable x 2 does

not. Therefore, we replace x 2 by two variables x 0

2

and x 00

2

, and we modify the linear

program to obtain

maximize 2x 1 3x

0

2

C 3x

00

2

subject to

x 1 C x

0

2

x

00

2

D 7 (29.22)

x 1 2x

0

2

C 2x

00

2

4

x 1 ; x

0

2

; x

00

2

0 :

Next, we convert equality constraints into inequality constraints. Suppose that a

linear program has an equality constraint f .x 1 ; x 2 ; : : : ; x n / D b. Since x D y if

and only if both x y and x y, we can replace this equality constraint by the

pair of inequality constraints f .x 1 ; x 2 ; : : : ; x n / b and f .x 1 ; x 2 ; : : : ; x n / b.

Repeating this conversion for each equality constraint yields a linear program in

which all constraints are inequalities.

Finally, we can convert the greater-than-or-equal-to constraints to less-than-or-

equal-to constraints by multiplying these constraints through by 1. That is, any

inequality of the form

854 Chapter 29 Linear Programming

n X

j D1

a ij x j b i

is equivalent to

n X

j D1

a ij x j b i :

Thus, by replacing each coefﬁcient a ij by a ij and each value b i by b i , we obtain

an equivalent less-than-or-equal-to constraint.

Finishing our example, we replace the equality in constraint (29.22) by two in-

equalities, obtaining

maximize 2x 1 3x

0

2

C 3x

00

2

subject to

x 1 C x

0

2

x

00

2

7

x 1 C x

0

2

x

00

2

7 (29.23)

x 1 2x

0

2

C 2x

00

2

4

x 1 ; x

0

2

; x

00

2

0 :

Finally, we negate constraint (29.23). For consistency in variable names, we re-

name x 0

2

to x 2 and x 00

2

to x 3 , obtaining the standard form

maximize 2x 1 3x 2 C 3x 3 (29.24)

subject to

x 1 C x 2 x 3 7 (29.25)

x 1 x 2 C x 3 7 (29.26)

x 1 2x 2 C 2x 3 4 (29.27)

x 1 ; x 2 ; x 3 0 : (29.28)

Converting linear programs into slack form

To efﬁciently solve a linear program with the simplex algorithm, we prefer to ex-

press it in a form in which some of the constraints are equality constraints. More

precisely, we shall convert it into a form in which the nonnegativity constraints are

the only inequality constraints, and the remaining constraints are equalities. Let

n X

j D1

a ij x j b i (29.29)

29.1 Standard and slack forms 855

be an inequality constraint. We introduce a new variable s and rewrite inequal-

ity (29.29) as the two constraints

s D b i

n X

j D1

a ij x j ; (29.30)

s 0 : (29.31)

We call s a slack variable because it measures the slack, or difference, between

the left-hand and right-hand sides of equation (29.29). (We shall soon see why we

ﬁnd it convenient to write the constraint with only the slack variable on the left-

hand side.) Because inequality (29.29) is true if and only if both equation (29.30)

and inequality (29.31) are true, we can convert each inequality constraint of a lin-

ear program in this way to obtain an equivalent linear program in which the only

inequality constraints are the nonnegativity constraints. When converting from

standard to slack form, we shall use x nCi (instead of s) to denote the slack variable

associated with the ith inequality. The ith constraint is therefore

x nCi D b i

n X

j D1

a ij x j ; (29.32)

along with the nonnegativity constraint x nCi 0.

By converting each constraint of a linear program in standard form, we obtain a

linear program in a different form. For example, for the linear program described

in (29.24)–(29.28), we introduce slack variables x 4 , x 5 , and x 6 , obtaining

maximize 2x 1 3x 2 C 3x 3 (29.33)

subject to

x 4 D 7 x 1 x 2 C x 3 (29.34)

x 5 D 7 C x 1 C x 2 x 3 (29.35)

x 6 D 4 x 1 C 2x 2 2x 3 (29.36)

x 1 ; x 2 ; x 3 ; x 4 ; x 5 ; x 6 0 : (29.37)

In this linear program, all the constraints except for the nonnegativity constraints

are equalities, and each variable is subject to a nonnegativity constraint. We write

each equality constraint with one of the variables on the left-hand side of the equal-

ity and all others on the right-hand side. Furthermore, each equation has the same

set of variables on the right-hand side, and these variables are also the only ones

that appear in the objective function. We call the variables on the left-hand side of

the equalities basic variables and those on the right-hand side nonbasic variables.

For linear programs that satisfy these conditions, we shall sometimes omit the

words “maximize” and “subject to,” as well as the explicit nonnegativity con-

straints. We shall also use the variable ´ to denote the value of the objective func-

856 Chapter 29 Linear Programming

tion. We call the resulting format slack form. If we write the linear program given

in (29.33)–(29.37) in slack form, we obtain

´ D 2x 1 3x 2 C 3x 3 (29.38)

x 4 D 7 x 1 x 2 C x 3 (29.39)

x 5 D 7 C x 1 C x 2 x 3 (29.40)

x 6 D 4 x 1 C 2x 2 2x 3 : (29.41)

As with standard form, we ﬁnd it convenient to have a more concise notation

for describing a slack form. As we shall see in Section 29.3, the sets of basic and

nonbasic variables will change as the simplex algorithm runs. We use N to denote

the set of indices of the nonbasic variables and B to denote the set of indices of

the basic variables. We always have that jN j D n, jBj D m, and N [ B D

f1; 2; : : : ; n C mg. The equations are indexed by the entries of B, and the variables

on the right-hand sides are indexed by the entries of N . As in standard form, we use

b i , c j , and a ij to denote constant terms and coefﬁcients. We also use to denote

an optional constant term in the objective function. (We shall see a little later that

including the constant term in the objective function makes it easy to determine the

value of the objective function.) Thus we can concisely deﬁne a slack form by a

tuple .N; B; A; b; c; /, denoting the slack form

´ D C

X

j 2N

c j x j (29.42)

x i D b i

X

j 2N

a ij x j for i 2 B ; (29.43)

in which all variables x are constrained to be nonnegative. Because we subtract

the sum

P

j 2N

a ij x j in (29.43), the values a ij are actually the negatives of the

coefﬁcients as they “appear” in the slack form.

For example, in the slack form

´ D 28

x 3

6

x 5

6

2x 6

3

x 1 D 8 C

x 3

6

C

x 5

6

x 6

3

x 2 D 4

8x 3

3

2x 5

3

C

x 6

3

x 4 D 18

x 3

2

C

x 5

2

;

we have B D f1; 2; 4g, N D f3; 5; 6g,

29.1 Standard and slack forms 857

A D

a 13 a 15 a 16

a 23 a 25 a 26

a 43 a 45 a 46

D

1=6 1=6 1=3

8=3 2=3 1=3

1=2 1=2 0

;

b D

b 1

b 2

b 4

D

8

4

18

;

c D

c 3 c 5 c 6

T

D

1=6 1=6 2=3

T

, and D 28. Note that the

indices into A, b, and c are not necessarily sets of contiguous integers; they depend

on the index sets B and N . As an example of the entries of A being the negatives

of the coefﬁcients as they appear in the slack form, observe that the equation for x 1

includes the term x 3 =6, yet the coefﬁcient a 13 is actually 1=6 rather than C1=6.

Exercises

29.1-1

If we express the linear program in (29.24)–(29.28) in the compact notation of

(29.19)–(29.21), what are n, m, A, b, and c?

29.1-2

Give three feasible solutions to the linear program in (29.24)–(29.28). What is the

objective value of each one?

29.1-3

For the slack form in (29.38)–(29.41), what are N , B, A, b, c, and ?

29.1-4

Convert the following linear program into standard form:

minimize 2x 1 C 7x 2 C x 3

subject to

x 1 x 3 D 7

3x 1 C x 2 24

x 2 0

x 3 0 :

858 Chapter 29 Linear Programming

29.1-5

Convert the following linear program into slack form:

maximize 2x 1 6x 3

subject to

x 1 C x 2 x 3 7

3x 1 x 2 8

x 1 C 2x 2 C 2x 3 0

x 1 ; x 2 ; x 3 0 :

What are the basic and nonbasic variables?

29.1-6

Show that the following linear program is infeasible:

maximize 3x 1 2x 2

subject to

x 1 C x 2 2

2x 1 2x 2 10

x 1 ; x 2 0 :

29.1-7

Show that the following linear program is unbounded:

maximize x 1 x 2

subject to

2x 1 C x 2 1

x 1 2x 2 2

x 1 ; x 2 0 :

29.1-8

Suppose that we have a general linear program with n variables and m constraints,

and suppose that we convert it into standard form. Give an upper bound on the

number of variables and constraints in the resulting linear program.

29.1-9

Give an example of a linear program for which the feasible region is not bounded,

but the optimal objective value is ﬁnite.

29.2 Formulating problems as linear programs 859

29.2 Formulating problems as linear programs

Although we shall focus on the simplex algorithm in this chapter, it is also impor-

tant to be able to recognize when we can formulate a problem as a linear program.

Once we cast a problem as a polynomial-sized linear program, we can solve it

in polynomial time by the ellipsoid algorithm or interior-point methods. Several

linear-programming software packages can solve problems efﬁciently, so that once

the problem is in the form of a linear program, such a package can solve it.

We shall look at several concrete examples of linear-programming problems. We

start with two problems that we have already studied: the single-source shortest-

paths problem (see Chapter 24) and the maximum-ﬂow problem (see Chapter 26).

We then describe the minimum-cost-ﬂow problem. Although the minimum-cost-

ﬂow problem has a polynomial-time algorithm that is not based on linear program-

ming, we won’t describe the algorithm. Finally, we describe the multicommodity-

ﬂow problem, for which the only known polynomial-time algorithm is based on

linear programming.

When we solved graph problems in Part VI, we used attribute notation, such

as :d and .u; /:f . Linear programs typically use subscripted variables rather

than objects with attached attributes, however. Therefore, when we express vari-

ables in linear programs, we shall indicate vertices and edges through subscripts.

For example, we denote the shortest-path weight for vertex not by :d but by d .

Similarly, we denote the ﬂow from vertex u to vertex not by .u; /:f but by f u .

For quantities that are given as inputs to problems, such as edge weights or capac-

ities, we shall continue to use notations such as w.u; / and c.u:/.

Shortest paths

We can formulate the single-source shortest-paths problem as a linear program.

In this section, we shall focus on how to formulate the single-pair shortest-path

problem, leaving the extension to the more general single-source shortest-paths

problem as Exercise 29.2-3.

In the single-pair shortest-path problem, we are given a weighted, directed graph

G D .V; E/, with weight function w W E ! R mapping edges to real-valued

weights, a source vertex s, and destination vertex t. We wish to compute the

value d t , which is the weight of a shortest path from s to t. To express this prob-

lem as a linear program, we need to determine a set of variables and constraints that

deﬁne when we have a shortest path from s to t. Fortunately, the Bellman-Ford al-

gorithm does exactly this. When the Bellman-Ford algorithm terminates, it has

computed, for each vertex , a value d (using subscript notation here rather than

attribute notation) such that for each edge .u; / 2 E, we have d d u C w.u; /.

860 Chapter 29 Linear Programming

The source vertex initially receives a value d s D 0, which never changes. Thus

we obtain the following linear program to compute the shortest-path weight from s

to t:

maximize d t (29.44)

subject to

d d u C w.u; / for each edge .u; / 2 E ; (29.45)

d s D 0 : (29.46)

You might be surprised that this linear program maximizes an objective function

when it is supposed to compute shortest paths. We do not want to minimize the

objective function, since then setting

N

d D 0 for all 2 V would yield an optimal

solution to the linear program without solving the shortest-paths problem. We

maximize because an optimal solution to the shortest-paths problem sets each

N

d

to min uW.u;/2E

˚

N

d u C w.u; /

, so that

N

d is the largest value that is less than or

equal to all of the values in the set

˚

N

d u C w.u; /

. We want to maximize d

for all vertices on a shortest path from s to t subject to these constraints on all

vertices , and maximizing d t achieves this goal.

This linear program has jV j variables d , one for each vertex 2 V . It also

has jEj C 1 constraints: one for each edge, plus the additional constraint that the

source vertex’s shortest-path weight always has the value 0.

Maximum ﬂow

Next, we express the maximum-ﬂow problem as a linear program. Recall that we

are given a directed graph G D .V; E/ in which each edge .u; / 2 E has a

nonnegative capacity c.u; / 0, and two distinguished vertices: a source s and

a sink t. As deﬁned in Section 26.1, a ﬂow is a nonnegative real-valued function

f W V V ! R that satisﬁes the capacity constraint and ﬂow conservation. A

maximum ﬂow is a ﬂow that satisﬁes these constraints and maximizes the ﬂow

value, which is the total ﬂow coming out of the source minus the total ﬂow into the

source. A ﬂow, therefore, satisﬁes linear constraints, and the value of a ﬂow is a

linear function. Recalling also that we assume that c.u; / D 0 if .u; / 62 E and

that there are no antiparallel edges, we can express the maximum-ﬂow problem as

a linear program:

maximize

X

2V

f s

X

2V

f s (29.47)

subject to

f u c.u; / for each u; 2 V ; (29.48)

X

2V

f u D

X

2V

f u for each u 2 V fs; tg ; (29.49)

f u 0 for each u; 2 V : (29.50)

29.2 Formulating problems as linear programs 861

This linear program has jV j

2

variables, corresponding to the ﬂow between each

pair of vertices, and it has 2 jV j

2

C jV j 2 constraints.

It is usually more efﬁcient to solve a smaller-sized linear program. The linear

program in (29.47)–(29.50) has, for ease of notation, a ﬂow and capacity of 0 for

each pair of vertices u; with .u; / 62 E. It would be more efﬁcient to rewrite the

linear program so that it has O.V C E/ constraints. Exercise 29.2-5 asks you to

do so.

Minimum-cost ﬂow

In this section, we have used linear programming to solve problems for which we

already knew efﬁcient algorithms. In fact, an efﬁcient algorithm designed specif-

ically for a problem, such as Dijkstra’s algorithm for the single-source shortest-

paths problem, or the push-relabel method for maximum ﬂow, will often be more

efﬁcient than linear programming, both in theory and in practice.

The real power of linear programming comes from the ability to solve new prob-

lems. Recall the problem faced by the politician in the beginning of this chapter.

The problem of obtaining a sufﬁcient number of votes, while not spending too

much money, is not solved by any of the algorithms that we have studied in this

book, yet we can solve it by linear programming. Books abound with such real-

world problems that linear programming can solve. Linear programming is also

particularly useful for solving variants of problems for which we may not already

know of an efﬁcient algorithm.

Consider, for example, the following generalization of the maximum-ﬂow prob-

lem. Suppose that, in addition to a capacity c.u; / for each edge .u; /, we are

given a real-valued cost a.u; /. As in the maximum-ﬂow problem, we assume that

c.u; / D 0 if .u; / 62 E, and that there are no antiparallel edges. If we send f u

units of ﬂow over edge .u; /, we incur a cost of a.u; /f u . We are also given a

ﬂow demand d. We wish to send d units of ﬂow from s to t while minimizing the

total cost

P

.u;/2E

a.u; /f u incurred by the ﬂow. This problem is known as the

minimum-cost-ﬂow problem.

Figure 29.3(a) shows an example of the minimum-cost-ﬂow problem. We wish

to send 4 units of ﬂow from s to t while incurring the minimum total cost. Any

particular legal ﬂow, that is, a function f satisfying constraints (29.48)–(29.49),

incurs a total cost of

P

.u;/2E

a.u; /f u . We wish to ﬁnd the particular 4-unit

ﬂow that minimizes this cost. Figure 29.3(b) shows an optimal solution, with total

cost

P

.u;/2E

a.u; /f u D .2 2/ C .5 2/ C .3 1/ C .7 1/ C .1 3/ D 27:

There are polynomial-time algorithms speciﬁcally designed for the minimum-

cost-ﬂow problem, but they are beyond the scope of this book. We can, however,

express the minimum-cost-ﬂow problem as a linear program. The linear program

looks similar to the one for the maximum-ﬂow problem with the additional con-

862 Chapter 29 Linear Programming

s

x

t

y

(a)

c = 1

a = 3

c = 5

a = 2

c = 4

a = 1

c = 2

a = 7

c = 2

a = 5

s

x

t

y

(b)

1/1

a = 3

2/5

a = 2

3/4

a = 1

1/2

a = 7

2/2

a = 5

Figure 29.3 (a) An example of a minimum-cost-ﬂow problem. We denote the capacities by c and

the costs by a. Vertex s is the source and vertex t is the sink, and we wish to send 4 units of ﬂow

from s to t. (b) A solution to the minimum-cost ﬂow problem in which 4 units of ﬂow are sent from s

to t. For each edge, the ﬂow and capacity are written as ﬂow/capacity.

straint that the value of the ﬂow be exactly d units, and with the new objective

function of minimizing the cost:

minimize

X

.u;/2E

a.u; /f u (29.51)

subject to

f u c.u; / for each u; 2 V ;

X

2V

f u

X

2V

f u D 0 for each u 2 V fs; tg ;

X

2V

f s

X

2V

f s D d ;

f u 0 for each u; 2 V : (29.52)

Multicommodity ﬂow

As a ﬁnal example, we consider another ﬂow problem. Suppose that the Lucky

Puck company from Section 26.1 decides to diversify its product line and ship

not only hockey pucks, but also hockey sticks and hockey helmets. Each piece of

equipment is manufactured in its own factory, has its own warehouse, and must

be shipped, each day, from factory to warehouse. The sticks are manufactured in

Vancouver and must be shipped to Saskatoon, and the helmets are manufactured in

Edmonton and must be shipped to Regina. The capacity of the shipping network

does not change, however, and the different items, or commodities, must share the

same network.

This example is an instance of a multicommodity-ﬂow problem. In this problem,

we are again given a directed graph G D .V; E/ in which each edge .u; / 2 E

has a nonnegative capacity c.u; / 0. As in the maximum-ﬂow problem, we im-

plicitly assume that c.u; / D 0 for .u; / 62 E, and that the graph has no antipar-

29.2 Formulating problems as linear programs 863

allel edges. In addition, we are given k different commodities, K 1 ; K 2 ; : : : ; K k ,

where we specify commodity i by the triple K i D .s i ; t i ; d i /. Here, vertex s i is

the source of commodity i, vertex t i is the sink of commodity i, and d i is the de-

mand for commodity i, which is the desired ﬂow value for the commodity from s i

to t i . We deﬁne a ﬂow for commodity i, denoted by f i , (so that f iu is the ﬂow of

commodity i from vertex u to vertex ) to be a real-valued function that satisﬁes

the ﬂow-conservation and capacity constraints. We now deﬁne f u , the aggregate

ﬂow, to be the sum of the various commodity ﬂows, so that f u D

P k

iD1

f iu . The

aggregate ﬂow on edge .u; / must be no more than the capacity of edge .u; /.

We are not trying to minimize any objective function in this problem; we need

only determine whether such a ﬂow exists. Thus, we write a linear program with a

“null” objective function:

minimize 0

subject to

k X

iD1

f iu c.u; / for each u; 2 V ;

X

2V

f iu

X

2V

f iu D 0 for each i D 1; 2; : : : ; k and

for each u 2 V fs i ; t i g ;

X

2V

f i;s i ;

X

2V

f i;;s i D d i for each i D 1; 2; : : : ; k ;

f iu 0 for each u; 2 V and

for each i D 1; 2; : : : ; k :

The only known polynomial-time algorithm for this problem expresses it as a linear

program and then solves it with a polynomial-time linear-programming algorithm.

Exercises

29.2-1

Put the single-pair shortest-path linear program from (29.44)–(29.46) into standard

form.

29.2-2

Write out explicitly the linear program corresponding to ﬁnding the shortest path

from node s to node y in Figure 24.2(a).

29.2-3

In the single-source shortest-paths problem, we want to ﬁnd the shortest-path

weights from a source vertex s to all vertices 2 V . Given a graph G, write a

864 Chapter 29 Linear Programming

linear program for which the solution has the property that d is the shortest-path

weight from s to for each vertex 2 V .

29.2-4

Write out explicitly the linear program corresponding to ﬁnding the maximum ﬂow

in Figure 26.1(a).

29.2-5

Rewrite the linear program for maximum ﬂow (29.47)–(29.50) so that it uses only

O.V C E/ constraints.

29.2-6

Write a linear program that, given a bipartite graph G D .V; E/, solves the maxi-

mum-bipartite-matching problem.

29.2-7

In the minimum-cost multicommodity-ﬂow problem, we are given directed graph

G D .V; E/ in which each edge .u; / 2 E has a nonnegative capacity c.u; / 0

and a cost a.u; /. As in the multicommodity-ﬂow problem, we are given k dif-

ferent commodities, K 1 ; K 2 ; : : : ; K k , where we specify commodity i by the triple

K i D .s i ; t i ; d i /. We deﬁne the ﬂow f i for commodity i and the aggregate ﬂow f u

on edge .u; / as in the multicommodity-ﬂow problem. A feasible ﬂow is one

in which the aggregate ﬂow on each edge .u; / is no more than the capacity of

edge .u; /. The cost of a ﬂow is

P

u;2V

a.u; /f u , and the goal is to ﬁnd the

feasible ﬂow of minimum cost. Express this problem as a linear program.

29.3 The simplex algorithm

The simplex algorithm is the classical method for solving linear programs. In con-

trast to most of the other algorithms in this book, its running time is not polynomial

in the worst case. It does yield insight into linear programs, however, and is often

remarkably fast in practice.

In addition to having a geometric interpretation, described earlier in this chapter,

the simplex algorithm bears some similarity to Gaussian elimination, discussed in

Section 28.1. Gaussian elimination begins with a system of linear equalities whose

solution is unknown. In each iteration, we rewrite this system in an equivalent

form that has some additional structure. After some number of iterations, we have

rewritten the system so that the solution is simple to obtain. The simplex algo-

rithm proceeds in a similar manner, and we can view it as Gaussian elimination for

inequalities.

29.3 The simplex algorithm 865

We now describe the main idea behind an iteration of the simplex algorithm.

Associated with each iteration will be a “basic solution” that we can easily obtain

from the slack form of the linear program: set each nonbasic variable to 0 and

compute the values of the basic variables from the equality constraints. An iteration

converts one slack form into an equivalent slack form. The objective value of the

associated basic feasible solution will be no less than that at the previous iteration,

and usually greater. To achieve this increase in the objective value, we choose a

nonbasic variable such that if we were to increase that variable’s value from 0, then

the objective value would increase, too. The amount by which we can increase

the variable is limited by the other constraints. In particular, we raise it until some

basic variable becomes 0. We then rewrite the slack form, exchanging the roles

of that basic variable and the chosen nonbasic variable. Although we have used a

particular setting of the variables to guide the algorithm, and we shall use it in our

proofs, the algorithm does not explicitly maintain this solution. It simply rewrites

the linear program until an optimal solution becomes “obvious.”

An example of the simplex algorithm

We begin with an extended example. Consider the following linear program in

standard form:

maximize 3x 1 C x 2 C 2x 3 (29.53)

subject to

x 1 C x 2 C 3x 3 30 (29.54)

2x 1 C 2x 2 C 5x 3 24 (29.55)

4x 1 C x 2 C 2x 3 36 (29.56)

x 1 ; x 2 ; x 3 0 : (29.57)

In order to use the simplex algorithm, we must convert the linear program into

slack form; we saw how to do so in Section 29.1. In addition to being an algebraic

manipulation, slack is a useful algorithmic concept. Recalling from Section 29.1

that each variable has a corresponding nonnegativity constraint, we say that an

equality constraint is tight for a particular setting of its nonbasic variables if they

cause the constraint’s basic variable to become 0. Similarly, a setting of the non-

basic variables that would make a basic variable become negative violates that

constraint. Thus, the slack variables explicitly maintain how far each constraint is

from being tight, and so they help to determine how much we can increase values

of nonbasic variables without violating any constraints.

Associating the slack variables x 4 , x 5 , and x 6 with inequalities (29.54)–(29.56),

respectively, and putting the linear program into slack form, we obtain

866 Chapter 29 Linear Programming

´ D 3x 1 C x 2 C 2x 3 (29.58)

x 4 D 30 x 1 x 2 3x 3 (29.59)

x 5 D 24 2x 1 2x 2 5x 3 (29.60)

x 6 D 36 4x 1 x 2 2x 3 : (29.61)

The system of constraints (29.59)–(29.61) has 3 equations and 6 variables. Any

setting of the variables x 1 , x 2 , and x 3 deﬁnes values for x 4 , x 5 , and x 6 ; therefore,

we have an inﬁnite number of solutions to this system of equations. A solution is

feasible if all of x 1 ; x 2 ; : : : ; x 6 are nonnegative, and there can be an inﬁnite num-

ber of feasible solutions as well. The inﬁnite number of possible solutions to a

system such as this one will be useful in later proofs. We focus on the basic solu-

tion: set all the (nonbasic) variables on the right-hand side to 0 and then compute

the values of the (basic) variables on the left-hand side. In this example, the ba-

sic solution is . Nx 1 ; Nx 2 ; : : : ; Nx 6 / D .0; 0; 0; 30; 24; 36/ and it has objective value

´ D .3 0/ C .1 0/ C .2 0/ D 0. Observe that this basic solution sets Nx i D b i

for each i 2 B. An iteration of the simplex algorithm rewrites the set of equations

and the objective function so as to put a different set of variables on the right-

hand side. Thus, a different basic solution is associated with the rewritten problem.

We emphasize that the rewrite does not in any way change the underlying linear-

programming problem; the problem at one iteration has the identical set of feasible

solutions as the problem at the previous iteration. The problem does, however,

have a different basic solution than that of the previous iteration.

If a basic solution is also feasible, we call it a basic feasible solution. As we run

the simplex algorithm, the basic solution is almost always a basic feasible solution.

We shall see in Section 29.5, however, that for the ﬁrst few iterations of the simplex

algorithm, the basic solution might not be feasible.

Our goal, in each iteration, is to reformulate the linear program so that the basic

solution has a greater objective value. We select a nonbasic variable x e whose

coefﬁcient in the objective function is positive, and we increase the value of x e as

much as possible without violating any of the constraints. The variable x e becomes

basic, and some other variable x l becomes nonbasic. The values of other basic

variables and of the objective function may also change.

To continue the example, let’s think about increasing the value of x 1 . As we

increase x 1 , the values of x 4 , x 5 , and x 6 all decrease. Because we have a nonnega-

tivity constraint for each variable, we cannot allow any of them to become negative.

If x 1 increases above 30, then x 4 becomes negative, and x 5 and x 6 become nega-

tive when x 1 increases above 12 and 9, respectively. The third constraint (29.61) is

the tightest constraint, and it limits how much we can increase x 1 . Therefore, we

switch the roles of x 1 and x 6 . We solve equation (29.61) for x 1 and obtain

x 1 D 9

x 2

4

x 3

2

x 6

4

: (29.62)

29.3 The simplex algorithm 867

To rewrite the other equations with x 6 on the right-hand side, we substitute for x 1

using equation (29.62). Doing so for equation (29.59), we obtain

x 4 D 30 x 1 x 2 3x 3

D 30

9

x 2

4

x 3

2

x 6

4

x 2 3x 3

D 21

3x 2

4

5x 3

2

C

x 6

4

: (29.63)

Similarly, we combine equation (29.62) with constraint (29.60) and with objective

function (29.58) to rewrite our linear program in the following form:

´ D 27 C

x 2

4

C

x 3

2

3x 6

4

(29.64)

x 1 D 9

x 2

4

x 3

2

x 6

4

(29.65)

x 4 D 21

3x 2

4

5x 3

2

C

x 6

4

(29.66)

x 5 D 6

3x 2

2

4x 3 C

x 6

2

: (29.67)

We call this operation a pivot. As demonstrated above, a pivot chooses a nonbasic

variable x e , called the entering variable, and a basic variable x l , called the leaving

variable, and exchanges their roles.

The linear program described in equations (29.64)–(29.67) is equivalent to the

linear program described in equations (29.58)–(29.61). We perform two operations

in the simplex algorithm: rewrite equations so that variables move between the left-

hand side and the right-hand side, and substitute one equation into another. The ﬁrst

operation trivially creates an equivalent problem, and the second, by elementary

linear algebra, also creates an equivalent problem. (See Exercise 29.3-3.)

To demonstrate this equivalence, observe that our original basic solution .0; 0;

0; 30; 24; 36/ satisﬁes the new equations (29.65)–(29.67) and has objective value

27 C .1=4/ 0 C .1=2/ 0 .3=4/ 36 D 0. The basic solution associated with the

new linear program sets the nonbasic values to 0 and is .9; 0; 0; 21; 6; 0/, with ob-

jective value ´ D 27. Simple arithmetic veriﬁes that this solution also satisﬁes

equations (29.59)–(29.61) and, when plugged into objective function (29.58), has

objective value .3 9/ C .1 0/ C .2 0/ D 27.

Continuing the example, we wish to ﬁnd a new variable whose value we wish to

increase. We do not want to increase x 6 , since as its value increases, the objective

value decreases. We can attempt to increase either x 2 or x 3 ; let us choose x 3 . How

far can we increase x 3 without violating any of the constraints? Constraint (29.65)

limits it to 18, constraint (29.66) limits it to 42=5, and constraint (29.67) limits

it to 3=2. The third constraint is again the tightest one, and therefore we rewrite

the third constraint so that x 3 is on the left-hand side and x 5 is on the right-hand

868 Chapter 29 Linear Programming

side. We then substitute this new equation, x 3 D 3=2 3x 2 =8 x 5 =4 C x 6 =8, into

equations (29.64)–(29.66) and obtain the new, but equivalent, system

´ D

111

4

C

x 2

16

x 5

8

11x 6

16

(29.68)

x 1 D

33

4

x 2

16

C

x 5

8

5x 6

16

(29.69)

x 3 D

3

2

3x 2

8

x 5

4

C

x 6

8

(29.70)

x 4 D

69

4

C

3x 2

16

C

5x 5

8

x 6

16

: (29.71)

This system has the associated basic solution .33=4; 0; 3=2; 69=4; 0; 0/, with ob-

jective value 111=4. Now the only way to increase the objective value is to in-

crease x 2 . The three constraints give upper bounds of 132, 4, and 1, respectively.

(We get an upper bound of 1 from constraint (29.71) because, as we increase x 2 ,

the value of the basic variable x 4 increases also. This constraint, therefore, places

no restriction on how much we can increase x 2 .) We increase x 2 to 4, and it be-

comes nonbasic. Then we solve equation (29.70) for x 2 and substitute in the other

equations to obtain

´ D 28

x 3

6

x 5

6

2x 6

3

(29.72)

x 1 D 8 C

x 3

6

C

x 5

6

x 6

3

(29.73)

x 2 D 4

8x 3

3

2x 5

3

C

x 6

3

(29.74)

x 4 D 18

x 3

2

C

x 5

2

: (29.75)

At this point, all coefﬁcients in the objective function are negative. As we shall see

later in this chapter, this situation occurs only when we have rewritten the linear

program so that the basic solution is an optimal solution. Thus, for this problem,

the solution .8; 4; 0; 18; 0; 0/, with objective value 28, is optimal. We can now

return to our original linear program given in (29.53)–(29.57). The only variables

in the original linear program are x 1 , x 2 , and x 3 , and so our solution is x 1 D 8,

x 2 D 4, and x 3 D 0, with objective value .3 8/ C .1 4/ C .2 0/ D 28. Note

that the values of the slack variables in the ﬁnal solution measure how much slack

remains in each inequality. Slack variable x 4 is 18, and in inequality (29.54), the

left-hand side, with value 8 C 4 C 0 D 12, is 18 less than the right-hand side of 30.

Slack variables x 5 and x 6 are 0 and indeed, in inequalities (29.55) and (29.56),

the left-hand and right-hand sides are equal. Observe also that even though the

coefﬁcients in the original slack form are integral, the coefﬁcients in the other

linear programs are not necessarily integral, and the intermediate solutions are not

29.3 The simplex algorithm 869

necessarily integral. Furthermore, the ﬁnal solution to a linear program need not

be integral; it is purely coincidental that this example has an integral solution.

Pivoting

We now formalize the procedure for pivoting. The procedure PIVOT takes as in-

put a slack form, given by the tuple .N; B; A; b; c; /, the index l of the leav-

ing variable x l , and the index e of the entering variable x e . It returns the tuple

. yN ; yB; yA;

y

b; yc; y/ describing the new slack form. (Recall again that the entries of

the m n matrices A and yA are actually the negatives of the coefﬁcients that appear

in the slack form.)

PIVOT.N; B; A; b; c; ; l; e/

1 // Compute the coefﬁcients of the equation for new basic variable x e .

2 let yA be a new m n matrix

3

y

b e D b l =a le

4 for each j 2 N feg

5 ya ej D a lj =a le

6 ya el D 1=a le

7 // Compute the coefﬁcients of the remaining constraints.

8 for each i 2 B flg

9

y

b i D b i a ie

y

b e

10 for each j 2 N feg

11 ya ij D a ij a ie ya ej

12 ya il D a ie ya el

13 // Compute the objective function.

14 y D C c e

y

b e

15 for each j 2 N feg

16 yc j D c j c e ya ej

17 yc l D c e ya el

18 // Compute new sets of basic and nonbasic variables.

19 yN D N feg [ flg

20 yB D B flg [ feg

21 return . yN ; yB; yA;

y

b; yc; y/

PIVOT works as follows. Lines 3–6 compute the coefﬁcients in the new equation

for x e by rewriting the equation that has x l on the left-hand side to instead have x e

on the left-hand side. Lines 8–12 update the remaining equations by substituting

the right-hand side of this new equation for each occurrence of x e . Lines 14–17

do the same substitution for the objective function, and lines 19 and 20 update the

870 Chapter 29 Linear Programming

sets of nonbasic and basic variables. Line 21 returns the new slack form. As given,

if a le D 0, PIVOT would cause an error by dividing by 0, but as we shall see in the

proofs of Lemmas 29.2 and 29.12, we call PIVOT only when a le ¤ 0.

We now summarize the effect that PIVOT has on the values of the variables in

the basic solution.

Lemma 29.1

Consider a call to PIVOT.N; B; A; b; c; ; l; e/ in which a le ¤ 0. Let the values

returned from the call be . yN ; yB; yA;

y

b; yc; y/, and let Nx denote the basic solution after

the call. Then

1. Nx j D 0 for each j 2 yN .

2. Nx e D b l =a le .

3. Nx i D b i a ie

y

b e for each i 2 yB feg.

Proof The ﬁrst statement is true because the basic solution always sets all non-

basic variables to 0. When we set each nonbasic variable to 0 in a constraint

x i D

y

b i

X

j 2 yN

ya ij x j ;

we have that Nx i D

y

b i for each i 2 yB. Since e 2 yB, line 3 of PIVOT gives

Nx e D

y

b e D b l =a le ;

which proves the second statement. Similarly, using line 9 for each i 2 yB feg,

we have

Nx i D

y

b i D b i a ie

y

b e ;

which proves the third statement.

The formal simplex algorithm

We are now ready to formalize the simplex algorithm, which we demonstrated by

example. That example was a particularly nice one, and we could have had several

other issues to address:

How do we determine whether a linear program is feasible?

What do we do if the linear program is feasible, but the initial basic solution is

not feasible?

How do we determine whether a linear program is unbounded?

How do we choose the entering and leaving variables?

29.3 The simplex algorithm 871

In Section 29.5, we shall show how to determine whether a problem is feasible,

and if so, how to ﬁnd a slack form in which the initial basic solution is feasible.

Therefore, let us assume that we have a procedure INITIALIZE-SIMPLEX.A; b; c/

that takes as input a linear program in standard form, that is, an m n matrix

A D .a ij /, an m-vector b D .b i /, and an n-vector c D .c j /. If the problem is

infeasible, the procedure returns a message that the program is infeasible and then

terminates. Otherwise, the procedure returns a slack form for which the initial

basic solution is feasible.

The procedure SIMPLEX takes as input a linear program in standard form, as just

described. It returns an n-vector Nx D . Nx j / that is an optimal solution to the linear

program described in (29.19)–(29.21).

SIMPLEX.A; b; c/

1 .N; B; A; b; c; / D INITIALIZE-SIMPLEX.A; b; c/

2 let be a new vector of length n

3 while some index j 2 N has c j > 0

4 choose an index e 2 N for which c e > 0

5 for each index i 2 B

6 if a ie > 0

7 i D b i =a ie

8 else i D 1

9 choose an index l 2 B that minimizes i

10 if l == 1

11 return “unbounded”

12 else .N; B; A; b; c; / D PIVOT.N; B; A; b; c; ; l; e/

13 for i D 1 to n

14 if i 2 B

15 Nx i D b i

16 else Nx i D 0

17 return . Nx 1 ; Nx 2 ; : : : ; Nx n /

The SIMPLEX procedure works as follows. In line 1, it calls the procedure

INITIALIZE-SIMPLEX.A; b; c/, described above, which either determines that the

linear program is infeasible or returns a slack form for which the basic solution is

feasible. The while loop of lines 3–12 forms the main part of the algorithm. If all

coefﬁcients in the objective function are negative, then the while loop terminates.

Otherwise, line 4 selects a variable x e , whose coefﬁcient in the objective function

is positive, as the entering variable. Although we may choose any such variable as

the entering variable, we assume that we use some prespeciﬁed deterministic rule.

Next, lines 5–9 check each constraint and pick the one that most severely limits

the amount by which we can increase x e without violating any of the nonnegativ-

872 Chapter 29 Linear Programming

ity constraints; the basic variable associated with this constraint is x l . Again, we

are free to choose one of several variables as the leaving variable, but we assume

that we use some prespeciﬁed deterministic rule. If none of the constraints lim-

its the amount by which the entering variable can increase, the algorithm returns

“unbounded” in line 11. Otherwise, line 12 exchanges the roles of the entering

and leaving variables by calling PIVOT.N; B; A; b; c; ; l; e/, as described above.

Lines 13–16 compute a solution Nx 1 ; Nx 2 ; : : : ; Nx n for the original linear-programming

variables by setting all the nonbasic variables to 0 and each basic variable Nx i to b i ,

and line 17 returns these values.

To show that SIMPLEX is correct, we ﬁrst show that if SIMPLEX has an initial

feasible solution and eventually terminates, then it either returns a feasible solution

or determines that the linear program is unbounded. Then, we show that SIMPLEX

terminates. Finally, in Section 29.4 (Theorem 29.10) we show that the solution

returned is optimal.

Lemma 29.2

Given a linear program .A; b; c/, suppose that the call to INITIALIZE-SIMPLEX in

line 1 of SIMPLEX returns a slack form for which the basic solution is feasible.

Then if SIMPLEX returns a solution in line 17, that solution is a feasible solution to

the linear program. If SIMPLEX returns “unbounded” in line 11, the linear program

is unbounded.

Proof We use the following three-part loop invariant:

At the start of each iteration of the while loop of lines 3–12,

1. the slack form is equivalent to the slack form returned by the call of

INITIALIZE-SIMPLEX,

2. for each i 2 B, we have b i 0, and

3. the basic solution associated with the slack form is feasible.

Initialization: The equivalence of the slack forms is trivial for the ﬁrst itera-

tion. We assume, in the statement of the lemma, that the call to INITIALIZE-

SIMPLEX in line 1 of SIMPLEX returns a slack form for which the basic solution

is feasible. Thus, the third part of the invariant is true. Because the basic so-

lution is feasible, each basic variable x i is nonnegative. Furthermore, since the

basic solution sets each basic variable x i to b i , we have that b i 0 for all

i 2 B. Thus, the second part of the invariant holds.

Maintenance: We shall show that each iteration of the while loop maintains the

loop invariant, assuming that the return statement in line 11 does not execute.

We shall handle the case in which line 11 executes when we discuss termination.

29.3 The simplex algorithm 873

An iteration of the while loop exchanges the role of a basic and a nonbasic

variable by calling the PIVOT procedure. By Exercise 29.3-3, the slack form is

equivalent to the one from the previous iteration which, by the loop invariant,

is equivalent to the initial slack form.

We now demonstrate the second part of the loop invariant. We assume that at

the start of each iteration of the while loop, b i 0 for each i 2 B, and we shall

show that these inequalities remain true after the call to PIVOT in line 12. Since

the only changes to the variables b i and the set B of basic variables occur in this

assignment, it sufﬁces to show that line 12 maintains this part of the invariant.

We let b i , a ij , and B refer to values before the call of PIVOT, and

y

b i refer to

values returned from PIVOT.

First, we observe that

y

b e 0 because b l 0 by the loop invariant, a le > 0 by

lines 6 and 9 of SIMPLEX, and

y

b e D b l =a le by line 3 of PIVOT.

For the remaining indices i 2 B flg, we have that

y

b i D b i a ie

y

b e (by line 9 of PIVOT)

D b i a ie .b l =a le / (by line 3 of PIVOT) . (29.76)

We have two cases to consider, depending on whether a ie > 0 or a ie 0.

If a ie > 0, then since we chose l such that

b l =a le b i =a ie for all i 2 B ; (29.77)

we have

y

b i D b i a ie .b l =a le / (by equation (29.76))

b i a ie .b i =a ie / (by inequality (29.77))

D b i b i

D 0 ;

and thus

y

b i 0. If a ie 0, then because a le , b i , and b l are all nonnegative,

equation (29.76) implies that

y

b i must be nonnegative, too.

We now argue that the basic solution is feasible, i.e., that all variables have non-

negative values. The nonbasic variables are set to 0 and thus are nonnegative.

Each basic variable x i is deﬁned by the equation

x i D b i

X

j 2N

a ij x j :

The basic solution sets Nx i D b i . Using the second part of the loop invariant, we

conclude that each basic variable Nx i is nonnegative.

874 Chapter 29 Linear Programming

Termination: The while loop can terminate in one of two ways. If it terminates

because of the condition in line 3, then the current basic solution is feasible and

line 17 returns this solution. The other way it terminates is by returning “un-

bounded” in line 11. In this case, for each iteration of the for loop in lines 5–8,

when line 6 is executed, we ﬁnd that a ie 0. Consider the solution Nx deﬁned as

Nx i D

1 if i D e ;

0 if i 2 N feg ;

b i

P

j 2N

a ij Nx j if i 2 B :

We now show that this solution is feasible, i.e., that all variables are nonneg-

ative. The nonbasic variables other than Nx e are 0, and Nx e D 1 > 0; thus all

nonbasic variables are nonnegative. For each basic variable Nx i , we have

Nx i D b i

X

j 2N

a ij Nx j

D b i a ie Nx e :

The loop invariant implies that b i 0, and we have a ie 0 and Nx e D 1 > 0.

Thus, Nx i 0.

Now we show that the objective value for the solution Nx is unbounded. From

equation (29.42), the objective value is

´ D C

X

j 2N

c j Nx j

D C c e Nx e :

Since c e > 0 (by line 4 of SIMPLEX) and Nx e D 1, the objective value is 1,

and thus the linear program is unbounded.

It remains to show that SIMPLEX terminates, and when it does terminate, the

solution it returns is optimal. Section 29.4 will address optimality. We now discuss

termination.

Termination

In the example given in the beginning of this section, each iteration of the simplex

algorithm increased the objective value associated with the basic solution. As Ex-

ercise 29.3-2 asks you to show, no iteration of SIMPLEX can decrease the objective

value associated with the basic solution. Unfortunately, it is possible that an itera-

tion leaves the objective value unchanged. This phenomenon is called degeneracy,

and we shall now study it in greater detail.

29.3 The simplex algorithm 875

The assignment in line 14 of PIVOT, y D C c e

y

b e , changes the objective value.

Since SIMPLEX calls PIVOT only when c e > 0, the only way for the objective

value to remain unchanged (i.e., y D ) is for

y

b e to be 0. This value is assigned

as

y

b e D b l =a le in line 3 of PIVOT. Since we always call PIVOT with a le ¤ 0, we

see that for

y

b e to equal 0, and hence the objective value to be unchanged, we must

have b l D 0.

Indeed, this situation can occur. Consider the linear program

´ D x 1 C x 2 C x 3

x 4 D 8 x 1 x 2

x 5 D x 2 x 3 :

Suppose that we choose x 1 as the entering variable and x 4 as the leaving variable.

After pivoting, we obtain

´ D 8 C x 3 x 4

x 1 D 8 x 2 x 4

x 5 D x 2 x 3 :

At this point, our only choice is to pivot with x 3 entering and x 5 leaving. Since

b 5 D 0, the objective value of 8 remains unchanged after pivoting:

´ D 8 C x 2 x 4 x 5

x 1 D 8 x 2 x 4

x 3 D x 2 x 5 :

The objective value has not changed, but our slack form has. Fortunately, if we

pivot again, with x 2 entering and x 1 leaving, the objective value increases (to 16),

and the simplex algorithm can continue.

Degeneracy can prevent the simplex algorithm from terminating, because it can

lead to a phenomenon known as cycling: the slack forms at two different itera-

tions of SIMPLEX are identical. Because of degeneracy, SIMPLEX could choose a

sequence of pivot operations that leave the objective value unchanged but repeat

a slack form within the sequence. Since SIMPLEX is a deterministic algorithm, if

it cycles, then it will cycle through the same series of slack forms forever, never

terminating.

Cycling is the only reason that SIMPLEX might not terminate. To show this fact,

we must ﬁrst develop some additional machinery.

At each iteration, SIMPLEX maintains A, b, c, and in addition to the sets

N and B. Although we need to explicitly maintain A, b, c, and in order to

implement the simplex algorithm efﬁciently, we can get by without maintaining

them. In other words, the sets of basic and nonbasic variables sufﬁce to uniquely

determine the slack form. Before proving this fact, we prove a useful algebraic

lemma.

876 Chapter 29 Linear Programming

Lemma 29.3

Let I be a set of indices. For each j 2 I, let ˛ j and ˇ j be real numbers, and let x j

be a real-valued variable. Let be any real number. Suppose that for any settings

of the x j , we have

X

j 2I

˛ j x j D C

X

j 2I

ˇ j x j : (29.78)

Then ˛ j D ˇ j for each j 2 I, and D 0.

Proof Since equation (29.78) holds for any values of the x j , we can use particular

values to draw conclusions about ˛, ˇ, and . If we let x j D 0 for each j 2 I,

we conclude that D 0. Now pick an arbitrary index j 2 I, and set x j D 1 and

x k D 0 for all k ¤ j . Then we must have ˛ j D ˇ j . Since we picked j as any

index in I, we conclude that ˛ j D ˇ j for each j 2 I.

A particular linear program has many different slack forms; recall that each slack

form has the same set of feasible and optimal solutions as the original linear pro-

gram. We now show that the slack form of a linear program is uniquely determined

by the set of basic variables. That is, given the set of basic variables, a unique slack

form (unique set of coefﬁcients and right-hand sides) is associated with those basic

variables.

Lemma 29.4

Let .A; b; c/ be a linear program in standard form. Given a set B of basic variables,

the associated slack form is uniquely determined.

Proof Assume for the purpose of contradiction that there are two different slack

forms with the same set B of basic variables. The slack forms must also have

identical sets N D f1; 2; : : : ; n C mg B of nonbasic variables. We write the ﬁrst

slack form as

´ D C

X

j 2N

c j x j (29.79)

x i D b i

X

j 2N

a ij x j for i 2 B ; (29.80)

and the second as

´ D

0

C

X

j 2N

c

0

j

x j (29.81)

x i D b

0

i

X

j 2N

a

0

ij

x j for i 2 B : (29.82)

29.3 The simplex algorithm 877

Consider the system of equations formed by subtracting each equation in

line (29.82) from the corresponding equation in line (29.80). The resulting sys-

tem is

0 D .b i b

0

i

/

X

j 2N

.a ij a

0

ij

/x j for i 2 B

or, equivalently,

X

j 2N

a ij x j D .b i b

0

i

/ C

X

j 2N

a

0

ij

x j for i 2 B :

Now, for each i 2 B, apply Lemma 29.3 with ˛ j D a ij , ˇ j D a 0

ij

, D b i b 0

i

, and

I D N . Since ˛ i D ˇ i , we have that a ij D a 0

ij

for each j 2 N , and since D 0,

we have that b i D b 0

i

. Thus, for the two slack forms, A and b are identical to A 0

and b 0

. Using a similar argument, Exercise 29.3-1 shows that it must also be the

case that c D c 0

and D 0

, and hence that the slack forms must be identical.

We can now show that cycling is the only possible reason that SIMPLEX might

not terminate.

Lemma 29.5

If SIMPLEX fails to terminate in at most

nCm

m

iterations, then it cycles.

Proof By Lemma 29.4, the set B of basic variables uniquely determines a slack

form. There are n C m variables and jBj D m, and therefore, there are at most   
 nCm

m

ways to choose B. Thus, there are only at most

nCm

m

unique slack forms.

Therefore, if SIMPLEX runs for more than

nCm

m

iterations, it must cycle.

Cycling is theoretically possible, but extremely rare. We can prevent it by choos-

ing the entering and leaving variables somewhat more carefully. One option is to

perturb the input slightly so that it is impossible to have two solutions with the

same objective value. Another option is to break ties by always choosing the vari-

able with the smallest index, a strategy known as Bland’s rule. We omit the proof

that these strategies avoid cycling.

Lemma 29.6

If lines 4 and 9 of SIMPLEX always break ties by choosing the variable with the

smallest index, then SIMPLEX must terminate.

We conclude this section with the following lemma.

878 Chapter 29 Linear Programming

Lemma 29.7

Assuming that INITIALIZE-SIMPLEX returns a slack form for which the basic so-

lution is feasible, SIMPLEX either reports that a linear program is unbounded, or it

terminates with a feasible solution in at most

nCm

m

iterations.

Proof Lemmas 29.2 and 29.6 show that if INITIALIZE-SIMPLEX returns a slack

form for which the basic solution is feasible, SIMPLEX either reports that a linear

program is unbounded, or it terminates with a feasible solution. By the contra-

positive of Lemma 29.5, if SIMPLEX terminates with a feasible solution, then it

terminates in at most

nCm

m

iterations.

Exercises

29.3-1

Complete the proof of Lemma 29.4 by showing that it must be the case that c D c 0

and D 0

.

29.3-2

Show that the call to PIVOT in line 12 of SIMPLEX never decreases the value of .

29.3-3

Prove that the slack form given to the PIVOT procedure and the slack form that the

procedure returns are equivalent.

29.3-4

Suppose we convert a linear program .A; b; c/ in standard form to slack form.

Show that the basic solution is feasible if and only if b i 0 for i D 1; 2; : : : ; m.

29.3-5

Solve the following linear program using SIMPLEX:

maximize 18x 1 C 12:5x 2

subject to

x 1 C x 2 20

x 1 12

x 2 16

x 1 ; x 2 0 :

29.4 Duality 879

29.3-6

Solve the following linear program using SIMPLEX:

maximize 5x 1 3x 2

subject to

x 1 x 2 1

2x 1 C x 2 2

x 1 ; x 2 0 :

29.3-7

Solve the following linear program using SIMPLEX:

minimize x 1 C x 2 C x 3

subject to

2x 1 C 7:5x 2 C 3x 3 10000

20x 1 C 5x 2 C 10x 3 30000

x 1 ; x 2 ; x 3 0 :

29.3-8

In the proof of Lemma 29.5, we argued that there are at most

mCn

n

ways to choose

a set B of basic variables. Give an example of a linear program in which there are

strictly fewer than

mCn

n

ways to choose the set B.

29.4 Duality

We have proven that, under certain assumptions, SIMPLEX terminates. We have not

yet shown that it actually ﬁnds an optimal solution to a linear program, however.

In order to do so, we introduce a powerful concept called linear-programming

duality.

Duality enables us to prove that a solution is indeed optimal. We saw an exam-

ple of duality in Chapter 26 with Theorem 26.6, the max-ﬂow min-cut theorem.

Suppose that, given an instance of a maximum-ﬂow problem, we ﬁnd a ﬂow f

with value jf j. How do we know whether f is a maximum ﬂow? By the max-ﬂow

min-cut theorem, if we can ﬁnd a cut whose value is also jf j, then we have ver-

iﬁed that f is indeed a maximum ﬂow. This relationship provides an example of

duality: given a maximization problem, we deﬁne a related minimization problem

such that the two problems have the same optimal objective values.

Given a linear program in which the objective is to maximize, we shall describe

how to formulate a dual linear program in which the objective is to minimize and

880 Chapter 29 Linear Programming

whose optimal value is identical to that of the original linear program. When refer-

ring to dual linear programs, we call the original linear program the primal.

Given a primal linear program in standard form, as in (29.16)–(29.18), we deﬁne

the dual linear program as

minimize

m X

iD1

b i y i (29.83)

subject to

m X

iD1

a ij y i c j for j D 1; 2; : : : ; n ; (29.84)

y i 0 for i D 1; 2; : : : ; m : (29.85)

To form the dual, we change the maximization to a minimization, exchange the

roles of coefﬁcients on the right-hand sides and the objective function, and replace

each less-than-or-equal-to by a greater-than-or-equal-to. Each of the m constraints

in the primal has an associated variable y i in the dual, and each of the n constraints

in the dual has an associated variable x j in the primal. For example, consider the

linear program given in (29.53)–(29.57). The dual of this linear program is

minimize 30y 1 C 24y 2 C 36y 3 (29.86)

subject to

y 1 C 2y 2 C 4y 3 3 (29.87)

y 1 C 2y 2 C y 3 1 (29.88)

3y 1 C 5y 2 C 2y 3 2 (29.89)

y 1 ; y 2 ; y 3 0 : (29.90)

We shall show in Theorem 29.10 that the optimal value of the dual linear pro-

gram is always equal to the optimal value of the primal linear program. Further-

more, the simplex algorithm actually implicitly solves both the primal and the dual

linear programs simultaneously, thereby providing a proof of optimality.

We begin by demonstrating weak duality, which states that any feasible solu-

tion to the primal linear program has a value no greater than that of any feasible

solution to the dual linear program.

Lemma 29.8 (Weak linear-programming duality)

Let Nx be any feasible solution to the primal linear program in (29.16)–(29.18) and

let Ny be any feasible solution to the dual linear program in (29.83)–(29.85). Then,

we have

n X

j D1

c j Nx j

m X

iD1

b i Ny i :

29.4 Duality 881

Proof We have

n X

j D1

c j Nx j

n X

j D1

m X

iD1

a ij Ny i

!

Nx j (by inequalities (29.84))

D

m X

iD1

n X

j D1

a ij Nx j

!

Ny i

m X

iD1

b i Ny i (by inequalities (29.17)) .

Corollary 29.9

Let Nx be a feasible solution to a primal linear program .A; b; c/, and let Ny be a

feasible solution to the corresponding dual linear program. If

n X

j D1

c j Nx j D

m X

iD1

b i Ny i ;

then Nx and Ny are optimal solutions to the primal and dual linear programs, respec-

tively.

Proof By Lemma 29.8, the objective value of a feasible solution to the primal

cannot exceed that of a feasible solution to the dual. The primal linear program is

a maximization problem and the dual is a minimization problem. Thus, if feasible

solutions Nx and Ny have the same objective value, neither can be improved.

Before proving that there always is a dual solution whose value is equal to that

of an optimal primal solution, we describe how to ﬁnd such a solution. When

we ran the simplex algorithm on the linear program in (29.53)–(29.57), the ﬁnal

iteration yielded the slack form (29.72)–(29.75) with objective ´ D 28 x 3 =6

x 5 =62x 6 =3, B D f1; 2; 4g, and N D f3; 5; 6g. As we shall show below, the basic

solution associated with the ﬁnal slack form is indeed an optimal solution to the

linear program; an optimal solution to linear program (29.53)–(29.57) is therefore

. Nx 1 ; Nx 2 ; Nx 3 / D .8; 4; 0/, with objective value .3 8/ C .1 4/ C .2 0/ D 28. As

we also show below, we can read off an optimal dual solution: the negatives of the

coefﬁcients of the primal objective function are the values of the dual variables.

More precisely, suppose that the last slack form of the primal is

´ D

0

C

X

j 2N

c

0

j

x j

x i D b

0

i

X

j 2N

a

0

ij

x j for i 2 B :

882 Chapter 29 Linear Programming

Then, to produce an optimal dual solution, we set

Ny i D

(

c 0

nCi

if .n C i/ 2 N ;

0 otherwise :

(29.91)

Thus, an optimal solution to the dual linear program deﬁned in (29.86)–(29.90)

is Ny 1 D 0 (since n C 1 D 4 2 B), Ny 2 D c 0

5

D 1=6, and Ny 3 D c 0

6

D 2=3.

Evaluating the dual objective function (29.86), we obtain an objective value of

.30 0/ C .24 .1=6// C .36 .2=3// D 28, which conﬁrms that the objective value

of the primal is indeed equal to the objective value of the dual. Combining these

calculations with Lemma 29.8 yields a proof that the optimal objective value of the

primal linear program is 28. We now show that this approach applies in general:

we can ﬁnd an optimal solution to the dual and simultaneously prove that a solution

to the primal is optimal.

Theorem 29.10 (Linear-programming duality)

Suppose that SIMPLEX returns values Nx D . Nx 1 ; Nx 2 ; : : : ; Nx n / for the primal lin-

ear program .A; b; c/. Let N and B denote the nonbasic and basic variables for

the ﬁnal slack form, let c 0

denote the coefﬁcients in the ﬁnal slack form, and let

Ny D . Ny 1 ; Ny 2 ; : : : ; Ny m / be deﬁned by equation (29.91). Then Nx is an optimal so-

lution to the primal linear program, Ny is an optimal solution to the dual linear

program, and

n X

j D1

c j Nx j D

m X

iD1

b i Ny i : (29.92)

Proof By Corollary 29.9, if we can ﬁnd feasible solutions Nx and Ny that satisfy

equation (29.92), then Nx and Ny must be optimal primal and dual solutions. We

shall now show that the solutions Nx and Ny described in the statement of the theorem

satisfy equation (29.92).

Suppose that we run SIMPLEX on a primal linear program, as given in lines

(29.16)–(29.18). The algorithm proceeds through a series of slack forms until it

terminates with a ﬁnal slack form with objective function

´ D

0

C

X

j 2N

c

0

j

x j : (29.93)

Since SIMPLEX terminated with a solution, by the condition in line 3 we know that

c

0

j

0 for all j 2 N : (29.94)

29.4 Duality 883

If we deﬁne

c

0

j

D 0 for all j 2 B ; (29.95)

we can rewrite equation (29.93) as

´ D

0

C

X

j 2N

c

0

j

x j

D

0

C

X

j 2N

c

0

j

x j C

X

j 2B

c

0

j

x j (because c 0

j

D 0 if j 2 B)

D

0

C

nCm X

j D1

c

0

j

x j (because N [ B D f1; 2; : : : ; n C mg) . (29.96)

For the basic solution Nx associated with this ﬁnal slack form, Nx j D 0 for all j 2 N ,

and ´ D 0

. Since all slack forms are equivalent, if we evaluate the original objec-

tive function on Nx, we must obtain the same objective value:

n X

j D1

c j Nx j D

0

C

nCm X

j D1

c

0

j

Nx j (29.97)

D

0

C

X

j 2N

c

0

j

Nx j C

X

j 2B

c

0

j

Nx j

D

0

C

X

j 2N

.c

0

j

0/ C

X

j 2B

.0 Nx j / (29.98)

D

0

:

We shall now show that Ny, deﬁned by equation (29.91), is feasible for the dual

linear program and that its objective value

P m

iD1

b i Ny i equals

P n

j D1

c j Nx j . Equa-

tion (29.97) says that the ﬁrst and last slack forms, evaluated at Nx, are equal. More

generally, the equivalence of all slack forms implies that for any set of values

x D .x 1 ; x 2 ; : : : ; x n /, we have

n X

j D1

c j x j D

0

C

nCm X

j D1

c

0

j

x j :

Therefore, for any particular set of values Nx D . Nx 1 ; Nx 2 ; : : : ; Nx n /, we have

884 Chapter 29 Linear Programming

n X

j D1

c j Nx j

D

0

C

nCm X

j D1

c

0

j

Nx j

D

0

C

n X

j D1

c

0

j

Nx j C

nCm X

j DnC1

c

0

j

Nx j

D

0

C

n X

j D1

c

0

j

Nx j C

m X

iD1

c

0

nCi

Nx nCi

D

0

C

n X

j D1

c

0

j

Nx j C

m X

iD1

. Ny i / Nx nCi (by equations (29.91) and (29.95))

D

0

C

n X

j D1

c

0

j

Nx j C

m X

iD1

. Ny i /

b i

n X

j D1

a ij Nx j

!

(by equation (29.32))

D

0

C

n X

j D1

c

0

j

Nx j

m X

iD1

b i Ny i C

m X

iD1

n X

j D1

.a ij Nx j / Ny i

D

0

C

n X

j D1

c

0

j

Nx j

m X

iD1

b i Ny i C

n X

j D1

m X

iD1

.a ij Ny i / Nx j

D

0

m X

iD1

b i Ny i

!

C

n X

j D1

c

0

j

C

m X

iD1

a ij Ny i

!

Nx j ;

so that

n X

j D1

c j Nx j D

0

m X

iD1

b i Ny i

!

C

n X

j D1

c

0

j

C

m X

iD1

a ij Ny i

!

Nx j : (29.99)

Applying Lemma 29.3 to equation (29.99), we obtain

0

m X

iD1

b i Ny i D 0 ; (29.100)

c

0

j

C

m X

iD1

a ij Ny i D c j for j D 1; 2; : : : ; n : (29.101)

By equation (29.100), we have that

P m

iD1

b i Ny i D 0

, and hence the objective value

of the dual

P m

iD1

b i Ny i

is equal to that of the primal ( 0

). It remains to show

29.4 Duality 885

that the solution Ny is feasible for the dual problem. From inequalities (29.94) and

equations (29.95), we have that c 0

j

0 for all j D 1; 2; : : : ; n C m. Hence, for any

j D 1; 2; : : : ; n, equations (29.101) imply that

c j D c

0

j

C

m X

iD1

a ij Ny i

m X

iD1

a ij Ny i ;

which satisﬁes the constraints (29.84) of the dual. Finally, since c 0

j

0 for each

j 2 N [B, when we set Ny according to equation (29.91), we have that each Ny i 0,

and so the nonnegativity constraints are satisﬁed as well.

We have shown that, given a feasible linear program, if INITIALIZE-SIMPLEX

returns a feasible solution, and if SIMPLEX terminates without returning “un-

bounded,” then the solution returned is indeed an optimal solution. We have also

shown how to construct an optimal solution to the dual linear program.

Exercises

29.4-1

Formulate the dual of the linear program given in Exercise 29.3-5.

29.4-2

Suppose that we have a linear program that is not in standard form. We could

produce the dual by ﬁrst converting it to standard form, and then taking the dual.

It would be more convenient, however, to be able to produce the dual directly.

Explain how we can directly take the dual of an arbitrary linear program.

29.4-3

Write down the dual of the maximum-ﬂow linear program, as given in lines

(29.47)–(29.50) on page 860. Explain how to interpret this formulation as a

minimum-cut problem.

29.4-4

Write down the dual of the minimum-cost-ﬂow linear program, as given in lines

(29.51)–(29.52) on page 862. Explain how to interpret this problem in terms of

graphs and ﬂows.

29.4-5

Show that the dual of the dual of a linear program is the primal linear program.

886 Chapter 29 Linear Programming

29.4-6

Which result from Chapter 26 can be interpreted as weak duality for the maximum-

ﬂow problem?

29.5 The initial basic feasible solution

In this section, we ﬁrst describe how to test whether a linear program is feasible,

and if it is, how to produce a slack form for which the basic solution is feasible.

We conclude by proving the fundamental theorem of linear programming, which

says that the SIMPLEX procedure always produces the correct result.

Finding an initial solution

In Section 29.3, we assumed that we had a procedure INITIALIZE-SIMPLEX that

determines whether a linear program has any feasible solutions, and if it does, gives

a slack form for which the basic solution is feasible. We describe this procedure

here.

A linear program can be feasible, yet the initial basic solution might not be

feasible. Consider, for example, the following linear program:

maximize 2x 1 x 2 (29.102)

subject to

2x 1 x 2 2 (29.103)

x 1 5x 2 4 (29.104)

x 1 ; x 2 0 : (29.105)

If we were to convert this linear program to slack form, the basic solution would

set x 1 D 0 and x 2 D 0. This solution violates constraint (29.104), and so it is not a

feasible solution. Thus, INITIALIZE-SIMPLEX cannot just return the obvious slack

form. In order to determine whether a linear program has any feasible solutions,

we will formulate an auxiliary linear program. For this auxiliary linear program,

we can ﬁnd (with a little work) a slack form for which the basic solution is feasible.

Furthermore, the solution of this auxiliary linear program determines whether the

initial linear program is feasible and if so, it provides a feasible solution with which

we can initialize SIMPLEX.

Lemma 29.11

Let L be a linear program in standard form, given as in (29.16)–(29.18). Let x 0 be

a new variable, and let L aux be the following linear program with n C 1 variables:

29.5 The initial basic feasible solution 887

maximize x 0 (29.106)

subject to

n X

j D1

a ij x j x 0 b i for i D 1; 2; : : : ; m ; (29.107)

x j 0 for j D 0; 1; : : : ; n : (29.108)

Then L is feasible if and only if the optimal objective value of L aux is 0.

Proof Suppose that L has a feasible solution Nx D . Nx 1 ; Nx 2 ; : : : ; Nx n /. Then the

solution Nx 0 D 0 combined with Nx is a feasible solution to L aux with objective

value 0. Since x 0 0 is a constraint of L aux and the objective function is to

maximize x 0 , this solution must be optimal for L aux .

Conversely, suppose that the optimal objective value of L aux is 0. Then Nx 0 D 0,

and the remaining solution values of Nx satisfy the constraints of L.

We now describe our strategy to ﬁnd an initial basic feasible solution for a linear

program L in standard form:

INITIALIZE-SIMPLEX.A; b; c/

1 let k be the index of the minimum b i

2 if b k 0 // is the initial basic solution feasible?

3 return .f1; 2; : : : ; ng ; fn C 1; n C 2; : : : ; n C mg ; A; b; c; 0/

4 form L aux by adding x 0 to the left-hand side of each constraint

and setting the objective function to x 0

5 let .N; B; A; b; c; / be the resulting slack form for L aux

6 l D n C k

7 // L aux has n C 1 nonbasic variables and m basic variables.

8 .N; B; A; b; c; / D PIVOT.N; B; A; b; c; ; l; 0/

9 // The basic solution is now feasible for L aux .

10 iterate the while loop of lines 3–12 of SIMPLEX until an optimal solution

to L aux is found

11 if the optimal solution to L aux sets Nx 0 to 0

12 if Nx 0 is basic

13 perform one (degenerate) pivot to make it nonbasic

14 from the ﬁnal slack form of L aux , remove x 0 from the constraints and

restore the original objective function of L, but replace each basic

variable in this objective function by the right-hand side of its

associated constraint

15 return the modiﬁed ﬁnal slack form

16 else return “infeasible”

888 Chapter 29 Linear Programming

INITIALIZE-SIMPLEX works as follows. In lines 1–3, we implicitly test the

basic solution to the initial slack form for L given by N D f1; 2; : : : ; ng, B D

fn C 1; n C 2; : : : ; n C mg, Nx i D b i for all i 2 B, and Nx j D 0 for all j 2 N .

(Creating the slack form requires no explicit effort, as the values of A, b, and c are

the same in both slack and standard forms.) If line 2 ﬁnds this basic solution to be

feasible—that is, Nx i 0 for all i 2 N [ B—then line 3 returns the slack form.

Otherwise, in line 4, we form the auxiliary linear program L aux as in Lemma 29.11.

Since the initial basic solution to L is not feasible, the initial basic solution to the

slack form for L aux cannot be feasible either. To ﬁnd a basic feasible solution, we

perform a single pivot operation. Line 6 selects l D n C k as the index of the

basic variable that will be the leaving variable in the upcoming pivot operation.

Since the basic variables are x nC1 ; x nC2 ; : : : ; x nCm , the leaving variable x l will be

the one with the most negative value. Line 8 performs that call of PIVOT, with

x 0 entering and x l leaving. We shall see shortly that the basic solution resulting

from this call of PIVOT will be feasible. Now that we have a slack form for which

the basic solution is feasible, we can, in line 10, repeatedly call PIVOT to fully

solve the auxiliary linear program. As the test in line 11 demonstrates, if we ﬁnd

an optimal solution to L aux with objective value 0, then in lines 12–14, we create

a slack form for L for which the basic solution is feasible. To do so, we ﬁrst,

in lines 12–13, handle the degenerate case in which x 0 may still be basic with

value Nx 0 D 0. In this case, we perform a pivot step to remove x 0 from the basis,

using any e 2 N such that a 0e ¤ 0 as the entering variable. The new basic

solution remains feasible; the degenerate pivot does not change the value of any

variable. Next we delete all x 0 terms from the constraints and restore the original

objective function for L. The original objective function may contain both basic

and nonbasic variables. Therefore, in the objective function we replace each basic

variable by the right-hand side of its associated constraint. Line 15 then returns

this modiﬁed slack form. If, on the other hand, line 11 discovers that the original

linear program L is infeasible, then line 16 returns this information.

We now demonstrate the operation of INITIALIZE-SIMPLEX on the linear pro-

gram (29.102)–(29.105). This linear program is feasible if we can ﬁnd nonneg-

ative values for x 1 and x 2 that satisfy inequalities (29.103) and (29.104). Using

Lemma 29.11, we formulate the auxiliary linear program

maximize x 0 (29.109)

subject to

2x 1 x 2 x 0 2 (29.110)

x 1 5x 2 x 0 4 (29.111)

x 1 ; x 2 ; x 0 0 :

By Lemma 29.11, if the optimal objective value of this auxiliary linear program

is 0, then the original linear program has a feasible solution. If the optimal objective

29.5 The initial basic feasible solution 889

value of this auxiliary linear program is negative, then the original linear program

does not have a feasible solution.

We write this linear program in slack form, obtaining

´ D x 0

x 3 D 2 2x 1 C x 2 C x 0

x 4 D 4 x 1 C 5x 2 C x 0 :

We are not out of the woods yet, because the basic solution, which would set

x 4 D 4, is not feasible for this auxiliary linear program. We can, however, with

one call to PIVOT, convert this slack form into one in which the basic solution is

feasible. As line 8 indicates, we choose x 0 to be the entering variable. In line 6, we

choose as the leaving variable x 4 , which is the basic variable whose value in the

basic solution is most negative. After pivoting, we have the slack form

´ D 4 x 1 C 5x 2 x 4

x 0 D 4 C x 1 5x 2 C x 4

x 3 D 6 x 1 4x 2 C x 4 :

The associated basic solution is . Nx 0 ; Nx 1 ; Nx 2 ; Nx 3 ; Nx 4 / D .4; 0; 0; 6; 0/, which is feasi-

ble. We now repeatedly call PIVOT until we obtain an optimal solution to L aux . In

this case, one call to PIVOT with x 2 entering and x 0 leaving yields

´ D x 0

x 2 D

4

5

x 0

5

C

x 1

5

C

x 4

5

x 3 D

14

5

C

4x 0

5

9x 1

5

C

x 4

5

:

This slack form is the ﬁnal solution to the auxiliary problem. Since this solution

has x 0 D 0, we know that our initial problem was feasible. Furthermore, since

x 0 D 0, we can just remove it from the set of constraints. We then restore the

original objective function, with appropriate substitutions made to include only

nonbasic variables. In our example, we get the objective function

2x 1 x 2 D 2x 1

4

5

x 0

5

C

x 1

5

C

x 4

5

:

Setting x 0 D 0 and simplifying, we get the objective function

4

5

C

9x 1

5

x 4

5

;

and the slack form

890 Chapter 29 Linear Programming

´ D

4

5

C

9x 1

5

x 4

5

x 2 D

4

5

C

x 1

5

C

x 4

5

x 3 D

14

5

9x 1

5

C

x 4

5

:

This slack form has a feasible basic solution, and we can return it to procedure

SIMPLEX.

We now formally show the correctness of INITIALIZE-SIMPLEX.

Lemma 29.12

If a linear program L has no feasible solution, then INITIALIZE-SIMPLEX returns

“infeasible.” Otherwise, it returns a valid slack form for which the basic solution

is feasible.

Proof First suppose that the linear program L has no feasible solution. Then by

Lemma 29.11, the optimal objective value of L aux , deﬁned in (29.106)–(29.108),

is nonzero, and by the nonnegativity constraint on x 0 , the optimal objective value

must be negative. Furthermore, this objective value must be ﬁnite, since setting

x i D 0, for i D 1; 2; : : : ; n, and x 0 D jmin

m

iD1

fb i gj is feasible, and this solution

has objective value jmin

m

iD1

fb i gj. Therefore, line 10 of INITIALIZE-SIMPLEX

ﬁnds a solution with a nonpositive objective value. Let Nx be the basic solution

associated with the ﬁnal slack form. We cannot have Nx 0 D 0, because then L aux

would have objective value 0, which contradicts that the objective value is negative.

Thus the test in line 11 results in line 16 returning “infeasible.”

Suppose now that the linear program L does have a feasible solution. From

Exercise 29.3-4, we know that if b i 0 for i D 1; 2; : : : ; m, then the basic solution

associated with the initial slack form is feasible. In this case, lines 2–3 return the

slack form associated with the input. (Converting the standard form to slack form

is easy, since A, b, and c are the same in both.)

In the remainder of the proof, we handle the case in which the linear program is

feasible but we do not return in line 3. We argue that in this case, lines 4–10 ﬁnd a

feasible solution to L aux with objective value 0. First, by lines 1–2, we must have

b k < 0 ;

and

b k b i for each i 2 B : (29.112)

In line 8, we perform one pivot operation in which the leaving variable x l (recall

that l D n C k, so that b l < 0) is the left-hand side of the equation with mini-

mum b i , and the entering variable is x 0 , the extra added variable. We now show

29.5 The initial basic feasible solution 891

that after this pivot, all entries of b are nonnegative, and hence the basic solution

to L aux is feasible. Letting Nx be the basic solution after the call to PIVOT, and

letting

y

b and yB be values returned by PIVOT, Lemma 29.1 implies that

Nx i D

(

b i a ie

y

b e if i 2 yB feg ;

b l =a le if i D e :

(29.113)

The call to PIVOT in line 8 has e D 0. If we rewrite inequalities (29.107), to

include coefﬁcients a i0 ,

n X

j D0

a ij x j b i for i D 1; 2; : : : ; m ; (29.114)

then

a i0 D a ie D 1 for each i 2 B : (29.115)

(Note that a i0 is the coefﬁcient of x 0 as it appears in inequalities (29.114), not

the negation of the coefﬁcient, because L aux is in standard rather than slack form.)

Since l 2 B, we also have that a le D 1. Thus, b l =a le > 0, and so Nx e > 0. For

the remaining basic variables, we have

Nx i D b i a ie

y

b e (by equation (29.113))

D b i a ie .b l =a le / (by line 3 of PIVOT)

D b i b l (by equation (29.115) and a le D 1)

0 (by inequality (29.112)) ,

which implies that each basic variable is now nonnegative. Hence the basic solu-

tion after the call to PIVOT in line 8 is feasible. We next execute line 10, which

solves L aux . Since we have assumed that L has a feasible solution, Lemma 29.11

implies that L aux has an optimal solution with objective value 0. Since all the slack

forms are equivalent, the ﬁnal basic solution to L aux must have Nx 0 D 0, and after

removing x 0 from the linear program, we obtain a slack form that is feasible for L.

Line 15 then returns this slack form.

Fundamental theorem of linear programming

We conclude this chapter by showing that the SIMPLEX procedure works. In par-

ticular, any linear program either is infeasible, is unbounded, or has an optimal

solution with a ﬁnite objective value. In each case, SIMPLEX acts appropriately.

892 Chapter 29 Linear Programming

Theorem 29.13 (Fundamental theorem of linear programming)

Any linear program L, given in standard form, either

1. has an optimal solution with a ﬁnite objective value,

2. is infeasible, or

3. is unbounded.

If L is infeasible, SIMPLEX returns “infeasible.” If L is unbounded, SIMPLEX

returns “unbounded.” Otherwise, SIMPLEX returns an optimal solution with a ﬁnite

objective value.

Proof By Lemma 29.12, if linear program L is infeasible, then SIMPLEX returns

“infeasible.” Now suppose that the linear program L is feasible. By Lemma 29.12,

INITIALIZE-SIMPLEX returns a slack form for which the basic solution is feasible.

By Lemma 29.7, therefore, SIMPLEX either returns “unbounded” or terminates

with a feasible solution. If it terminates with a ﬁnite solution, then Theorem 29.10

tells us that this solution is optimal. On the other hand, if SIMPLEX returns “un-

bounded,” Lemma 29.2 tells us the linear program L is indeed unbounded. Since

SIMPLEX always terminates in one of these ways, the proof is complete.

Exercises

29.5-1

Give detailed pseudocode to implement lines 5 and 14 of INITIALIZE-SIMPLEX.

29.5-2

Show that when the main loop of SIMPLEX is run by INITIALIZE-SIMPLEX, it can

never return “unbounded.”

29.5-3

Suppose that we are given a linear program L in standard form, and suppose that

for both L and the dual of L, the basic solutions associated with the initial slack

forms are feasible. Show that the optimal objective value of L is 0.

29.5-4

Suppose that we allow strict inequalities in a linear program. Show that in this

case, the fundamental theorem of linear programming does not hold.

29.5 The initial basic feasible solution 893

29.5-5

Solve the following linear program using SIMPLEX:

maximize x 1 C 3x 2

subject to

x 1 x 2 8

x 1 x 2 3

x 1 C 4x 2 2

x 1 ; x 2 0 :

29.5-6

Solve the following linear program using SIMPLEX:

maximize x 1 2x 2

subject to

x 1 C 2x 2 4

2x 1 6x 2 12

x 2 1

x 1 ; x 2 0 :

29.5-7

Solve the following linear program using SIMPLEX:

maximize x 1 C 3x 2

subject to

x 1 C x 2 1

x 1 x 2 3

x 1 C 4x 2 2

x 1 ; x 2 0 :

29.5-8

Solve the linear program given in (29.6)–(29.10).

29.5-9

Consider the following 1-variable linear program, which we call P :

maximize tx

subject to

rx s

x 0 ;

where r, s, and t are arbitrary real numbers. Let D be the dual of P .

894 Chapter 29 Linear Programming

State for which values of r, s, and t you can assert that

1. Both P and D have optimal solutions with ﬁnite objective values.

2. P is feasible, but D is infeasible.

3. D is feasible, but P is infeasible.

4. Neither P nor D is feasible.

Problems

29-1 Linear-inequality feasibility

Given a set of m linear inequalities on n variables x 1 ; x 2 ; : : : ; x n , the linear-

inequality feasibility problem asks whether there is a setting of the variables that

simultaneously satisﬁes each of the inequalities.

a. Show that if we have an algorithm for linear programming, we can use it to

solve a linear-inequality feasibility problem. The number of variables and con-

straints that you use in the linear-programming problem should be polynomial

in n and m.

b. Show that if we have an algorithm for the linear-inequality feasibility problem,

we can use it to solve a linear-programming problem. The number of variables

and linear inequalities that you use in the linear-inequality feasibility problem

should be polynomial in n and m, the number of variables and constraints in

the linear program.

29-2 Complementary slackness

Complementary slackness describes a relationship between the values of primal

variables and dual constraints and between the values of dual variables and pri-

mal constraints. Let Nx be a feasible solution to the primal linear program given

in (29.16)–(29.18), and let Ny be a feasible solution to the dual linear program given

in (29.83)–(29.85). Complementary slackness states that the following conditions

are necessary and sufﬁcient for Nx and Ny to be optimal:

m X

iD1

a ij Ny i D c j or Nx j D 0 for j D 1; 2; : : : ; n

and

n X

j D1

a ij Nx j D b i or Ny i D 0 for i D 1; 2; : : : ; m :

Problems for Chapter 29 895

a. Verify that complementary slackness holds for the linear program in lines

(29.53)–(29.57).

b. Prove that complementary slackness holds for any primal linear program and

its corresponding dual.

c. Prove that a feasible solution Nx to a primal linear program given in lines

(29.16)–(29.18) is optimal if and only if there exist values Ny D . Ny 1 ; Ny 2 ; : : : ; Ny m /

such that

1. Ny is a feasible solution to the dual linear program given in (29.83)–(29.85),

2.

P m

iD1

a ij Ny i D c j for all j such that Nx j > 0, and

3. Ny i D 0 for all i such that

P n

j D1

a ij Nx j < b i .

29-3 Integer linear programming

An integer linear-programming problem is a linear-programming problem with

the additional constraint that the variables x must take on integral values. Exer-

cise 34.5-3 shows that just determining whether an integer linear program has a

feasible solution is NP-hard, which means that there is no known polynomial-time

algorithm for this problem.

a. Show that weak duality (Lemma 29.8) holds for an integer linear program.

b. Show that duality (Theorem 29.10) does not always hold for an integer linear

program.

c. Given a primal linear program in standard form, let us deﬁne P to be the opti-

mal objective value for the primal linear program, D to be the optimal objective

value for its dual, IP to be the optimal objective value for the integer version of

the primal (that is, the primal with the added constraint that the variables take

on integer values), and ID to be the optimal objective value for the integer ver-

sion of the dual. Assuming that both the primal integer program and the dual

integer program are feasible and bounded, show that

IP P D D ID :

29-4 Farkas’s lemma

Let A be an m n matrix and c be an n-vector. Then Farkas’s lemma states that

exactly one of the systems

896 Chapter 29 Linear Programming

Ax 0 ;

c T x > 0

and

A T y D c ;

y 0

is solvable, where x is an n-vector and y is an m-vector. Prove Farkas’s lemma.

29-5 Minimum-cost circulation

In this problem, we consider a variant of the minimum-cost-ﬂow problem from

Section 29.2 in which we are not given a demand, a source, or a sink. Instead,

we are given, as before, a ﬂow network and edge costs a.u; /. A ﬂow is feasible

if it satisﬁes the capacity constraint on every edge and ﬂow conservation at every

vertex. The goal is to ﬁnd, among all feasible ﬂows, the one of minimum cost. We

call this problem the minimum-cost-circulation problem.

a. Formulate the minimum-cost-circulation problem as a linear program.

b. Suppose that for all edges .u; / 2 E, we have a.u; / > 0. Characterize an

optimal solution to the minimum-cost-circulation problem.

c. Formulate the maximum-ﬂow problem as a minimum-cost-circulation problem

linear program. That is given a maximum-ﬂow problem instance G D .V; E/

with source s, sink t and edge capacities c, create a minimum-cost-circulation

problem by giving a (possibly different) network G 0 D .V 0 ; E 0 / with edge

capacities c 0

and edge costs a 0

such that you can discern a solution to the

maximum-ﬂow problem from a solution to the minimum-cost-circulation prob-

lem.

d. Formulate the single-source shortest-path problem as a minimum-cost-circu-

lation problem linear program.

Chapter notes

This chapter only begins to study the wide ﬁeld of linear programming. A num-

ber of books are devoted exclusively to linear programming, including those by

Chv´atal [69], Gass [130], Karloff [197], Schrijver [303], and Vanderbei [344].

Many other books give a good coverage of linear programming, including those

by Papadimitriou and Steiglitz [271] and Ahuja, Magnanti, and Orlin [7]. The

coverage in this chapter draws on the approach taken by Chv´atal.

Notes for Chapter 29 897

The simplex algorithm for linear programming was invented by G. Dantzig

in 1947. Shortly after, researchers discovered how to formulate a number of prob-

lems in a variety of ﬁelds as linear programs and solve them with the simplex

algorithm. As a result, applications of linear programming ﬂourished, along with

several algorithms. Variants of the simplex algorithm remain the most popular

methods for solving linear-programming problems. This history appears in a num-

ber of places, including the notes in [69] and [197].

The ellipsoid algorithm was the ﬁrst polynomial-time algorithm for linear pro-

gramming and is due to L. G. Khachian in 1979; it was based on earlier work by

N. Z. Shor, D. B. Judin, and A. S. Nemirovskii. Gr¨otschel, Lov´asz, and Schrijver

[154] describe how to use the ellipsoid algorithm to solve a variety of problems in

combinatorial optimization. To date, the ellipsoid algorithm does not appear to be

competitive with the simplex algorithm in practice.

Karmarkar’s paper [198] includes a description of the ﬁrst interior-point algo-

rithm. Many subsequent researchers designed interior-point algorithms. Good sur-

veys appear in the article of Goldfarb and Todd [141] and the book by Ye [361].

Analysis of the simplex algorithm remains an active area of research. V. Klee

and G. J. Minty constructed an example on which the simplex algorithm runs

through 2 n 1 iterations. The simplex algorithm usually performs very well in

practice and many researchers have tried to give theoretical justiﬁcation for this

empirical observation. A line of research begun by K. H. Borgwardt, and carried

on by many others, shows that under certain probabilistic assumptions on the in-

put, the simplex algorithm converges in expected polynomial time. Spielman and

Teng [322] made progress in this area, introducing the “smoothed analysis of algo-

rithms” and applying it to the simplex algorithm.

The simplex algorithm is known to run efﬁciently in certain special cases. Par-

ticularly noteworthy is the network-simplex algorithm, which is the simplex al-

gorithm, specialized to network-ﬂow problems. For certain network problems,

including the shortest-paths, maximum-ﬂow, and minimum-cost-ﬂow problems,

variants of the network-simplex algorithm run in polynomial time. See, for exam-

ple, the article by Orlin [268] and the citations therein.

30 Polynomials and the FFT

The straightforward method of adding two polynomials of degree n takes ‚.n/

time, but the straightforward method of multiplying them takes ‚.n 2 / time. In this

chapter, we shall show how the fast Fourier transform, or FFT, can reduce the time

to multiply polynomials to ‚.n lg n/.

The most common use for Fourier transforms, and hence the FFT, is in signal

processing. A signal is given in the time domain: as a function mapping time to

amplitude. Fourier analysis allows us to express the signal as a weighted sum of

phase-shifted sinusoids of varying frequencies. The weights and phases associated

with the frequencies characterize the signal in the frequency domain. Among the

many everyday applications of FFT’s are compression techniques used to encode

digital video and audio information, including MP3 ﬁles. Several ﬁne books delve

into the rich area of signal processing; the chapter notes reference a few of them.

Polynomials

A polynomial in the variable x over an algebraic ﬁeld F represents a function A.x/

as a formal sum:

A.x/ D

n1 X

j D0

a j x

j

:

We call the values a 0 ; a 1 ; : : : ; a n1 the coefﬁcients of the polynomial. The co-

efﬁcients are drawn from a ﬁeld F , typically the set C of complex numbers. A

polynomial A.x/ has degree k if its highest nonzero coefﬁcient is a k ; we write

that degree.A/ D k. Any integer strictly greater than the degree of a polynomial

is a degree-bound of that polynomial. Therefore, the degree of a polynomial of

degree-bound n may be any integer between 0 and n 1, inclusive.

We can deﬁne a variety of operations on polynomials. For polynomial addi-

tion, if A.x/ and B.x/ are polynomials of degree-bound n, their sum is a polyno-

Chapter 30 Polynomials and the FFT 899

mial C.x/, also of degree-bound n, such that C.x/ D A.x/ C B.x/ for all x in the

underlying ﬁeld. That is, if

A.x/ D

n1 X

j D0

a j x

j

and

B.x/ D

n1 X

j D0

b j x

j

;

then

C.x/ D

n1 X

j D0

c j x

j

;

where c j D a j C b j for j D 0; 1; : : : ; n 1. For example, if we have the

polynomials A.x/ D 6x 3 C 7x 2 10x C 9 and B.x/ D 2x 3 C 4x 5, then

C.x/ D 4x 3 C 7x 2 6x C 4.

For polynomial multiplication, if A.x/ and B.x/ are polynomials of degree-

bound n, their product C.x/ is a polynomial of degree-bound 2n 1 such that

C.x/ D A.x/B.x/ for all x in the underlying ﬁeld. You probably have multi-

plied polynomials before, by multiplying each term in A.x/ by each term in B.x/

and then combining terms with equal powers. For example, we can multiply

A.x/ D 6x 3 C 7x 2 10x C 9 and B.x/ D 2x 3 C 4x 5 as follows:

6x 3 C 7x 2 10x C 9

2x 3 C 4x 5

30x 3 35x 2 C 50x 45

24x 4 C 28x 3 40x 2 C 36x

12x 6 14x 5 C 20x 4 18x 3

12x 6 14x 5 C 44x 4 20x 3 75x 2 C 86x 45

Another way to express the product C.x/ is

C.x/ D

2n2 X

j D0

c j x

j

; (30.1)

where

c j D

j X

kD0

a k b j k : (30.2)

900 Chapter 30 Polynomials and the FFT

Note that degree.C / D degree.A/ C degree.B/, implying that if A is a polyno-

mial of degree-bound n a and B is a polynomial of degree-bound n b , then C is a

polynomial of degree-bound n a C n b 1. Since a polynomial of degree-bound k

is also a polynomial of degree-bound k C 1, we will normally say that the product

polynomial C is a polynomial of degree-bound n a C n b .

Chapter outline

Section 30.1 presents two ways to represent polynomials: the coefﬁcient represen-

tation and the point-value representation. The straightforward methods for multi-

plying polynomials—equations (30.1) and (30.2)—take ‚.n 2 / time when we rep-

resent polynomials in coefﬁcient form, but only ‚.n/ time when we represent them

in point-value form. We can, however, multiply polynomials using the coefﬁcient

representation in only ‚.n lg n/ time by converting between the two representa-

tions. To see why this approach works, we must ﬁrst study complex roots of unity,

which we do in Section 30.2. Then, we use the FFT and its inverse, also described

in Section 30.2, to perform the conversions. Section 30.3 shows how to implement

the FFT quickly in both serial and parallel models.

This chapter uses complex numbers extensively, and within this chapter we use

the symbol i exclusively to denote

p

1.

30.1 Representing polynomials

The coefﬁcient and point-value representations of polynomials are in a sense equiv-

alent; that is, a polynomial in point-value form has a unique counterpart in co-

efﬁcient form. In this section, we introduce the two representations and show

how to combine them so that we can multiply two degree-bound n polynomials

in ‚.n lg n/ time.

Coefﬁcient representation

A coefﬁcient representation of a polynomial A.x/ D

P n1

j D0

a j x j

of degree-

bound n is a vector of coefﬁcients a D .a 0 ; a 1 ; : : : ; a n1 /. In matrix equations

in this chapter, we shall generally treat vectors as column vectors.

The coefﬁcient representation is convenient for certain operations on polyno-

mials. For example, the operation of evaluating the polynomial A.x/ at a given

point x 0 consists of computing the value of A.x 0 /. We can evaluate a polynomial

in ‚.n/ time using Horner’s rule:

A.x 0 / D a 0 C x 0 .a 1 C x 0 .a 2 C C x 0 .a n2 C x 0 .a n1 // // :

30.1 Representing polynomials 901

Similarly, adding two polynomials represented by the coefﬁcient vectors a D

.a 0 ; a 1 ; : : : ; a n1 / and b D .b 0 ; b 1 ; : : : ; b n1 / takes ‚.n/ time: we just produce

the coefﬁcient vector c D .c 0 ; c 1 ; : : : ; c n1 /, where c j D a j C b j for j D

0; 1; : : : ; n 1.

Now, consider multiplying two degree-bound n polynomials A.x/ and B.x/ rep-

resented in coefﬁcient form. If we use the method described by equations (30.1)

and (30.2), multiplying polynomials takes time ‚.n 2 /, since we must multiply

each coefﬁcient in the vector a by each coefﬁcient in the vector b. The operation

of multiplying polynomials in coefﬁcient form seems to be considerably more difﬁ-

cult than that of evaluating a polynomial or adding two polynomials. The resulting

coefﬁcient vector c, given by equation (30.2), is also called the convolution of the

input vectors a and b, denoted c D a ˝ b. Since multiplying polynomials and

computing convolutions are fundamental computational problems of considerable

practical importance, this chapter concentrates on efﬁcient algorithms for them.

Point-value representation

A point-value representation of a polynomial A.x/ of degree-bound n is a set of

n point-value pairs

f.x 0 ; y 0 /; .x 1 ; y 1 /; : : : ; .x n1 ; y n1 /g

such that all of the x k are distinct and

y k D A.x k / (30.3)

for k D 0; 1; : : : ; n 1. A polynomial has many different point-value representa-

tions, since we can use any set of n distinct points x 0 ; x 1 ; : : : ; x n1 as a basis for

the representation.

Computing a point-value representation for a polynomial given in coefﬁcient

form is in principle straightforward, since all we have to do is select n distinct

points x 0 ; x 1 ; : : : ; x n1 and then evaluate A.x k / for k D 0; 1; : : : ; n 1. With

Horner’s method, evaluating a polynomial at n points takes time ‚.n 2 /. We shall

see later that if we choose the points x k cleverly, we can accelerate this computation

to run in time ‚.n lg n/.

The inverse of evaluation—determining the coefﬁcient form of a polynomial

from a point-value representation—is interpolation. The following theorem shows

that interpolation is well deﬁned when the desired interpolating polynomial must

have a degree-bound equal to the given number of point-value pairs.

Theorem 30.1 (Uniqueness of an interpolating polynomial)

For any set f.x 0 ; y 0 /; .x 1 ; y 1 /; : : : ; .x n1 ; y n1 /g of n point-value pairs such that

all the x k values are distinct, there is a unique polynomial A.x/ of degree-bound n

such that y k D A.x k / for k D 0; 1; : : : ; n 1.

902 Chapter 30 Polynomials and the FFT

Proof The proof relies on the existence of the inverse of a certain matrix. Equa-

tion (30.3) is equivalent to the matrix equation ˙

1 x 0 x 2

0

x n1

0

1 x 1 x 2

1

x n1

1

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

1 x n1 x 2

n1

x n1

n1

˙

a 0

a 1

:

:

:

a n1

D

˙

y 0

y 1

:

:

:

y n1

: (30.4)

The matrix on the left is denoted V.x 0 ; x 1 ; : : : ; x n1 / and is known as a Vander-

monde matrix. By Problem D-1, this matrix has determinant

Y

0j <kn1

.x k x j / ;

and therefore, by Theorem D.5, it is invertible (that is, nonsingular) if the x k are

distinct. Thus, we can solve for the coefﬁcients a j uniquely given the point-value

representation:

a D V.x 0 ; x 1 ; : : : ; x n1 /

1

y :

The proof of Theorem 30.1 describes an algorithm for interpolation based on

solving the set (30.4) of linear equations. Using the LU decomposition algorithms

of Chapter 28, we can solve these equations in time O.n 3 /.

A faster algorithm for n-point interpolation is based on Lagrange’s formula:

A.x/ D

n1 X

kD0

y k

Y

j ¤k

.x x j /

Y

j ¤k

.x k x j /

: (30.5)

You may wish to verify that the right-hand side of equation (30.5) is a polynomial

of degree-bound n that satisﬁes A.x k / D y k for all k. Exercise 30.1-5 asks you

how to compute the coefﬁcients of A using Lagrange’s formula in time ‚.n 2 /.

Thus, n-point evaluation and interpolation are well-deﬁned inverse operations

that transform between the coefﬁcient representation of a polynomial and a point-

value representation. 1 The algorithms described above for these problems take

time ‚.n 2 /.

The point-value representation is quite convenient for many operations on poly-

nomials. For addition, if C.x/ D A.x/ C B.x/, then C.x k / D A.x k / C B.x k / for

any point x k . More precisely, if we have a point-value representation for A,

1

Interpolation is a notoriously tricky problem from the point of view of numerical stability. Although

the approaches described here are mathematically correct, small differences in the inputs or round-off

errors during computation can cause large differences in the result.

30.1 Representing polynomials 903

f.x 0 ; y 0 /; .x 1 ; y 1 /; : : : ; .x n1 ; y n1 /g ;

and for B,

f.x 0 ; y

0

0

/; .x 1 ; y

0

1

/; : : : ; .x n1 ; y

0

n1

/g

(note that A and B are evaluated at the same n points), then a point-value repre-

sentation for C is

f.x 0 ; y 0 C y

0

0

/; .x 1 ; y 1 C y

0

1

/; : : : ; .x n1 ; y n1 C y

0

n1

/g :

Thus, the time to add two polynomials of degree-bound n in point-value form

is ‚.n/.

Similarly, the point-value representation is convenient for multiplying polyno-

mials. If C.x/ D A.x/B.x/, then C.x k / D A.x k /B.x k / for any point x k , and

we can pointwise multiply a point-value representation for A by a point-value rep-

resentation for B to obtain a point-value representation for C . We must face the

problem, however, that degree.C / D degree.A/ C degree.B/; if A and B are of

degree-bound n, then C is of degree-bound 2n. A standard point-value represen-

tation for A and B consists of n point-value pairs for each polynomial. When we

multiply these together, we get n point-value pairs, but we need 2n pairs to interpo-

late a unique polynomial C of degree-bound 2n. (See Exercise 30.1-4.) We must

therefore begin with “extended” point-value representations for A and for B con-

sisting of 2n point-value pairs each. Given an extended point-value representation

for A,

f.x 0 ; y 0 /; .x 1 ; y 1 /; : : : ; .x 2n1 ; y 2n1 /g ;

and a corresponding extended point-value representation for B,

f.x 0 ; y

0

0

/; .x 1 ; y

0

1

/; : : : ; .x 2n1 ; y

0

2n1

/g ;

then a point-value representation for C is

f.x 0 ; y 0 y

0

0

/; .x 1 ; y 1 y

0

1

/; : : : ; .x 2n1 ; y 2n1 y

0

2n1

/g :

Given two input polynomials in extended point-value form, we see that the time to

multiply them to obtain the point-value form of the result is ‚.n/, much less than

the time required to multiply polynomials in coefﬁcient form.

Finally, we consider how to evaluate a polynomial given in point-value form at a

new point. For this problem, we know of no simpler approach than converting the

polynomial to coefﬁcient form ﬁrst, and then evaluating it at the new point.

Fast multiplication of polynomials in coefﬁcient form

Can we use the linear-time multiplication method for polynomials in point-value

form to expedite polynomial multiplication in coefﬁcient form? The answer hinges

904 Chapter 30 Polynomials and the FFT

a 0 ; a 1 ; : : : ; a n1

b 0 ; b 1 ; : : : ; b n1

c 0 ; c 1 ; : : : ; c 2n2

Ordinary multiplication

Time ‚.n 2 /

Evaluation

Time ‚.n lg n/ Time ‚.n lg n/

Interpolation

Pointwise multiplication

Time ‚.n/

A.! 0

2n

/; B.! 0

2n

/

A.! 1

2n

/; B.! 1

2n

/

A.! 2n1

2n

/; B.! 2n1

2n

/

:

:

:

:

:

:

C.! 0

2n

/

C.! 1

2n

/

C.! 2n1

2n

/

Coefﬁcient

Point-value

representations

representations

Figure 30.1 A graphical outline of an efﬁcient polynomial-multiplication process. Representations

on the top are in coefﬁcient form, while those on the bottom are in point-value form. The arrows

from left to right correspond to the multiplication operation. The !2n terms are complex .2n/th roots

of unity.

on whether we can convert a polynomial quickly from coefﬁcient form to point-

value form (evaluate) and vice versa (interpolate).

We can use any points we want as evaluation points, but by choosing the eval-

uation points carefully, we can convert between representations in only ‚.n lg n/

time. As we shall see in Section 30.2, if we choose “complex roots of unity” as

the evaluation points, we can produce a point-value representation by taking the

discrete Fourier transform (or DFT) of a coefﬁcient vector. We can perform the

inverse operation, interpolation, by taking the “inverse DFT” of point-value pairs,

yielding a coefﬁcient vector. Section 30.2 will show how the FFT accomplishes

the DFT and inverse DFT operations in ‚.n lg n/ time.

Figure 30.1 shows this strategy graphically. One minor detail concerns degree-

bounds. The product of two polynomials of degree-bound n is a polynomial of

degree-bound 2n. Before evaluating the input polynomials A and B, therefore,

we ﬁrst double their degree-bounds to 2n by adding n high-order coefﬁcients of 0.

Because the vectors have 2n elements, we use “complex .2n/th roots of unity,”

which are denoted by the ! 2n terms in Figure 30.1.

Given the FFT, we have the following ‚.n lg n/-time procedure for multiplying

two polynomials A.x/ and B.x/ of degree-bound n, where the input and output

representations are in coefﬁcient form. We assume that n is a power of 2; we can

always meet this requirement by adding high-order zero coefﬁcients.

1. Double degree-bound: Create coefﬁcient representations of A.x/ and B.x/ as

degree-bound 2n polynomials by adding n high-order zero coefﬁcients to each.

30.1 Representing polynomials 905

2. Evaluate: Compute point-value representations of A.x/ and B.x/ of length 2n

by applying the FFT of order 2n on each polynomial. These representations

contain the values of the two polynomials at the .2n/th roots of unity.

3. Pointwise multiply: Compute a point-value representation for the polynomial

C.x/ D A.x/B.x/ by multiplying these values together pointwise. This repre-

sentation contains the value of C.x/ at each .2n/th root of unity.

4. Interpolate: Create the coefﬁcient representation of the polynomial C.x/ by

applying the FFT on 2n point-value pairs to compute the inverse DFT.

Steps (1) and (3) take time ‚.n/, and steps (2) and (4) take time ‚.n lg n/. Thus,

once we show how to use the FFT, we will have proven the following.

Theorem 30.2

We can multiply two polynomials of degree-bound n in time ‚.n lg n/, with both

the input and output representations in coefﬁcient form.

Exercises

30.1-1

Multiply the polynomials A.x/ D 7x 3 x 2 C x 10 and B.x/ D 8x 3 6x C 3

using equations (30.1) and (30.2).

30.1-2

Another way to evaluate a polynomial A.x/ of degree-bound n at a given point x 0

is to divide A.x/ by the polynomial .x x 0 /, obtaining a quotient polynomial q.x/

of degree-bound n 1 and a remainder r, such that

A.x/ D q.x/.x x 0 / C r :

Clearly, A.x 0 / D r. Show how to compute the remainder r and the coefﬁcients

of q.x/ in time ‚.n/ from x 0 and the coefﬁcients of A.

30.1-3

Derive a point-value representation for A rev .x/ D

P n1

j D0

a n1j x j

from a point-

value representation for A.x/ D

P n1

j D0

a j x j

, assuming that none of the points is 0.

30.1-4

Prove that n distinct point-value pairs are necessary to uniquely specify a polyno-

mial of degree-bound n, that is, if fewer than n distinct point-value pairs are given,

they fail to specify a unique polynomial of degree-bound n. (Hint: Using Theo-

rem 30.1, what can you say about a set of n 1 point-value pairs to which you add

one more arbitrarily chosen point-value pair?)

906 Chapter 30 Polynomials and the FFT

30.1-5

Show how to use equation (30.5) to interpolate in time ‚.n 2 /. (Hint: First compute

the coefﬁcient representation of the polynomial

Q

j

.x x j / and then divide by

.x x k / as necessary for the numerator of each term; see Exercise 30.1-2. You can

compute each of the n denominators in time O.n/.)

30.1-6

Explain what is wrong with the “obvious” approach to polynomial division using

a point-value representation, i.e., dividing the corresponding y values. Discuss

separately the case in which the division comes out exactly and the case in which

it doesn’t.

30.1-7

Consider two sets A and B, each having n integers in the range from 0 to 10n. We

wish to compute the Cartesian sum of A and B, deﬁned by

C D fx C y W x 2 A and y 2 Bg :

Note that the integers in C are in the range from 0 to 20n. We want to ﬁnd the

elements of C and the number of times each element of C is realized as a sum of

elements in A and B. Show how to solve the problem in O.n lg n/ time. (Hint:

Represent A and B as polynomials of degree at most 10n.)

30.2 The DFT and FFT

In Section 30.1, we claimed that if we use complex roots of unity, we can evaluate

and interpolate polynomials in ‚.n lg n/ time. In this section, we deﬁne complex

roots of unity and study their properties, deﬁne the DFT, and then show how the

FFT computes the DFT and its inverse in ‚.n lg n/ time.

Complex roots of unity

A complex nth root of unity is a complex number ! such that

!

n

D 1 :

There are exactly n complex nth roots of unity: e 2ik=n

for k D 0; 1; : : : ; n 1.

To interpret this formula, we use the deﬁnition of the exponential of a complex

number:

e

iu

D cos.u/ C i sin.u/ :

Figure 30.2 shows that the n complex roots of unity are equally spaced around the

circle of unit radius centered at the origin of the complex plane. The value

30.2 The DFT and FFT 907

1 1

i

i

! 0

8

D ! 8

8

! 1

8

! 2

8

! 3

8

! 4

8

! 5

8

! 6

8

! 7

8

Figure 30.2 The values of !

0

8

; !

1

8

; : : : ; !

7

8

in the complex plane, where !8 D e

2i=8

is the prin-

cipal 8th root of unity.

! n D e

2i=n

(30.6)

is the principal nth root of unity; 2 all other complex nth roots of unity are powers

of ! n .

The n complex nth roots of unity,

!

0

n

; !

1

n

; : : : ; !

n1

n

;

form a group under multiplication (see Section 31.3). This group has the same

structure as the additive group . Z n ; C/ modulo n, since ! n

n

D ! 0

n

D 1 implies that

! j

n

! k

n

D ! j Ck

n

D ! .j Ck/ mod n

n

. Similarly, ! 1

n

D ! n1

n

. The following lemmas

furnish some essential properties of the complex nth roots of unity.

Lemma 30.3 (Cancellation lemma)

For any integers n 0, k 0, and d > 0,

!

dk

dn

D !

k

n

: (30.7)

Proof The lemma follows directly from equation (30.6), since

!

dk

dn

D

e

2i=dn

dk

D

e

2i=n

k

D !

k

n

:

2

Many other authors deﬁne !n differently: !n D e

2i=n

. This alternative deﬁnition tends to be

used for signal-processing applications. The underlying mathematics is substantially the same with

either deﬁnition of !n.

908 Chapter 30 Polynomials and the FFT

Corollary 30.4

For any even integer n > 0,

!

n=2

n

D ! 2 D 1 :

Proof The proof is left as Exercise 30.2-1.

Lemma 30.5 (Halving lemma)

If n > 0 is even, then the squares of the n complex nth roots of unity are the n=2

complex .n=2/th roots of unity.

Proof By the cancellation lemma, we have .! k

n

/ 2 D ! k

n=2

, for any nonnegative

integer k. Note that if we square all of the complex nth roots of unity, then we

obtain each .n=2/th root of unity exactly twice, since

.!

kCn=2

n

/

2

D !

2kCn

n

D !

2k

n

!

n

n

D !

2k

n

D .!

k

n

/

2

:

Thus, ! k

n

and ! kCn=2

n

have the same square. We could also have used Corol-

lary 30.4 to prove this property, since ! n=2

n

D 1 implies ! kCn=2

n

D ! k

n

, and

thus .! kCn=2

n

/ 2 D .! k

n

/ 2

.

As we shall see, the halving lemma is essential to our divide-and-conquer ap-

proach for converting between coefﬁcient and point-value representations of poly-

nomials, since it guarantees that the recursive subproblems are only half as large.

Lemma 30.6 (Summation lemma)

For any integer n 1 and nonzero integer k not divisible by n,

n1 X

j D0

!

k

n

j

D 0 :

Proof Equation (A.5) applies to complex values as well as to reals, and so we

have

30.2 The DFT and FFT 909

n1 X

j D0

!

k

n

j

D

.! k

n

/ n 1

! k

n

1

D

.! n

n

/ k 1

! k

n

1

D

.1/ k 1

! k

n

1

D 0 :

Because we require that k is not divisible by n, and because ! k

n

D 1 only when k

is divisible by n, we ensure that the denominator is not 0.

The DFT

Recall that we wish to evaluate a polynomial

A.x/ D

n1 X

j D0

a j x

j

of degree-bound n at ! 0

n

; ! 1

n

; ! 2

n

; : : : ; ! n1

n

(that is, at the n complex nth roots of

unity). 3 We assume that A is given in coefﬁcient form: a D .a 0 ; a 1 ; : : : ; a n1 /. Let

us deﬁne the results y k , for k D 0; 1; : : : ; n 1, by

y k D A.!

k

n

/

D

n1 X

j D0

a j !

kj

n

: (30.8)

The vector y D .y 0 ; y 1 ; : : : ; y n1 / is the discrete Fourier transform (DFT) of the

coefﬁcient vector a D .a 0 ; a 1 ; : : : ; a n1 /. We also write y D DFT n .a/.

The FFT

By using a method known as the fast Fourier transform (FFT), which takes ad-

vantage of the special properties of the complex roots of unity, we can compute

DFT n .a/ in time ‚.n lg n/, as opposed to the ‚.n 2 / time of the straightforward

method. We assume throughout that n is an exact power of 2. Although strategies

3

The length n is actually what we referred to as 2n in Section 30.1, since we double the degree-bound

of the given polynomials prior to evaluation. In the context of polynomial multiplication, therefore,

we are actually working with complex .2n/th roots of unity.

910 Chapter 30 Polynomials and the FFT

for dealing with non-power-of-2 sizes are known, they are beyond the scope of this

book.

The FFT method employs a divide-and-conquer strategy, using the even-indexed

and odd-indexed coefﬁcients of A.x/ separately to deﬁne the two new polynomials

A Œ0 .x/ and A Œ1 .x/ of degree-bound n=2:

A

Œ0

.x/ D a 0 C a 2 x C a 4 x

2

C C a n2 x

n=21

;

A

Œ1

.x/ D a 1 C a 3 x C a 5 x

2

C C a n1 x

n=21

:

Note that A Œ0

contains all the even-indexed coefﬁcients of A (the binary represen-

tation of the index ends in 0) and A Œ1

contains all the odd-indexed coefﬁcients (the

binary representation of the index ends in 1). It follows that

A.x/ D A

Œ0

.x

2

/ C xA

Œ1

.x

2

/ ; (30.9)

so that the problem of evaluating A.x/ at ! 0

n

; ! 1

n

; : : : ; ! n1

n

reduces to

1. evaluating the degree-bound n=2 polynomials A Œ0 .x/ and A Œ1 .x/ at the points

.!

0

n

/

2

; .!

1

n

/

2

; : : : ; .!

n1

n

/

2

; (30.10)

and then

2. combining the results according to equation (30.9).

By the halving lemma, the list of values (30.10) consists not of n distinct val-

ues but only of the n=2 complex .n=2/th roots of unity, with each root occurring

exactly twice. Therefore, we recursively evaluate the polynomials A Œ0

and A Œ1

of degree-bound n=2 at the n=2 complex .n=2/th roots of unity. These subprob-

lems have exactly the same form as the original problem, but are half the size.

We have now successfully divided an n-element DFT n computation into two n=2-

element DFT n=2 computations. This decomposition is the basis for the follow-

ing recursive FFT algorithm, which computes the DFT of an n-element vector

a D .a 0 ; a 1 ; : : : ; a n1 /, where n is a power of 2.

30.2 The DFT and FFT 911

RECURSIVE-FFT.a/

1 n D a:length // n is a power of 2

2 if n == 1

3 return a

4 ! n D e 2i=n

5 ! D 1

6 a Œ0 D .a 0 ; a 2 ; : : : ; a n2 /

7 a Œ1 D .a 1 ; a 3 ; : : : ; a n1 /

8 y Œ0 D RECURSIVE-FFT.a Œ0 /

9 y Œ1 D RECURSIVE-FFT.a Œ1 /

10 for k D 0 to n=2 1

11 y k D y

Œ0

k

C ! y

Œ1

k

12 y kC.n=2/ D y

Œ0

k

! y

Œ1

k

13 ! D ! ! n

14 return y // y is assumed to be a column vector

The RECURSIVE-FFT procedure works as follows. Lines 2–3 represent the basis

of the recursion; the DFT of one element is the element itself, since in this case

y 0 D a 0 !

0

1

D a 0 1

D a 0 :

Lines 6–7 deﬁne the coefﬁcient vectors for the polynomials A Œ0

and A Œ1

. Lines

4, 5, and 13 guarantee that ! is updated properly so that whenever lines 11–12

are executed, we have ! D ! k

n

. (Keeping a running value of ! from iteration

to iteration saves time over computing ! k

n

from scratch each time through the for

loop.) Lines 8–9 perform the recursive DFT n=2 computations, setting, for k D

0; 1; : : : ; n=2 1,

y

Œ0

k

D A

Œ0

.!

k

n=2

/ ;

y

Œ1

k

D A

Œ1

.!

k

n=2

/ ;

or, since ! k

n=2

D ! 2k

n

by the cancellation lemma,

y

Œ0

k

D A

Œ0

.!

2k

n

/ ;

y

Œ1

k

D A

Œ1

.!

2k

n

/ :

912 Chapter 30 Polynomials and the FFT

Lines 11–12 combine the results of the recursive DFT n=2 calculations. For y 0 ; y 1 ;

: : : ; y n=21 , line 11 yields

y k D y

Œ0

k

C !

k

n

y

Œ1

k

D A

Œ0

.!

2k

n

/ C !

k

n

A

Œ1

.!

2k

n

/

D A.!

k

n

/ (by equation (30.9)) .

For y n=2 ; y n=2C1 ; : : : ; y n1 , letting k D 0; 1; : : : ; n=2 1, line 12 yields

y kC.n=2/ D y

Œ0

k

!

k

n

y

Œ1

k

D y

Œ0

k

C !

kC.n=2/

n

y

Œ1

k

(since ! kC.n=2/

n

D ! k

n

)

D A

Œ0

.!

2k

n

/ C !

kC.n=2/

n

A

Œ1

.!

2k

n

/

D A

Œ0

.!

2kCn

n

/ C !

kC.n=2/

n

A

Œ1

.!

2kCn

n

/ (since ! 2kCn

n

D ! 2k

n

)

D A.!

kC.n=2/

n

/ (by equation (30.9)) .

Thus, the vector y returned by RECURSIVE-FFT is indeed the DFT of the input

vector a.

Lines 11 and 12 multiply each value y

Œ1

k

by ! k

n

, for k D 0; 1; : : : ; n=2 1.

Line 11 adds this product to y

Œ0

k

, and line 12 subtracts it. Because we use each

factor ! k

n

in both its positive and negative forms, we call the factors ! k

n

twiddle

factors.

To determine the running time of procedure RECURSIVE-FFT, we note that

exclusive of the recursive calls, each invocation takes time ‚.n/, where n is the

length of the input vector. The recurrence for the running time is therefore

T .n/ D 2T .n=2/ C ‚.n/

D ‚.n lg n/ :

Thus, we can evaluate a polynomial of degree-bound n at the complex nth roots of

unity in time ‚.n lg n/ using the fast Fourier transform.

Interpolation at the complex roots of unity

We now complete the polynomial multiplication scheme by showing how to in-

terpolate the complex roots of unity by a polynomial, which enables us to convert

from point-value form back to coefﬁcient form. We interpolate by writing the DFT

as a matrix equation and then looking at the form of the matrix inverse.

From equation (30.4), we can write the DFT as the matrix product y D V n a,

where V n is a Vandermonde matrix containing the appropriate powers of ! n :

30.2 The DFT and FFT 913 y 0

y 1

y 2

y 3

:

:

:

y n1

D

1 1 1 1 1

1 ! n ! 2

n

! 3

n

! n1

n

1 ! 2

n

! 4

n

! 6

n

! 2.n1/

n

1 ! 3

n

! 6

n

! 9

n

! 3.n1/

n

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

1 ! n1

n

! 2.n1/

n

! 3.n1/

n

! .n1/.n1/

n

a 0

a 1

a 2

a 3

:

:

:

a n1

:

The .k; j / entry of V n is ! kj

n

, for j; k D 0; 1; : : : ; n 1. The exponents of the

entries of V n form a multiplication table.

For the inverse operation, which we write as a D DFT

1

n

.y/, we proceed by

multiplying y by the matrix V 1

n

, the inverse of V n .

Theorem 30.7

For j; k D 0; 1; : : : ; n 1, the .j; k/ entry of V 1

n

is ! kj

n

=n.

Proof We show that V 1

n

V n D I n , the n n identity matrix. Consider the .j; j 0 /

entry of V 1

n

V n :

ŒV

1

n

V n jj 0 D

n1 X

kD0

.!

kj

n

=n/.!

kj

0

n

/

D

n1 X

kD0

!

k.j

0

j /

n

=n :

This summation equals 1 if j 0 D j , and it is 0 otherwise by the summation lemma

(Lemma 30.6). Note that we rely on .n 1/ j 0 j n 1, so that j 0 j is

not divisible by n, in order for the summation lemma to apply.

Given the inverse matrix V 1

n

, we have that DFT

1

n

.y/ is given by

a j D

1

n

n1 X

kD0

y k !

kj

n

(30.11)

for j D 0; 1; : : : ; n 1. By comparing equations (30.8) and (30.11), we see that

by modifying the FFT algorithm to switch the roles of a and y, replace ! n by ! 1

n

,

and divide each element of the result by n, we compute the inverse DFT (see Ex-

ercise 30.2-4). Thus, we can compute DFT

1

n

in ‚.n lg n/ time as well.

We see that, by using the FFT and the inverse FFT, we can transform a poly-

nomial of degree-bound n back and forth between its coefﬁcient representation

and a point-value representation in time ‚.n lg n/. In the context of polynomial

multiplication, we have shown the following.

914 Chapter 30 Polynomials and the FFT

Theorem 30.8 (Convolution theorem)

For any two vectors a and b of length n, where n is a power of 2,

a ˝ b D DFT

1

2n

.DFT 2n .a/ DFT 2n .b// ;

where the vectors a and b are padded with 0s to length 2n and denotes the com-

ponentwise product of two 2n-element vectors.

Exercises

30.2-1

Prove Corollary 30.4.

30.2-2

Compute the DFT of the vector .0; 1; 2; 3/.

30.2-3

Do Exercise 30.1-1 by using the ‚.n lg n/-time scheme.

30.2-4

Write pseudocode to compute DFT

1

n

in ‚.n lg n/ time.

30.2-5

Describe the generalization of the FFT procedure to the case in which n is a power

of 3. Give a recurrence for the running time, and solve the recurrence.

30.2-6 ?

Suppose that instead of performing an n-element FFT over the ﬁeld of complex

numbers (where n is even), we use the ring Z m of integers modulo m, where

m D 2 tn=2 C 1 and t is an arbitrary positive integer. Use ! D 2 t

instead of ! n

as a principal nth root of unity, modulo m. Prove that the DFT and the inverse DFT

are well deﬁned in this system.

30.2-7

Given a list of values ´ 0 ; ´ 1 ; : : : ; ´ n1 (possibly with repetitions), show how to ﬁnd

the coefﬁcients of a polynomial P.x/ of degree-bound n C 1 that has zeros only

at ´ 0 ; ´ 1 ; : : : ; ´ n1 (possibly with repetitions). Your procedure should run in time

O.n lg

2

n/. (Hint: The polynomial P.x/ has a zero at ´ j if and only if P.x/ is a

multiple of .x ´ j /.)

30.2-8 ?

The chirp transform of a vector a D .a 0 ; a 1 ; : : : ; a n1 / is the vector y D

.y 0 ; y 1 ; : : : ; y n1 /, where y k D

P n1

j D0

a j ´ kj

and ´ is any complex number. The

30.3 Efﬁcient FFT implementations 915

DFT is therefore a special case of the chirp transform, obtained by taking ´ D ! n .

Show how to evaluate the chirp transform in time O.n lg n/ for any complex num-

ber ´. (Hint: Use the equation

y k D ´

k

2

=2

n1 X

j D0

a j ´

j

2

=2

´

.kj /

2

=2

to view the chirp transform as a convolution.)

30.3 Efﬁcient FFT implementations

Since the practical applications of the DFT, such as signal processing, demand the

utmost speed, this section examines two efﬁcient FFT implementations. First, we

shall examine an iterative version of the FFT algorithm that runs in ‚.n lg n/ time

but can have a lower constant hidden in the ‚-notation than the recursive version

in Section 30.2. (Depending on the exact implementation, the recursive version

may use the hardware cache more efﬁciently.) Then, we shall use the insights that

led us to the iterative implementation to design an efﬁcient parallel FFT circuit.

An iterative FFT implementation

We ﬁrst note that the for loop of lines 10–13 of RECURSIVE-FFT involves com-

puting the value ! k

n

y

Œ1

k

twice. In compiler terminology, we call such a value a

common subexpression. We can change the loop to compute it only once, storing

it in a temporary variable t.

for k D 0 to n=2 1

t D ! y

Œ1

k

y k D y

Œ0

k

C t

y kC.n=2/ D y

Œ0

k

t

! D ! ! n

The operation in this loop, multiplying the twiddle factor ! D ! k

n

by y

Œ1

k

, storing

the product into t, and adding and subtracting t from y

Œ0

k

, is known as a butterﬂy

operation and is shown schematically in Figure 30.3.

We now show how to make the FFT algorithm iterative rather than recursive

in structure. In Figure 30.4, we have arranged the input vectors to the recursive

calls in an invocation of RECURSIVE-FFT in a tree structure, where the initial

call is for n D 8. The tree has one node for each call of the procedure, labeled

916 Chapter 30 Polynomials and the FFT

+

– •

(a) (b)

y

Œ0

k

y

Œ0

k

y

Œ1

k

y

Œ1

k

! k

n

! k

n

y

Œ0

k

C ! k

n

y

Œ1

k

y

Œ0

k

C ! k

n

y

Œ1

k

y

Œ0

k

! k

n

y

Œ1

k

y

Œ0

k

! k

n

y

Œ1

k

Figure 30.3 A butterﬂy operation. (a) The two input values enter from the left, the twiddle fac-

tor !

k

n

is multiplied by y

Œ1

k

, and the sum and difference are output on the right. (b) A simpliﬁed

drawing of a butterﬂy operation. We will use this representation in a parallel FFT circuit.

(a

0

,a

1

,a

2

,a

3

,a

4

,a

5

,a

6

,a

7

)

(a

0

,a

2

,a

4

,a

6

)

(a

0

,a

4

) (a

2

,a

6

)

(a

0

) (a

4

) (a

2

) (a

6

)

(a

1

,a

3

,a

5

,a

7

)

(a

1

,a

5

)

(a

1

) (a

5

)

(a

3

,a

7

)

(a

3

) (a

7

)

Figure 30.4 The tree of input vectors to the recursive calls of the RECURSIVE-FFT procedure. The

initial invocation is for n D 8.

by the corresponding input vector. Each RECURSIVE-FFT invocation makes two

recursive calls, unless it has received a 1-element vector. The ﬁrst call appears in

the left child, and the second call appears in the right child.

Looking at the tree, we observe that if we could arrange the elements of the

initial vector a into the order in which they appear in the leaves, we could trace

the execution of the RECURSIVE-FFT procedure, but bottom up instead of top

down. First, we take the elements in pairs, compute the DFT of each pair using

one butterﬂy operation, and replace the pair with its DFT. The vector then holds

n=2 2-element DFTs. Next, we take these n=2 DFTs in pairs and compute the

DFT of the four vector elements they come from by executing two butterﬂy oper-

ations, replacing two 2-element DFTs with one 4-element DFT. The vector then

holds n=4 4-element DFTs. We continue in this manner until the vector holds two

.n=2/-element DFTs, which we combine using n=2 butterﬂy operations into the

ﬁnal n-element DFT.

To turn this bottom-up approach into code, we use an array AŒ0 : : n 1 that

initially holds the elements of the input vector a in the order in which they appear

30.3 Efﬁcient FFT implementations 917

in the leaves of the tree of Figure 30.4. (We shall show later how to determine this

order, which is known as a bit-reversal permutation.) Because we have to combine

DFTs on each level of the tree, we introduce a variable s to count the levels, ranging

from 1 (at the bottom, when we are combining pairs to form 2-element DFTs)

to lg n (at the top, when we are combining two .n=2/-element DFTs to produce the

ﬁnal result). The algorithm therefore has the following structure:

1 for s D 1 to lg n

2 for k D 0 to n 1 by 2 s

3 combine the two 2 s1

-element DFTs in

AŒk : : k C 2 s1 1 and AŒk C 2 s1 : : k C 2 s 1

into one 2 s

-element DFT in AŒk : : k C 2 s 1

We can express the body of the loop (line 3) as more precise pseudocode. We

copy the for loop from the RECURSIVE-FFT procedure, identifying y Œ0

with

AŒk : : k C 2 s1 1 and y Œ1

with AŒk C 2 s1 : : k C 2 s 1. The twiddle fac-

tor used in each butterﬂy operation depends on the value of s; it is a power of ! m ,

where m D 2 s

. (We introduce the variable m solely for the sake of readability.)

We introduce another temporary variable u that allows us to perform the butterﬂy

operation in place. When we replace line 3 of the overall structure by the loop

body, we get the following pseudocode, which forms the basis of the parallel im-

plementation we shall present later. The code ﬁrst calls the auxiliary procedure

BIT-REVERSE-COPY.a; A/ to copy vector a into array A in the initial order in

which we need the values.

ITERATIVE-FFT.a/

1 BIT-REVERSE-COPY.a; A/

2 n D a:length // n is a power of 2

3 for s D 1 to lg n

4 m D 2 s

5 ! m D e 2i=m

6 for k D 0 to n 1 by m

7 ! D 1

8 for j D 0 to m=2 1

9 t D ! AŒk C j C m=2

10 u D AŒk C j

11 AŒk C j D u C t

12 AŒk C j C m=2 D u t

13 ! D ! ! m

14 return A

How does BIT-REVERSE-COPY get the elements of the input vector a into the

desired order in the array A? The order in which the leaves appear in Figure 30.4

918 Chapter 30 Polynomials and the FFT

is a bit-reversal permutation. That is, if we let rev.k/ be the lg n-bit integer

formed by reversing the bits of the binary representation of k, then we want to

place vector element a k in array position AŒrev.k/. In Figure 30.4, for exam-

ple, the leaves appear in the order 0; 4; 2; 6; 1; 5; 3; 7; this sequence in binary is

000; 100; 010; 110; 001; 101; 011; 111, and when we reverse the bits of each value

we get the sequence 000; 001; 010; 011; 100; 101; 110; 111. To see that we want a

bit-reversal permutation in general, we note that at the top level of the tree, indices

whose low-order bit is 0 go into the left subtree and indices whose low-order bit

is 1 go into the right subtree. Stripping off the low-order bit at each level, we con-

tinue this process down the tree, until we get the order given by the bit-reversal

permutation at the leaves.

Since we can easily compute the function rev.k/, the BIT-REVERSE-COPY pro-

cedure is simple:

BIT-REVERSE-COPY.a; A/

1 n D a:length

2 for k D 0 to n 1

3 AŒrev.k/ D a k

The iterative FFT implementation runs in time ‚.n lg n/. The call to BIT-

REVERSE-COPY.a; A/ certainly runs in O.n lg n/ time, since we iterate n times

and can reverse an integer between 0 and n 1, with lg n bits, in O.lg n/ time.

(In practice, because we usually know the initial value of n in advance, we would

probably code a table mapping k to rev.k/, making BIT-REVERSE-COPY run in

‚.n/ time with a low hidden constant. Alternatively, we could use the clever amor-

tized reverse binary counter scheme described in Problem 17-1.) To complete the

proof that ITERATIVE-FFT runs in time ‚.n lg n/, we show that L.n/, the number

of times the body of the innermost loop (lines 8–13) executes, is ‚.n lg n/. The

for loop of lines 6–13 iterates n=m D n=2 s

times for each value of s, and the

innermost loop of lines 8–13 iterates m=2 D 2 s1

times. Thus,

L.n/ D

lg n X

sD1

n

2 s

2

s1

D

lg n X

sD1

n

2

D ‚.n lg n/ :

30.3 Efﬁcient FFT implementations 919

a 0

a 1

a 2

a 3

a 4

a 5

a 6

a 7

y 0

y 1

y 2

y 3

y 4

y 5

y 6

y 7

stage s D 1 stage s D 2 stage s D 3

! 0

2

! 0

2

! 0

2

! 0

2

! 0

4

! 0

4

! 1

4

! 1

4

! 0

8

! 1

8

! 2

8

! 3

8

Figure 30.5 A circuit that computes the FFT in parallel, here shown on n D 8 inputs. Each

butterﬂy operation takes as input the values on two wires, along with a twiddle factor, and it produces

as outputs the values on two wires. The stages of butterﬂies are labeled to correspond to iterations

of the outermost loop of the ITERATIVE-FFT procedure. Only the top and bottom wires passing

through a butterﬂy interact with it; wires that pass through the middle of a butterﬂy do not affect

that butterﬂy, nor are their values changed by that butterﬂy. For example, the top butterﬂy in stage 2

has nothing to do with wire 1 (the wire whose output is labeled y1); its inputs and outputs are only

on wires 0 and 2 (labeled y0 and y2, respectively). This circuit has depth ‚.lg n/ and performs

‚.n lg n/ butterﬂy operations altogether.

A parallel FFT circuit

We can exploit many of the properties that allowed us to implement an efﬁcient

iterative FFT algorithm to produce an efﬁcient parallel algorithm for the FFT. We

will express the parallel FFT algorithm as a circuit. Figure 30.5 shows a parallel

FFT circuit, which computes the FFT on n inputs, for n D 8. The circuit begins

with a bit-reverse permutation of the inputs, followed by lg n stages, each stage

consisting of n=2 butterﬂies executed in parallel. The depth of the circuit—the

maximum number of computational elements between any output and any input

that can reach it—is therefore ‚.lg n/.

The leftmost part of the parallel FFT circuit performs the bit-reverse permuta-

tion, and the remainder mimics the iterative ITERATIVE-FFT procedure. Because

each iteration of the outermost for loop performs n=2 independent butterﬂy opera-

tions, the circuit performs them in parallel. The value of s in each iteration within

920 Chapter 30 Polynomials and the FFT

ITERATIVE-FFT corresponds to a stage of butterﬂies shown in Figure 30.5. For

s D 1; 2; : : : ; lg n, stage s consists of n=2 s

groups of butterﬂies (corresponding to

each value of k in ITERATIVE-FFT), with 2 s1

butterﬂies per group (corresponding

to each value of j in ITERATIVE-FFT). The butterﬂies shown in Figure 30.5 corre-

spond to the butterﬂy operations of the innermost loop (lines 9–12 of ITERATIVE-

FFT). Note also that the twiddle factors used in the butterﬂies correspond to those

used in ITERATIVE-FFT: in stage s, we use ! 0

m

; ! 1

m

; : : : ; ! m=21

m

, where m D 2 s

.

Exercises

30.3-1

Show how ITERATIVE-FFT computes the DFT of the input vector .0; 2; 3; 1; 4;

5; 7; 9/.

30.3-2

Show how to implement an FFT algorithm with the bit-reversal permutation occur-

ring at the end, rather than at the beginning, of the computation. (Hint: Consider

the inverse DFT.)

30.3-3

How many times does ITERATIVE-FFT compute twiddle factors in each stage?

Rewrite ITERATIVE-FFT to compute twiddle factors only 2 s1

times in stage s.

30.3-4 ?

Suppose that the adders within the butterﬂy operations of the FFT circuit some-

times fail in such a manner that they always produce a zero output, independent

of their inputs. Suppose that exactly one adder has failed, but that you don’t know

which one. Describe how you can identify the failed adder by supplying inputs to

the overall FFT circuit and observing the outputs. How efﬁcient is your method?

Problems

30-1 Divide-and-conquer multiplication

a. Show how to multiply two linear polynomials ax C b and cx C d using only

three multiplications. (Hint: One of the multiplications is .a C b/ .c C d/.)

b. Give two divide-and-conquer algorithms for multiplying two polynomials of

degree-bound n in ‚.n lg 3 / time. The ﬁrst algorithm should divide the input

polynomial coefﬁcients into a high half and a low half, and the second algorithm

should divide them according to whether their index is odd or even.

Problems for Chapter 30 921

c. Show how to multiply two n-bit integers in O.n lg 3 / steps, where each step

operates on at most a constant number of 1-bit values.

30-2 Toeplitz matrices

A Toeplitz matrix is an n n matrix A D .a ij / such that a ij D a i1;j 1 for

i D 2; 3; : : : ; n and j D 2; 3; : : : ; n.

a. Is the sum of two Toeplitz matrices necessarily Toeplitz? What about the prod-

uct?

b. Describe how to represent a Toeplitz matrix so that you can add two n n

Toeplitz matrices in O.n/ time.

c. Give an O.n lg n/-time algorithm for multiplying an n n Toeplitz matrix by a

vector of length n. Use your representation from part (b).

d. Give an efﬁcient algorithm for multiplying two n n Toeplitz matrices. Analyze

its running time.

30-3 Multidimensional fast Fourier transform

We can generalize the 1-dimensional discrete Fourier transform deﬁned by equa-

tion (30.8) to d dimensions. The input is a d-dimensional array A D .a j 1 ;j 2 ;:::;j d /

whose dimensions are n 1 ; n 2 ; : : : ; n d , where n 1 n 2 n d D n. We deﬁne the

d-dimensional discrete Fourier transform by the equation

y k 1 ;k 2 ;:::;k d D

n 1 1 X

j 1 D0

n 2 1 X

j 2 D0

n d 1 X

j d D0

a j 1 ;j 2 ;:::;j d !

j 1 k 1

n 1

!

j 2 k 2

n 2

!

j d k d

n d

for 0 k 1 < n 1 , 0 k 2 < n 2 , . . . , 0 k d < n d .

a. Show that we can compute a d-dimensional DFT by computing 1-dimensional

DFTs on each dimension in turn. That is, we ﬁrst compute n=n 1 separate

1-dimensional DFTs along dimension 1. Then, using the result of the DFTs

along dimension 1 as the input, we compute n=n 2 separate 1-dimensional DFTs

along dimension 2. Using this result as the input, we compute n=n 3 separate

1-dimensional DFTs along dimension 3, and so on, through dimension d.

b. Show that the ordering of dimensions does not matter, so that we can compute

a d-dimensional DFT by computing the 1-dimensional DFTs in any order of

the d dimensions.

922 Chapter 30 Polynomials and the FFT

c. Show that if we compute each 1-dimensional DFT by computing the fast Four-

ier transform, the total time to compute a d-dimensional DFT is O.n lg n/,

independent of d.

30-4 Evaluating all derivatives of a polynomial at a point

Given a polynomial A.x/ of degree-bound n, we deﬁne its tth derivative by

A

.t/

.x/ D

„

A.x/ if t D 0 ;

d

dx

A

.t1/

.x/ if 1 t n 1 ;

0 if t n :

From the coefﬁcient representation .a 0 ; a 1 ; : : : ; a n1 / of A.x/ and a given point x 0 ,

we wish to determine A .t/ .x 0 / for t D 0; 1; : : : ; n 1.

a. Given coefﬁcients b 0 ; b 1 ; : : : ; b n1 such that

A.x/ D

n1 X

j D0

b j .x x 0 /

j

;

show how to compute A .t/ .x 0 /, for t D 0; 1; : : : ; n 1, in O.n/ time.

b. Explain how to ﬁnd b 0 ; b 1 ; : : : ; b n1 in O.n lg n/ time, given A.x 0 C ! k

n

/ for

k D 0; 1; : : : ; n 1.

c. Prove that

A.x 0 C !

k

n

/ D

n1 X

rD0

! kr

n

rŠ

n1 X

j D0

f .j /g.r j /

!

;

where f .j / D a j j Š and

g.l/ D

(

x l

0

=.l/Š if .n 1/ l 0 ;

0 if 1 l n 1 :

d. Explain how to evaluate A.x 0 C ! k

n

/ for k D 0; 1; : : : ; n 1 in O.n lg n/

time. Conclude that we can evaluate all nontrivial derivatives of A.x/ at x 0 in

O.n lg n/ time.

Problems for Chapter 30 923

30-5 Polynomial evaluation at multiple points

We have seen how to evaluate a polynomial of degree-bound n at a single point in

O.n/ time using Horner’s rule. We have also discovered how to evaluate such a

polynomial at all n complex roots of unity in O.n lg n/ time using the FFT. We

shall now show how to evaluate a polynomial of degree-bound n at n arbitrary

points in O.n lg

2

n/ time.

To do so, we shall assume that we can compute the polynomial remainder when

one such polynomial is divided by another in O.n lg n/ time, a result that we state

without proof. For example, the remainder of 3x 3 C x 2 3x C 1 when divided by

x 2 C x C 2 is

.3x

3

C x

2

3x C 1/ mod .x

2

C x C 2/ D 7x C 5 :

Given the coefﬁcient representation of a polynomial A.x/ D

P n1

kD0

a k x k

and

n points x 0 ; x 1 ; : : : ; x n1 , we wish to compute the n values A.x 0 /; A.x 1 /; : : : ;

A.x n1 /. For 0 i j n 1, deﬁne the polynomials P ij .x/ D

Q j

kDi

.x x k /

and Q ij .x/ D A.x/ mod P ij .x/. Note that Q ij .x/ has degree at most j i.

a. Prove that A.x/ mod .x ´/ D A.´/ for any point ´.

b. Prove that Q kk .x/ D A.x k / and that Q 0;n1 .x/ D A.x/.

c. Prove that for i k j , we have Q ik .x/ D Q ij .x/ mod P ik .x/ and

Q kj .x/ D Q ij .x/ mod P kj .x/.

d. Give an O.n lg

2

n/-time algorithm to evaluate A.x 0 /; A.x 1 /; : : : ; A.x n1 /.

30-6 FFT using modular arithmetic

As deﬁned, the discrete Fourier transform requires us to compute with complex

numbers, which can result in a loss of precision due to round-off errors. For some

problems, the answer is known to contain only integers, and by using a variant of

the FFT based on modular arithmetic, we can guarantee that the answer is calcu-

lated exactly. An example of such a problem is that of multiplying two polynomials

with integer coefﬁcients. Exercise 30.2-6 gives one approach, using a modulus of

length .n/ bits to handle a DFT on n points. This problem gives another ap-

proach, which uses a modulus of the more reasonable length O.lg n/; it requires

that you understand the material of Chapter 31. Let n be a power of 2.

a. Suppose that we search for the smallest k such that p D kn C 1 is prime. Give

a simple heuristic argument why we might expect k to be approximately ln n.

(The value of k might be much larger or smaller, but we can reasonably expect

to examine O.lg n/ candidate values of k on average.) How does the expected

length of p compare to the length of n?

924 Chapter 30 Polynomials and the FFT

Let g be a generator of Z

p

, and let w D g k

mod p.

b. Argue that the DFT and the inverse DFT are well-deﬁned inverse operations

modulo p, where w is used as a principal nth root of unity.

c. Show how to make the FFT and its inverse work modulo p in time O.n lg n/,

where operations on words of O.lg n/ bits take unit time. Assume that the

algorithm is given p and w.

d. Compute the DFT modulo p D 17 of the vector .0; 5; 3; 7; 7; 2; 1; 6/. Note that

g D 3 is a generator of Z

17

.

Chapter notes

Van Loan’s book [343] provides an outstanding treatment of the fast Fourier trans-

form. Press, Teukolsky, Vetterling, and Flannery [283, 284] have a good descrip-

tion of the fast Fourier transform and its applications. For an excellent introduction

to signal processing, a popular FFT application area, see the texts by Oppenheim

and Schafer [266] and Oppenheim and Willsky [267]. The Oppenheim and Schafer

book also shows how to handle cases in which n is not an integer power of 2.

Fourier analysis is not limited to 1-dimensional data. It is widely used in image

processing to analyze data in 2 or more dimensions. The books by Gonzalez and

Woods [146] and Pratt [281] discuss multidimensional Fourier transforms and their

use in image processing, and books by Tolimieri, An, and Lu [338] and Van Loan

[343] discuss the mathematics of multidimensional fast Fourier transforms.

Cooley and Tukey [76] are widely credited with devising the FFT in the 1960s.

The FFT had in fact been discovered many times previously, but its importance was

not fully realized before the advent of modern digital computers. Although Press,

Teukolsky, Vetterling, and Flannery attribute the origins of the method to Runge

and K¨onig in 1924, an article by Heideman, Johnson, and Burrus [163] traces the

history of the FFT as far back as C. F. Gauss in 1805.

Frigo and Johnson [117] developed a fast and ﬂexible implementation of the

FFT, called FFTW (“fastest Fourier transform in the West”). FFTW is designed for

situations requiring multiple DFT computations on the same problem size. Before

actually computing the DFTs, FFTW executes a “planner,” which, by a series of

trial runs, determines how best to decompose the FFT computation for the given

problem size on the host machine. FFTW adapts to use the hardware cache ef-

ﬁciently, and once subproblems are small enough, FFTW solves them with opti-

mized, straight-line code. Furthermore, FFTW has the unusual advantage of taking

‚.n lg n/ time for any problem size n, even when n is a large prime.

Notes for Chapter 30 925

Although the standard Fourier transform assumes that the input represents points

that are uniformly spaced in the time domain, other techniques can approximate the

FFT on “nonequispaced” data. The article by Ware [348] provides an overview.

31 Number-Theoretic Algorithms

Number theory was once viewed as a beautiful but largely useless subject in pure

mathematics. Today number-theoretic algorithms are used widely, due in large part

to the invention of cryptographic schemes based on large prime numbers. These

schemes are feasible because we can ﬁnd large primes easily, and they are secure

because we do not know how to factor the product of large primes (or solve related

problems, such as computing discrete logarithms) efﬁciently. This chapter presents

some of the number theory and related algorithms that underlie such applications.

Section 31.1 introduces basic concepts of number theory, such as divisibility,

modular equivalence, and unique factorization. Section 31.2 studies one of the

world’s oldest algorithms: Euclid’s algorithm for computing the greatest common

divisor of two integers. Section 31.3 reviews concepts of modular arithmetic. Sec-

tion 31.4 then studies the set of multiples of a given number a, modulo n, and shows

how to ﬁnd all solutions to the equation ax b .mod n/ by using Euclid’s algo-

rithm. The Chinese remainder theorem is presented in Section 31.5. Section 31.6

considers powers of a given number a, modulo n, and presents a repeated-squaring

algorithm for efﬁciently computing a b

mod n, given a, b, and n. This operation is

at the heart of efﬁcient primality testing and of much modern cryptography. Sec-

tion 31.7 then describes the RSA public-key cryptosystem. Section 31.8 examines

a randomized primality test. We can use this test to ﬁnd large primes efﬁciently,

which we need to do in order to create keys for the RSA cryptosystem. Finally,

Section 31.9 reviews a simple but effective heuristic for factoring small integers. It

is a curious fact that factoring is one problem people may wish to be intractable,

since the security of RSA depends on the difﬁculty of factoring large integers.

Size of inputs and cost of arithmetic computations

Because we shall be working with large integers, we need to adjust how we think

about the size of an input and about the cost of elementary arithmetic operations.

In this chapter, a “large input” typically means an input containing “large in-

tegers” rather than an input containing “many integers” (as for sorting). Thus,

31.1 Elementary number-theoretic notions 927

we shall measure the size of an input in terms of the number of bits required to

represent that input, not just the number of integers in the input. An algorithm

with integer inputs a 1 ; a 2 ; : : : ; a k is a polynomial-time algorithm if it runs in time

polynomial in lg a 1 ; lg a 2 ; : : : ; lg a k , that is, polynomial in the lengths of its binary-

encoded inputs.

In most of this book, we have found it convenient to think of the elemen-

tary arithmetic operations (multiplications, divisions, or computing remainders)

as primitive operations that take one unit of time. By counting the number of such

arithmetic operations that an algorithm performs, we have a basis for making a

reasonable estimate of the algorithm’s actual running time on a computer. Elemen-

tary operations can be time-consuming, however, when their inputs are large. It

thus becomes convenient to measure how many bit operations a number-theoretic

algorithm requires. In this model, multiplying two ˇ-bit integers by the ordinary

method uses ‚.ˇ 2 / bit operations. Similarly, we can divide a ˇ-bit integer by a

shorter integer or take the remainder of a ˇ-bit integer when divided by a shorter in-

teger in time ‚.ˇ 2 / by simple algorithms. (See Exercise 31.1-12.) Faster methods

are known. For example, a simple divide-and-conquer method for multiplying two

ˇ-bit integers has a running time of ‚.ˇ lg 3 /, and the fastest known method has

a running time of ‚.ˇ lg ˇ lg lg ˇ/. For practical purposes, however, the ‚.ˇ 2 /

algorithm is often best, and we shall use this bound as a basis for our analyses.

We shall generally analyze algorithms in this chapter in terms of both the number

of arithmetic operations and the number of bit operations they require.

31.1 Elementary number-theoretic notions

This section provides a brief review of notions from elementary number theory

concerning the set Z D f: : : ; 2; 1; 0; 1; 2; : : :g of integers and the set N D

f0; 1; 2; : : :g of natural numbers.

Divisibility and divisors

The notion of one integer being divisible by another is key to the theory of numbers.

The notation d j a (read “d divides a”) means that a D kd for some integer k.

Every integer divides 0. If a > 0 and d j a, then jdj jaj. If d j a, then we also

say that a is a multiple of d. If d does not divide a, we write d − a.

If d j a and d 0, we say that d is a divisor of a. Note that d j a if and only

if d j a, so that no generality is lost by deﬁning the divisors to be nonnegative,

with the understanding that the negative of any divisor of a also divides a. A

928 Chapter 31 Number-Theoretic Algorithms

divisor of a nonzero integer a is at least 1 but not greater than jaj. For example, the

divisors of 24 are 1, 2, 3, 4, 6, 8, 12, and 24.

Every positive integer a is divisible by the trivial divisors 1 and a. The nontrivial

divisors of a are the factors of a. For example, the factors of 20 are 2, 4, 5, and 10.

Prime and composite numbers

An integer a > 1 whose only divisors are the trivial divisors 1 and a is a prime

number or, more simply, a prime. Primes have many special properties and play a

critical role in number theory. The ﬁrst 20 primes, in order, are

2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59; 61; 67; 71 :

Exercise 31.1-2 asks you to prove that there are inﬁnitely many primes. An integer

a > 1 that is not prime is a composite number or, more simply, a composite. For

example, 39 is composite because 3 j 39. We call the integer 1 a unit, and it is

neither prime nor composite. Similarly, the integer 0 and all negative integers are

neither prime nor composite.

The division theorem, remainders, and modular equivalence

Given an integer n, we can partition the integers into those that are multiples of n

and those that are not multiples of n. Much number theory is based upon reﬁning

this partition by classifying the nonmultiples of n according to their remainders

when divided by n. The following theorem provides the basis for this reﬁnement.

We omit the proof (but see, for example, Niven and Zuckerman [265]).

Theorem 31.1 (Division theorem)

For any integer a and any positive integer n, there exist unique integers q and r

such that 0 r < n and a D qn C r.

The value q D ba=nc is the quotient of the division. The value r D a mod n

is the remainder (or residue) of the division. We have that n j a if and only if

a mod n D 0.

We can partition the integers into n equivalence classes according to their re-

mainders modulo n. The equivalence class modulo n containing an integer a is

Œa n D fa C kn W k 2 Z g :

For example, Œ3 7 D f: : : ; 11; 4; 3; 10; 17; : : :g; we can also denote this set by

Œ4 7 and Œ10 7 . Using the notation deﬁned on page 54, we can say that writing

a 2 Œb n is the same as writing a b .mod n/. The set of all such equivalence

classes is

31.1 Elementary number-theoretic notions 929

Z n D fŒa n W 0 a n 1g : (31.1)

When you see the deﬁnition

Z n D f0; 1; : : : ; n 1g ; (31.2)

you should read it as equivalent to equation (31.1) with the understanding that 0

represents Œ0 n , 1 represents Œ1 n , and so on; each class is represented by its smallest

nonnegative element. You should keep the underlying equivalence classes in mind,

however. For example, if we refer to 1 as a member of Z n , we are really referring

to Œn 1 n , since 1 n 1 .mod n/.

Common divisors and greatest common divisors

If d is a divisor of a and d is also a divisor of b, then d is a common divisor of a

and b. For example, the divisors of 30 are 1, 2, 3, 5, 6, 10, 15, and 30, and so the

common divisors of 24 and 30 are 1, 2, 3, and 6. Note that 1 is a common divisor

of any two integers.

An important property of common divisors is that

d j a and d j b implies d j .a C b/ and d j .a b/ : (31.3)

More generally, we have that

d j a and d j b implies d j .ax C by/ (31.4)

for any integers x and y. Also, if a j b, then either jaj jbj or b D 0, which

implies that

a j b and b j a implies a D ˙b : (31.5)

The greatest common divisor of two integers a and b, not both zero, is the

largest of the common divisors of a and b; we denote it by gcd.a; b/. For example,

gcd.24; 30/ D 6, gcd.5; 7/ D 1, and gcd.0; 9/ D 9. If a and b are both nonzero,

then gcd.a; b/ is an integer between 1 and min.jaj ; jbj/. We deﬁne gcd.0; 0/ to

be 0; this deﬁnition is necessary to make standard properties of the gcd function

(such as equation (31.9) below) universally valid.

The following are elementary properties of the gcd function:

gcd.a; b/ D gcd.b; a/ ; (31.6)

gcd.a; b/ D gcd.a; b/ ; (31.7)

gcd.a; b/ D gcd.jaj ; jbj/ ; (31.8)

gcd.a; 0/ D jaj ; (31.9)

gcd.a; ka/ D jaj for any k 2 Z : (31.10)

The following theorem provides an alternative and useful characterization of

gcd.a; b/.

930 Chapter 31 Number-Theoretic Algorithms

Theorem 31.2

If a and b are any integers, not both zero, then gcd.a; b/ is the smallest positive

element of the set fax C by W x; y 2 Z g of linear combinations of a and b.

Proof Let s be the smallest positive such linear combination of a and b, and let

s D ax C by for some x; y 2 Z . Let q D ba=sc. Equation (3.8) then implies

a mod s D a qs

D a q.ax C by/

D a .1 qx/ C b .qy/ ;

and so a mod s is a linear combination of a and b as well. But, since 0

a mod s < s, we have that a mod s D 0, because s is the smallest positive such lin-

ear combination. Therefore, we have that s j a and, by analogous reasoning, s j b.

Thus, s is a common divisor of a and b, and so gcd.a; b/ s. Equation (31.4)

implies that gcd.a; b/ j s, since gcd.a; b/ divides both a and b and s is a linear

combination of a and b. But gcd.a; b/ j s and s > 0 imply that gcd.a; b/ s.

Combining gcd.a; b/ s and gcd.a; b/ s yields gcd.a; b/ D s. We conclude

that s is the greatest common divisor of a and b.

Corollary 31.3

For any integers a and b, if d j a and d j b, then d j gcd.a; b/.

Proof This corollary follows from equation (31.4), because gcd.a; b/ is a linear

combination of a and b by Theorem 31.2.

Corollary 31.4

For all integers a and b and any nonnegative integer n,

gcd.an; bn/ D n gcd.a; b/ :

Proof If n D 0, the corollary is trivial. If n > 0, then gcd.an; bn/ is the smallest

positive element of the set fanx C bny W x; y 2 Z g, which is n times the smallest

positive element of the set fax C by W x; y 2 Z g.

Corollary 31.5

For all positive integers n, a, and b, if n j ab and gcd.a; n/ D 1, then n j b.

Proof We leave the proof as Exercise 31.1-5.

31.1 Elementary number-theoretic notions 931

Relatively prime integers

Two integers a and b are relatively prime if their only common divisor is 1, that

is, if gcd.a; b/ D 1. For example, 8 and 15 are relatively prime, since the divisors

of 8 are 1, 2, 4, and 8, and the divisors of 15 are 1, 3, 5, and 15. The following

theorem states that if two integers are each relatively prime to an integer p, then

their product is relatively prime to p.

Theorem 31.6

For any integers a, b, and p, if both gcd.a; p/ D 1 and gcd.b; p/ D 1, then

gcd.ab; p/ D 1.

Proof It follows from Theorem 31.2 that there exist integers x, y, x 0

, and y 0

such

that

ax C py D 1 ;

bx

0

C py

0

D 1 :

Multiplying these equations and rearranging, we have

ab.xx

0

/ C p.ybx

0

C y

0

ax C pyy

0

/ D 1 :

Since 1 is thus a positive linear combination of ab and p, an appeal to Theo-

rem 31.2 completes the proof.

Integers n 1 , n 2 , . . . , n k are pairwise relatively prime if, whenever i ¤ j , we

have gcd.n i ; n j / D 1.

Unique factorization

An elementary but important fact about divisibility by primes is the following.

Theorem 31.7

For all primes p and all integers a and b, if p j ab, then p j a or p j b (or both).

Proof Assume for the purpose of contradiction that p j ab, but that p − a and

p − b. Thus, gcd.a; p/ D 1 and gcd.b; p/ D 1, since the only divisors of p are 1

and p, and we assume that p divides neither a nor b. Theorem 31.6 then implies

that gcd.ab; p/ D 1, contradicting our assumption that p j ab, since p j ab

implies gcd.ab; p/ D p. This contradiction completes the proof.

A consequence of Theorem 31.7 is that we can uniquely factor any composite

integer into a product of primes.

932 Chapter 31 Number-Theoretic Algorithms

Theorem 31.8 (Unique factorization)

There is exactly one way to write any composite integer a as a product of the form

a D p

e 1

1

p

e 2

2

p

e r

r

;

where the p i are prime, p 1 < p 2 < < p r , and the e i are positive integers.

Proof We leave the proof as Exercise 31.1-11.

As an example, the number 6000 is uniquely factored into primes as 2 4 3 5 3

.

Exercises

31.1-1

Prove that if a > b > 0 and c D a C b, then c mod a D b.

31.1-2

Prove that there are inﬁnitely many primes. (Hint: Show that none of the primes

p 1 ; p 2 ; : : : ; p k divide .p 1 p 2 p k / C 1.)

31.1-3

Prove that if a j b and b j c, then a j c.

31.1-4

Prove that if p is prime and 0 < k < p, then gcd.k; p/ D 1.

31.1-5

Prove Corollary 31.5.

31.1-6

Prove that if p is prime and 0 < k < p, then p j

p

k

. Conclude that for all integers

a and b and all primes p,

.a C b/

p

a

p

C b

p

.mod p/ :

31.1-7

Prove that if a and b are any positive integers such that a j b, then

.x mod b/ mod a D x mod a

for any x. Prove, under the same assumptions, that

x y .mod b/ implies x y .mod a/

for any integers x and y.

31.2 Greatest common divisor 933

31.1-8

For any integer k > 0, an integer n is a kth power if there exists an integer a such

that a k D n. Furthermore, n > 1 is a nontrivial power if it is a kth power for

some integer k > 1. Show how to determine whether a given ˇ-bit integer n is a

nontrivial power in time polynomial in ˇ.

31.1-9

Prove equations (31.6)–(31.10).

31.1-10

Show that the gcd operator is associative. That is, prove that for all integers a, b,

and c,

gcd.a; gcd.b; c// D gcd.gcd.a; b/; c/ :

31.1-11 ?

Prove Theorem 31.8.

31.1-12

Give efﬁcient algorithms for the operations of dividing a ˇ-bit integer by a shorter

integer and of taking the remainder of a ˇ-bit integer when divided by a shorter

integer. Your algorithms should run in time ‚.ˇ 2 /.

31.1-13

Give an efﬁcient algorithm to convert a given ˇ-bit (binary) integer to a decimal

representation. Argue that if multiplication or division of integers whose length

is at most ˇ takes time M.ˇ/, then we can convert binary to decimal in time

‚.M.ˇ/ lg ˇ/. (Hint: Use a divide-and-conquer approach, obtaining the top and

bottom halves of the result with separate recursions.)

31.2 Greatest common divisor

In this section, we describe Euclid’s algorithm for efﬁciently computing the great-

est common divisor of two integers. When we analyze the running time, we shall

see a surprising connection with the Fibonacci numbers, which yield a worst-case

input for Euclid’s algorithm.

We restrict ourselves in this section to nonnegative integers. This restriction is

justiﬁed by equation (31.8), which states that gcd.a; b/ D gcd.jaj ; jbj/.

934 Chapter 31 Number-Theoretic Algorithms

In principle, we can compute gcd.a; b/ for positive integers a and b from the

prime factorizations of a and b. Indeed, if

a D p

e 1

1

p

e 2

2

p

e r

r

; (31.11)

b D p

f 1

1 p

f 2

2 p

f r

r

; (31.12)

with zero exponents being used to make the set of primes p 1 ; p 2 ; : : : ; p r the same

for both a and b, then, as Exercise 31.2-1 asks you to show,

gcd.a; b/ D p

min.e 1 ;f 1 /

1

p

min.e 2 ;f 2 /

2

p min.e r ;f r /

r

: (31.13)

As we shall show in Section 31.9, however, the best algorithms to date for factoring

do not run in polynomial time. Thus, this approach to computing greatest common

divisors seems unlikely to yield an efﬁcient algorithm.

Euclid’s algorithm for computing greatest common divisors relies on the follow-

ing theorem.

Theorem 31.9 (GCD recursion theorem)

For any nonnegative integer a and any positive integer b,

gcd.a; b/ D gcd.b; a mod b/ :

Proof We shall show that gcd.a; b/ and gcd.b; a mod b/ divide each other, so

that by equation (31.5) they must be equal (since they are both nonnegative).

We ﬁrst show that gcd.a; b/ j gcd.b; a mod b/. If we let d D gcd.a; b/, then

d j a and d j b. By equation (3.8), a mod b D a qb, where q D ba=bc.

Since a mod b is thus a linear combination of a and b, equation (31.4) implies that

d j .a mod b/. Therefore, since d j b and d j .a mod b/, Corollary 31.3 implies

that d j gcd.b; a mod b/ or, equivalently, that

gcd.a; b/ j gcd.b; a mod b/: (31.14)

Showing that gcd.b; a mod b/ j gcd.a; b/ is almost the same. If we now let

d D gcd.b; a mod b/, then d j b and d j .a mod b/. Since a D qb C .a mod b/,

where q D ba=bc, we have that a is a linear combination of b and .a mod b/. By

equation (31.4), we conclude that d j a. Since d j b and d j a, we have that

d j gcd.a; b/ by Corollary 31.3 or, equivalently, that

gcd.b; a mod b/ j gcd.a; b/: (31.15)

Using equation (31.5) to combine equations (31.14) and (31.15) completes the

proof.

31.2 Greatest common divisor 935

Euclid’s algorithm

The Elements of Euclid (circa 300 B.C.) describes the following gcd algorithm,

although it may be of even earlier origin. We express Euclid’s algorithm as a

recursive program based directly on Theorem 31.9. The inputs a and b are arbitrary

nonnegative integers.

EUCLID.a; b/

1 if b == 0

2 return a

3 else return EUCLID.b; a mod b/

As an example of the running of EUCLID, consider the computation of gcd.30; 21/:

EUCLID.30; 21/ D EUCLID.21; 9/

D EUCLID.9; 3/

D EUCLID.3; 0/

D 3 :

This computation calls EUCLID recursively three times.

The correctness of EUCLID follows from Theorem 31.9 and the property that if

the algorithm returns a in line 2, then b D 0, so that equation (31.9) implies that

gcd.a; b/ D gcd.a; 0/ D a. The algorithm cannot recurse indeﬁnitely, since the

second argument strictly decreases in each recursive call and is always nonnegative.

Therefore, EUCLID always terminates with the correct answer.

The running time of Euclid’s algorithm

We analyze the worst-case running time of EUCLID as a function of the size of

a and b. We assume with no loss of generality that a > b 0. To justify this

assumption, observe that if b > a 0, then EUCLID.a; b/ immediately makes the

recursive call EUCLID.b; a/. That is, if the ﬁrst argument is less than the second

argument, EUCLID spends one recursive call swapping its arguments and then pro-

ceeds. Similarly, if b D a > 0, the procedure terminates after one recursive call,

since a mod b D 0.

The overall running time of EUCLID is proportional to the number of recursive

calls it makes. Our analysis makes use of the Fibonacci numbers F k , deﬁned by

the recurrence (3.22).

Lemma 31.10

If a > b 1 and the call EUCLID.a; b/ performs k 1 recursive calls, then

a F kC2 and b F kC1 .

936 Chapter 31 Number-Theoretic Algorithms

Proof The proof proceeds by induction on k. For the basis of the induction, let

k D 1. Then, b 1 D F 2 , and since a > b, we must have a 2 D F 3 . Since

b > .a mod b/, in each recursive call the ﬁrst argument is strictly larger than the

second; the assumption that a > b therefore holds for each recursive call.

Assume inductively that the lemma holds if k 1 recursive calls are made; we

shall then prove that the lemma holds for k recursive calls. Since k > 0, we have

b > 0, and EUCLID.a; b/ calls EUCLID.b; a mod b/ recursively, which in turn

makes k 1 recursive calls. The inductive hypothesis then implies that b F kC1

(thus proving part of the lemma), and a mod b F k . We have

b C .a mod b/ D b C .a b ba=bc/

a ;

since a > b > 0 implies ba=bc 1. Thus,

a b C .a mod b/

F kC1 C F k

D F kC2 :

The following theorem is an immediate corollary of this lemma.

Theorem 31.11 (Lam´e’s theorem)

For any integer k 1, if a > b 1 and b < F kC1 , then the call EUCLID.a; b/

makes fewer than k recursive calls.

We can show that the upper bound of Theorem 31.11 is the best possible by

showing that the call EUCLID.F kC1 ; F k / makes exactly k 1 recursive calls

when k 2. We use induction on k. For the base case, k D 2, and the call

EUCLID.F 3 ; F 2 / makes exactly one recursive call, to EUCLID.1; 0/. (We have to

start at k D 2, because when k D 1 we do not have F 2 > F 1 .) For the induc-

tive step, assume that EUCLID.F k ; F k1 / makes exactly k 2 recursive calls. For

k > 2, we have F k > F k1 > 0 and F kC1 D F k CF k1 , and so by Exercise 31.1-1,

we have F kC1 mod F k D F k1 . Thus, we have

gcd.F kC1 ; F k / D gcd.F k ; F kC1 mod F k /

D gcd.F k ; F k1 / :

Therefore, the call EUCLID.F kC1 ; F k / recurses one time more than the call

EUCLID.F k ; F k1 /, or exactly k 1 times, meeting the upper bound of Theo-

rem 31.11.

Since F k is approximately k =

p

5, where is the golden ratio .1 C

p

5/=2 de-

ﬁned by equation (3.24), the number of recursive calls in EUCLID is O.lg b/. (See

31.2 Greatest common divisor 937

a b ba=bc d x y

99 78 1 3 11 14

78 21 3 3 3 11

21 15 1 3 2 3

15 6 2 3 1 2

6 3 2 3 0 1

3 0 — 3 1 0

Figure 31.1 How EXTENDED-EUCLID computes gcd.99; 78/. Each line shows one level of the

recursion: the values of the inputs a and b, the computed value ba=bc, and the values d, x, and y

returned. The triple .d; x; y/ returned becomes the triple .d

0

; x

0

; y

0

/ used at the next higher level

of recursion. The call EXTENDED-EUCLID.99; 78/ returns .3; 11; 14/, so that gcd.99; 78/ D 3 D

99 .11/ C 78 14.

Exercise 31.2-5 for a tighter bound.) Therefore, if we call EUCLID on two ˇ-bit

numbers, then it performs O.ˇ/ arithmetic operations and O.ˇ 3 / bit operations

(assuming that multiplication and division of ˇ-bit numbers take O.ˇ 2 / bit oper-

ations). Problem 31-2 asks you to show an O.ˇ 2 / bound on the number of bit

operations.

The extended form of Euclid’s algorithm

We now rewrite Euclid’s algorithm to compute additional useful information.

Speciﬁcally, we extend the algorithm to compute the integer coefﬁcients x and y

such that

d D gcd.a; b/ D ax C by : (31.16)

Note that x and y may be zero or negative. We shall ﬁnd these coefﬁcients useful

later for computing modular multiplicative inverses. The procedure EXTENDED-

EUCLID takes as input a pair of nonnegative integers and returns a triple of the

form .d; x; y/ that satisﬁes equation (31.16).

EXTENDED-EUCLID.a; b/

1 if b == 0

2 return .a; 1; 0/

3 else .d 0 ; x 0 ; y 0 / D EXTENDED-EUCLID.b; a mod b/

4 .d; x; y/ D .d 0 ; y 0 ; x 0 ba=bc y 0 /

5 return .d; x; y/

Figure 31.1 illustrates how EXTENDED-EUCLID computes gcd.99; 78/.

The EXTENDED-EUCLID procedure is a variation of the EUCLID procedure.

Line 1 is equivalent to the test “b == 0” in line 1 of EUCLID. If b D 0, then

938 Chapter 31 Number-Theoretic Algorithms

EXTENDED-EUCLID returns not only d D a in line 2, but also the coefﬁcients

x D 1 and y D 0, so that a D ax C by. If b ¤ 0, EXTENDED-EUCLID ﬁrst

computes .d 0 ; x 0 ; y 0 / such that d 0 D gcd.b; a mod b/ and

d

0

D bx

0

C .a mod b/y

0

: (31.17)

As for EUCLID, we have in this case d D gcd.a; b/ D d 0 D gcd.b; a mod b/.

To obtain x and y such that d D ax C by, we start by rewriting equation (31.17)

using the equation d D d 0

and equation (3.8):

d D bx

0

C .a b ba=bc/y

0

D ay

0

C b.x

0

ba=bc y

0

/ :

Thus, choosing x D y 0

and y D x 0 ba=bc y 0

satisﬁes the equation d D ax C by,

proving the correctness of EXTENDED-EUCLID.

Since the number of recursive calls made in EUCLID is equal to the number

of recursive calls made in EXTENDED-EUCLID, the running times of EUCLID

and EXTENDED-EUCLID are the same, to within a constant factor. That is, for

a > b > 0, the number of recursive calls is O.lg b/.

Exercises

31.2-1

Prove that equations (31.11) and (31.12) imply equation (31.13).

31.2-2

Compute the values .d; x; y/ that the call EXTENDED-EUCLID.899; 493/ returns.

31.2-3

Prove that for all integers a, k, and n,

gcd.a; n/ D gcd.a C kn; n/ :

31.2-4

Rewrite EUCLID in an iterative form that uses only a constant amount of memory

(that is, stores only a constant number of integer values).

31.2-5

If a > b 0, show that the call EUCLID.a; b/ makes at most 1 C log

b recursive

calls. Improve this bound to 1 C log

.b= gcd.a; b//.

31.2-6

What does EXTENDED-EUCLID.F kC1 ; F k / return? Prove your answer correct.

31.3 Modular arithmetic 939

31.2-7

Deﬁne the gcd function for more than two arguments by the recursive equation

gcd.a 0 ; a 1 ; : : : ; a n / D gcd.a 0 ; gcd.a 1 ; a 2 ; : : : ; a n //. Show that the gcd function

returns the same answer independent of the order in which its arguments are speci-

ﬁed. Also show how to ﬁnd integers x 0 ; x 1 ; : : : ; x n such that gcd.a 0 ; a 1 ; : : : ; a n / D

a 0 x 0 C a 1 x 1 C C a n x n . Show that the number of divisions performed by your

algorithm is O.n C lg.max fa 0 ; a 1 ; : : : ; a n g//.

31.2-8

Deﬁne lcm.a 1 ; a 2 ; : : : ; a n / to be the least common multiple of the n integers

a 1 ; a 2 ; : : : ; a n , that is, the smallest nonnegative integer that is a multiple of each a i .

Show how to compute lcm.a 1 ; a 2 ; : : : ; a n / efﬁciently using the (two-argument) gcd

operation as a subroutine.

31.2-9

Prove that n 1 , n 2 , n 3 , and n 4 are pairwise relatively prime if and only if

gcd.n 1 n 2 ; n 3 n 4 / D gcd.n 1 n 3 ; n 2 n 4 / D 1 :

More generally, show that n 1 ; n 2 ; : : : ; n k are pairwise relatively prime if and only

if a set of dlg ke pairs of numbers derived from the n i are relatively prime.

31.3 Modular arithmetic

Informally, we can think of modular arithmetic as arithmetic as usual over the

integers, except that if we are working modulo n, then every result x is replaced

by the element of f0; 1; : : : ; n 1g that is equivalent to x, modulo n (that is, x is

replaced by x mod n). This informal model sufﬁces if we stick to the operations

of addition, subtraction, and multiplication. A more formal model for modular

arithmetic, which we now give, is best described within the framework of group

theory.

Finite groups

A group .S; ˚/ is a set S together with a binary operation ˚ deﬁned on S for

which the following properties hold:

1. Closure: For all a, b 2 S, we have a ˚ b 2 S.

2. Identity: There exists an element e 2 S, called the identity of the group, such

that e ˚ a D a ˚ e D a for all a 2 S.

3. Associativity: For all a, b, c 2 S, we have .a ˚ b/ ˚ c D a ˚ .b ˚ c/.

940 Chapter 31 Number-Theoretic Algorithms

4. Inverses: For each a 2 S, there exists a unique element b 2 S, called the

inverse of a, such that a ˚ b D b ˚ a D e.

As an example, consider the familiar group . Z ; C/ of the integers Z under the

operation of addition: 0 is the identity, and the inverse of a is a. If a group .S; ˚/

satisﬁes the commutative law a ˚ b D b ˚ a for all a; b 2 S, then it is an abelian

group. If a group .S; ˚/ satisﬁes jSj < 1, then it is a ﬁnite group.

The groups deﬁned by modular addition and multiplication

We can form two ﬁnite abelian groups by using addition and multiplication mod-

ulo n, where n is a positive integer. These groups are based on the equivalence

classes of the integers modulo n, deﬁned in Section 31.1.

To deﬁne a group on Z n , we need to have suitable binary operations, which

we obtain by redeﬁning the ordinary operations of addition and multiplication.

We can easily deﬁne addition and multiplication operations for Z n , because the

equivalence class of two integers uniquely determines the equivalence class of their

sum or product. That is, if a a 0 .mod n/ and b b 0 .mod n/, then

a C b a

0

C b

0

.mod n/ ;

ab a

0

b

0

.mod n/ :

Thus, we deﬁne addition and multiplication modulo n, denoted C n and n , by

Œa n C n Œb n D Œa C b n ; (31.18)

Œa n n Œb n D Œab n :

(We can deﬁne subtraction similarly on Z n by Œa n n Œb n D Œa b n , but divi-

sion is more complicated, as we shall see.) These facts justify the common and

convenient practice of using the smallest nonnegative element of each equivalence

class as its representative when performing computations in Z n . We add, subtract,

and multiply as usual on the representatives, but we replace each result x by the

representative of its class, that is, by x mod n.

Using this deﬁnition of addition modulo n, we deﬁne the additive group

modulo n as . Z n ; C n /. The size of the additive group modulo n is j Z n j D n.

Figure 31.2(a) gives the operation table for the group . Z 6 ; C 6 /.

Theorem 31.12

The system . Z n ; C n / is a ﬁnite abelian group.

Proof Equation (31.18) shows that . Z n ; C n / is closed. Associativity and com-

mutativity of C n follow from the associativity and commutativity of C:

31.3 Modular arithmetic 941

0 1 2 3 4 5

0

1

2

3

4

5

0 1 2 3 4 5

0 1 2 3 4 5

0 1 2 3 4 5

0 1 2 3 4 5

0 1 2 3 4 5

0 1 2 3 4 5

(a)

1 2 4 7 8 11 13 14

1

2

4

7

8

11

13

14

1 2 4 7 8 11 13 14

2 4 8 14 1 7 11 13

4 8 1 13 2 14 7 11

7 14 13 4 11 2 1 8

8 1 2 11 4 13 14 7

11 7 14 2 13 1 8 4

13 11 7 1 14 8 4 2

14 13 11 8 7 4 2 1

(b)

+

6

·

15

Figure 31.2 Two ﬁnite groups. Equivalence classes are denoted by their representative elements.

(a) The group . Z 6; C6/. (b) The group . Z

15

; 15/.

.Œa n C n Œb n / C n Œc n D Œa C b n C n Œc n

D Œ.a C b/ C c n

D Œa C .b C c/ n

D Œa n C n Œb C c n

D Œa n C n .Œb n C n Œc n / ;

Œa n C n Œb n D Œa C b n

D Œb C a n

D Œb n C n Œa n :

The identity element of . Z n ; C n / is 0 (that is, Œ0 n ). The (additive) inverse of

an element a (that is, of Œa n ) is the element a (that is, Œa n or Œn a n ), since

Œa n C n Œa n D Œa a n D Œ0 n .

Using the deﬁnition of multiplication modulo n, we deﬁne the multiplicative

group modulo n as . Z

n

; n /. The elements of this group are the set Z

n

of elements

in Z n that are relatively prime to n, so that each one has a unique inverse, modulo n:

Z

n

D fŒa n 2 Z n W gcd.a; n/ D 1g :

To see that Z

n

is well deﬁned, note that for 0 a < n, we have a .a C kn/

.mod n/ for all integers k. By Exercise 31.2-3, therefore, gcd.a; n/ D 1 implies

gcd.a C kn; n/ D 1 for all integers k. Since Œa n D fa C kn W k 2 Z g, the set Z

n

is well deﬁned. An example of such a group is

Z

15

D f1; 2; 4; 7; 8; 11; 13; 14g ;

942 Chapter 31 Number-Theoretic Algorithms

where the group operation is multiplication modulo 15. (Here we denote an el-

ement Œa 15 as a; for example, we denote Œ7 15 as 7.) Figure 31.2(b) shows the

group . Z

15

; 15 /. For example, 8 11 13 .mod 15/, working in Z

15

. The iden-

tity for this group is 1.

Theorem 31.13

The system . Z

n

; n / is a ﬁnite abelian group.

Proof Theorem 31.6 implies that . Z

n

; n / is closed. Associativity and commu-

tativity can be proved for n as they were for C n in the proof of Theorem 31.12.

The identity element is Œ1 n . To show the existence of inverses, let a be an element

of Z

n

and let .d; x; y/ be returned by EXTENDED-EUCLID.a; n/. Then, d D 1,

since a 2 Z

n

, and

ax C ny D 1 (31.19)

or, equivalently,

ax 1 .mod n/ :

Thus, Œx n is a multiplicative inverse of Œa n , modulo n. Furthermore, we claim

that Œx n 2 Z

n

. To see why, equation (31.19) demonstrates that the smallest pos-

itive linear combination of x and n must be 1. Therefore, Theorem 31.2 implies

that gcd.x; n/ D 1. We defer the proof that inverses are uniquely deﬁned until

Corollary 31.26.

As an example of computing multiplicative inverses, suppose that a D 5 and

n D 11. Then EXTENDED-EUCLID.a; n/ returns .d; x; y/ D .1; 2; 1/, so that

1 D 5 .2/ C 11 1. Thus, Œ2 11 (i.e., Œ9 11 ) is the multiplicative inverse of Œ5 11 .

When working with the groups . Z n ; C n / and . Z

n

; n / in the remainder of this

chapter, we follow the convenient practice of denoting equivalence classes by their

representative elements and denoting the operations C n and n by the usual arith-

metic notations C and (or juxtaposition, so that ab D a b) respectively. Also,

equivalences modulo n may also be interpreted as equations in Z n . For example,

the following two statements are equivalent:

ax b .mod n/ ;

Œa n n Œx n D Œb n :

As a further convenience, we sometimes refer to a group .S; ˚/ merely as S

when the operation ˚ is understood from context. We may thus refer to the groups

. Z n ; C n / and . Z

n

; n / as Z n and Z

n

, respectively.

We denote the (multiplicative) inverse of an element a by .a 1

mod n/. Division

in Z

n

is deﬁned by the equation a=b ab 1 .mod n/. For example, in Z

15

31.3 Modular arithmetic 943

we have that 7 1 13 .mod 15/, since 7 13 D 91 1 .mod 15/, so that

4=7 4 13 7 .mod 15/.

The size of Z

n

is denoted .n/. This function, known as Euler’s phi function,

satisﬁes the equation

.n/ D n

Y

p W p is prime and p j n

1

1

p

; (31.20)

so that p runs over all the primes dividing n (including n itself, if n is prime).

We shall not prove this formula here. Intuitively, we begin with a list of the n

remainders f0; 1; : : : ; n 1g and then, for each prime p that divides n, cross out

every multiple of p in the list. For example, since the prime divisors of 45 are 3

and 5,

.45/ D 45

1

1

3

1

1

5

D 45

2

3

4

5

D 24 :

If p is prime, then Z

p

D f1; 2; : : : ; p 1g, and

.p/ D p

1

1

p

D p 1 : (31.21)

If n is composite, then .n/ < n 1, although it can be shown that

.n/ >

n

e ln ln n C

3

ln ln n

(31.22)

for n 3, where D 0:5772156649 : : : is Euler’s constant. A somewhat simpler

(but looser) lower bound for n > 5 is

.n/ >

n

6 ln ln n

: (31.23)

The lower bound (31.22) is essentially the best possible, since

lim inf

n!1

.n/

n= ln ln n

D e

: (31.24)

Subgroups

If .S; ˚/ is a group, S 0 S, and .S 0 ; ˚/ is also a group, then .S 0 ; ˚/ is a subgroup

of .S; ˚/. For example, the even integers form a subgroup of the integers under the

operation of addition. The following theorem provides a useful tool for recognizing

subgroups.

944 Chapter 31 Number-Theoretic Algorithms

Theorem 31.14 (A nonempty closed subset of a ﬁnite group is a subgroup)

If .S; ˚/ is a ﬁnite group and S 0

is any nonempty subset of S such that a ˚ b 2 S 0

for all a; b 2 S 0

, then .S 0 ; ˚/ is a subgroup of .S; ˚/.

Proof We leave the proof as Exercise 31.3-3.

For example, the set f0; 2; 4; 6g forms a subgroup of Z 8 , since it is nonempty

and closed under the operation C (that is, it is closed under C 8 ).

The following theorem provides an extremely useful constraint on the size of a

subgroup; we omit the proof.

Theorem 31.15 (Lagrange’s theorem)

If .S; ˚/ is a ﬁnite group and .S 0 ; ˚/ is a subgroup of .S; ˚/, then jS 0 j is a divisor

of jSj.

A subgroup S 0

of a group S is a proper subgroup if S 0 ¤ S. We shall use the

following corollary in our analysis in Section 31.8 of the Miller-Rabin primality

test procedure.

Corollary 31.16

If S 0

is a proper subgroup of a ﬁnite group S, then jS 0 j jSj =2.

Subgroups generated by an element

Theorem 31.14 gives us an easy way to produce a subgroup of a ﬁnite group .S; ˚/:

choose an element a and take all elements that can be generated from a using the

group operation. Speciﬁcally, deﬁne a .k/

for k 1 by

a

.k/

D

k M

iD1

a D a ˚ a ˚ ˚ a œ

k

:

For example, if we take a D 2 in the group Z 6 , the sequence a .1/ ; a .2/ ; a .3/ ; : : : is

2; 4; 0; 2; 4; 0; 2; 4; 0; : : : :

In the group Z n , we have a .k/ D ka mod n, and in the group Z

n

, we have a .k/ D

a k

mod n. We deﬁne the subgroup generated by a, denoted hai or .hai; ˚/, by

hai D fa

.k/

W k 1g :

We say that a generates the subgroup hai or that a is a generator of hai. Since S is

ﬁnite, hai is a ﬁnite subset of S, possibly including all of S. Since the associativity

of ˚ implies

31.3 Modular arithmetic 945

a

.i/

˚ a

.j /

D a

.iCj /

;

hai is closed and therefore, by Theorem 31.14, hai is a subgroup of S. For example,

in Z 6 , we have

h0i D f0g ;

h1i D f0; 1; 2; 3; 4; 5g ;

h2i D f0; 2; 4g :

Similarly, in Z

7

, we have

h1i D f1g ;

h2i D f1; 2; 4g ;

h3i D f1; 2; 3; 4; 5; 6g :

The order of a (in the group S), denoted ord.a/, is deﬁned as the smallest posi-

tive integer t such that a .t/ D e.

Theorem 31.17

For any ﬁnite group .S; ˚/ and any a 2 S, the order of a is equal to the size of the

subgroup it generates, or ord.a/ D jhaij.

Proof Let t D ord.a/. Since a .t/ D e and a .tCk/ D a .t/ ˚ a .k/ D a .k/

for

k 1, if i > t, then a .i/ D a .j /

for some j < i. Thus, as we generate ele-

ments by a, we see no new elements after a .t/

. Thus, hai D fa .1/ ; a .2/ ; : : : ; a .t/ g,

and so jhaij t. To show that jhaij t, we show that each element of the se-

quence a .1/ ; a .2/ ; : : : ; a .t/

is distinct. Suppose for the purpose of contradiction that

a .i/ D a .j /

for some i and j satisfying 1 i < j t. Then, a .iCk/ D a .j Ck/

for k 0. But this equality implies that a .iC.tj // D a .j C.tj // D e, a contradic-

tion, since i C.t j / < t but t is the least positive value such that a .t/ D e. There-

fore, each element of the sequence a .1/ ; a .2/ ; : : : ; a .t/

is distinct, and jhaij t. We

conclude that ord.a/ D jhaij.

Corollary 31.18

The sequence a .1/ ; a .2/ ; : : : is periodic with period t D ord.a/; that is, a .i/ D a .j /

if and only if i j .mod t/.

Consistent with the above corollary, we deﬁne a .0/

as e and a .i/

as a .i mod t/

,

where t D ord.a/, for all integers i.

Corollary 31.19

If .S; ˚/ is a ﬁnite group with identity e, then for all a 2 S,

a

.jSj/

D e :

946 Chapter 31 Number-Theoretic Algorithms

Proof Lagrange’s theorem (Theorem 31.15) implies that ord.a/ j jSj, and so

jSj 0 .mod t/, where t D ord.a/. Therefore, a .jSj/ D a .0/ D e.

Exercises

31.3-1

Draw the group operation tables for the groups . Z 4 ; C 4 / and . Z

5

; 5 /. Show that

these groups are isomorphic by exhibiting a one-to-one correspondence ˛ between

their elements such that a C b c .mod 4/ if and only if ˛.a/ ˛.b/ ˛.c/

.mod 5/.

31.3-2

List all subgroups of Z 9 and of Z

13

.

31.3-3

Prove Theorem 31.14.

31.3-4

Show that if p is prime and e is a positive integer, then

.p

e

/ D p

e1

.p 1/ :

31.3-5

Show that for any integer n > 1 and for any a 2 Z

n

, the function f a W Z

n

! Z

n

deﬁned by f a .x/ D ax mod n is a permutation of Z

n

.

31.4 Solving modular linear equations

We now consider the problem of ﬁnding solutions to the equation

ax b .mod n/ ; (31.25)

where a > 0 and n > 0. This problem has several applications; for example,

we shall use it as part of the procedure for ﬁnding keys in the RSA public-key

cryptosystem in Section 31.7. We assume that a, b, and n are given, and we wish

to ﬁnd all values of x, modulo n, that satisfy equation (31.25). The equation may

have zero, one, or more than one such solution.

Let hai denote the subgroup of Z n generated by a. Since hai D fa .x/ W x > 0g D

fax mod n W x > 0g, equation (31.25) has a solution if and only if Œb 2 hai. La-

grange’s theorem (Theorem 31.15) tells us that jhaij must be a divisor of n. The

following theorem gives us a precise characterization of hai.

31.4 Solving modular linear equations 947

Theorem 31.20

For any positive integers a and n, if d D gcd.a; n/, then

hai D hdi D f0; d; 2d; : : : ; ..n=d/ 1/dg (31.26)

in Z n , and thus

jhaij D n=d :

Proof We begin by showing that d 2 hai. Recall that EXTENDED-EUCLID.a; n/

produces integers x 0

and y 0

such that ax 0 C ny 0 D d. Thus, ax 0 d .mod n/, so

that d 2 hai. In other words, d is a multiple of a in Z n .

Since d 2 hai, it follows that every multiple of d belongs to hai, because any

multiple of a multiple of a is itself a multiple of a. Thus, hai contains every element

in f0; d; 2d; : : : ; ..n=d/ 1/dg. That is, hdi hai.

We now show that hai hdi. If m 2 hai, then m D ax mod n for some

integer x, and so m D ax C ny for some integer y. However, d j a and d j n, and

so d j m by equation (31.4). Therefore, m 2 hdi.

Combining these results, we have that hai D hdi. To see that jhaij D n=d,

observe that there are exactly n=d multiples of d between 0 and n1, inclusive.

Corollary 31.21

The equation ax b .mod n/ is solvable for the unknown x if and only if d j b,

where d D gcd.a; n/.

Proof The equation ax b .mod n/ is solvable if and only if Œb 2 hai, which

is the same as saying

.b mod n/ 2 f0; d; 2d; : : : ; ..n=d/ 1/dg ;

by Theorem 31.20. If 0 b < n, then b 2 hai if and only if d j b, since the

members of hai are precisely the multiples of d. If b < 0 or b n, the corollary

then follows from the observation that d j b if and only if d j .b mod n/, since b

and b mod n differ by a multiple of n, which is itself a multiple of d.

Corollary 31.22

The equation ax b .mod n/ either has d distinct solutions modulo n, where

d D gcd.a; n/, or it has no solutions.

Proof If ax b .mod n/ has a solution, then b 2 hai. By Theorem 31.17,

ord.a/ D jhaij, and so Corollary 31.18 and Theorem 31.20 imply that the sequence

ai mod n, for i D 0; 1; : : :, is periodic with period jhaij D n=d. If b 2 hai, then b

appears exactly d times in the sequence ai mod n, for i D 0; 1; : : : ; n 1, since

948 Chapter 31 Number-Theoretic Algorithms

the length-.n=d/ block of values hai repeats exactly d times as i increases from 0

to n1. The indices x of the d positions for which ax mod n D b are the solutions

of the equation ax b .mod n/.

Theorem 31.23

Let d D gcd.a; n/, and suppose that d D ax 0 C ny 0

for some integers x 0

and y 0

(for example, as computed by EXTENDED-EUCLID). If d j b, then the equation

ax b .mod n/ has as one of its solutions the value x 0 , where

x 0 D x

0

.b=d/ mod n :

Proof We have

ax 0 ax

0

.b=d/ .mod n/

d.b=d/ .mod n/ (because ax 0 d .mod n/)

b .mod n/ ;

and thus x 0 is a solution to ax b .mod n/.

Theorem 31.24

Suppose that the equation ax b .mod n/ is solvable (that is, d j b, where

d D gcd.a; n/) and that x 0 is any solution to this equation. Then, this equa-

tion has exactly d distinct solutions, modulo n, given by x i D x 0 C i.n=d/ for

i D 0; 1; : : : ; d 1.

Proof Because n=d > 0 and 0 i.n=d/ < n for i D 0; 1; : : : ; d 1, the

values x 0 ; x 1 ; : : : ; x d1 are all distinct, modulo n. Since x 0 is a solution of ax b

.mod n/, we have ax 0 mod n b .mod n/. Thus, for i D 0; 1; : : : ; d 1, we

have

ax i mod n D a.x 0 C in=d/ mod n

D .ax 0 C ain=d/ mod n

D ax 0 mod n (because d j a implies that ain=d is a multiple of n)

b .mod n/ ;

and hence ax i b .mod n/, making x i a solution, too. By Corollary 31.22, the

equation ax b .mod n/ has exactly d solutions, so that x 0 ; x 1 ; : : : ; x d1 must

be all of them.

We have now developed the mathematics needed to solve the equation ax b

.mod n/; the following algorithm prints all solutions to this equation. The inputs

a and n are arbitrary positive integers, and b is an arbitrary integer.

31.4 Solving modular linear equations 949

MODULAR-LINEAR-EQUATION-SOLVER.a; b; n/

1 .d; x 0 ; y 0 / D EXTENDED-EUCLID.a; n/

2 if d j b

3 x 0 D x 0 .b=d/ mod n

4 for i D 0 to d 1

5 print .x 0 C i.n=d// mod n

6 else print “no solutions”

As an example of the operation of this procedure, consider the equation 14x

30 .mod 100/ (here, a D 14, b D 30, and n D 100). Calling EXTENDED-

EUCLID in line 1, we obtain .d; x 0 ; y 0 / D .2; 7; 1/. Since 2 j 30, lines 3–5

execute. Line 3 computes x 0 D .7/.15/ mod 100 D 95. The loop on lines 4–5

prints the two solutions 95 and 45.

The procedure MODULAR-LINEAR-EQUATION-SOLVER works as follows.

Line 1 computes d D gcd.a; n/, along with two values x 0

and y 0

such that d D

ax 0 C ny 0

, demonstrating that x 0

is a solution to the equation ax 0 d .mod n/.

If d does not divide b, then the equation ax b .mod n/ has no solution, by

Corollary 31.21. Line 2 checks to see whether d j b; if not, line 6 reports that there

are no solutions. Otherwise, line 3 computes a solution x 0 to ax b .mod n/,

in accordance with Theorem 31.23. Given one solution, Theorem 31.24 states that

adding multiples of .n=d/, modulo n, yields the other d 1 solutions. The for

loop of lines 4–5 prints out all d solutions, beginning with x 0 and spaced n=d

apart, modulo n.

MODULAR-LINEAR-EQUATION-SOLVER performs O.lg n C gcd.a; n// arith-

metic operations, since EXTENDED-EUCLID performs O.lg n/ arithmetic opera-

tions, and each iteration of the for loop of lines 4–5 performs a constant number of

arithmetic operations.

The following corollaries of Theorem 31.24 give specializations of particular

interest.

Corollary 31.25

For any n > 1, if gcd.a; n/ D 1, then the equation ax b .mod n/ has a unique

solution, modulo n.

If b D 1, a common case of considerable interest, the x we are looking for is a

multiplicative inverse of a, modulo n.

Corollary 31.26

For any n > 1, if gcd.a; n/ D 1, then the equation ax 1 .mod n/ has a unique

solution, modulo n. Otherwise, it has no solution.

950 Chapter 31 Number-Theoretic Algorithms

Thanks to Corollary 31.26, we can use the notation a 1

mod n to refer to the

multiplicative inverse of a, modulo n, when a and n are relatively prime. If

gcd.a; n/ D 1, then the unique solution to the equation ax 1 .mod n/ is the

integer x returned by EXTENDED-EUCLID, since the equation

gcd.a; n/ D 1 D ax C ny

implies ax 1 .mod n/. Thus, we can compute a 1

mod n efﬁciently using

EXTENDED-EUCLID.

Exercises

31.4-1

Find all solutions to the equation 35x 10 .mod 50/.

31.4-2

Prove that the equation ax ay .mod n/ implies x y .mod n/ whenever

gcd.a; n/ D 1. Show that the condition gcd.a; n/ D 1 is necessary by supplying a

counterexample with gcd.a; n/ > 1.

31.4-3

Consider the following change to line 3 of the procedure MODULAR-LINEAR-

EQUATION-SOLVER:

3 x 0 D x 0 .b=d/ mod .n=d/

Will this work? Explain why or why not.

31.4-4 ?

Let p be prime and f .x/ f 0 C f 1 x C C f t x t .mod p/ be a polyno-

mial of degree t, with coefﬁcients f i drawn from Z p . We say that a 2 Z p

is a zero of f if f .a/ 0 .mod p/. Prove that if a is a zero of f , then

f .x/ .x a/g.x/ .mod p/ for some polynomial g.x/ of degree t 1. Prove

by induction on t that if p is prime, then a polynomial f .x/ of degree t can have

at most t distinct zeros modulo p.

31.5 The Chinese remainder theorem

Around A.D. 100, the Chinese mathematician Sun-Ts˘u solved the problem of ﬁnd-

ing those integers x that leave remainders 2, 3, and 2 when divided by 3, 5, and 7

respectively. One such solution is x D 23; all solutions are of the form 23 C 105k

31.5 The Chinese remainder theorem 951

for arbitrary integers k. The “Chinese remainder theorem” provides a correspon-

dence between a system of equations modulo a set of pairwise relatively prime

moduli (for example, 3, 5, and 7) and an equation modulo their product (for exam-

ple, 105).

The Chinese remainder theorem has two major applications. Let the inte-

ger n be factored as n D n 1 n 2 n k , where the factors n i are pairwise relatively

prime. First, the Chinese remainder theorem is a descriptive “structure theorem”

that describes the structure of Z n as identical to that of the Cartesian product

Z n 1 Z n 2 Z n k with componentwise addition and multiplication modulo n i

in the ith component. Second, this description helps us to design efﬁcient algo-

rithms, since working in each of the systems Z n i can be more efﬁcient (in terms of

bit operations) than working modulo n.

Theorem 31.27 (Chinese remainder theorem)

Let n D n 1 n 2 n k , where the n i are pairwise relatively prime. Consider the

correspondence

a $ .a 1 ; a 2 ; : : : ; a k / ; (31.27)

where a 2 Z n , a i 2 Z n i , and

a i D a mod n i

for i D 1; 2; : : : ; k. Then, mapping (31.27) is a one-to-one correspondence (bijec-

tion) between Z n and the Cartesian product Z n 1 Z n 2 Z n k . Operations per-

formed on the elements of Z n can be equivalently performed on the corresponding

k-tuples by performing the operations independently in each coordinate position in

the appropriate system. That is, if

a $ .a 1 ; a 2 ; : : : ; a k / ;

b $ .b 1 ; b 2 ; : : : ; b k / ;

then

.a C b/ mod n $ ..a 1 C b 1 / mod n 1 ; : : : ; .a k C b k / mod n k / ; (31.28)

.a b/ mod n $ ..a 1 b 1 / mod n 1 ; : : : ; .a k b k / mod n k / ; (31.29)

.ab/ mod n $ .a 1 b 1 mod n 1 ; : : : ; a k b k mod n k / : (31.30)

Proof Transforming between the two representations is fairly straightforward.

Going from a to .a 1 ; a 2 ; : : : ; a k / is quite easy and requires only k “mod” opera-

tions.

Computing a from inputs .a 1 ; a 2 ; : : : ; a k / is a bit more complicated. We begin

by deﬁning m i D n=n i for i D 1; 2; : : : ; k; thus m i is the product of all of the n j ’s

other than n i : m i D n 1 n 2 n i1 n iC1 n k . We next deﬁne

952 Chapter 31 Number-Theoretic Algorithms

c i D m i .m

1

i

mod n i / (31.31)

for i D 1; 2; : : : ; k. Equation (31.31) is always well deﬁned: since m i and n i are

relatively prime (by Theorem 31.6), Corollary 31.26 guarantees that m 1

i

mod n i

exists. Finally, we can compute a as a function of a 1 , a 2 , . . . , a k as follows:

a .a 1 c 1 C a 2 c 2 C C a k c k / .mod n/ : (31.32)

We now show that equation (31.32) ensures that a a i .mod n i / for i D

1; 2; : : : ; k. Note that if j ¤ i, then m j 0 .mod n i /, which implies that c j

m j 0 .mod n i /. Note also that c i 1 .mod n i /, from equation (31.31). We

thus have the appealing and useful correspondence

c i $ .0; 0; : : : ; 0; 1; 0; : : : ; 0/ ;

a vector that has 0s everywhere except in the ith coordinate, where it has a 1; the c i

thus form a “basis” for the representation, in a certain sense. For each i, therefore,

we have

a a i c i .mod n i /

a i m i .m

1

i

mod n i / .mod n i /

a i .mod n i / ;

which is what we wished to show: our method of computing a from the a i ’s pro-

duces a result a that satisﬁes the constraints a a i .mod n i / for i D 1; 2; : : : ; k.

The correspondence is one-to-one, since we can transform in both directions.

Finally, equations (31.28)–(31.30) follow directly from Exercise 31.1-7, since

x mod n i D .x mod n/ mod n i for any x and i D 1; 2; : : : ; k.

We shall use the following corollaries later in this chapter.

Corollary 31.28

If n 1 ; n 2 ; : : : ; n k are pairwise relatively prime and n D n 1 n 2 n k , then for any

integers a 1 ; a 2 ; : : : ; a k , the set of simultaneous equations

x a i .mod n i / ;

for i D 1; 2; : : : ; k, has a unique solution modulo n for the unknown x.

Corollary 31.29

If n 1 ; n 2 ; : : : ; n k are pairwise relatively prime and n D n 1 n 2 n k , then for all

integers x and a,

x a .mod n i /

for i D 1; 2; : : : ; k if and only if

x a .mod n/ :

31.5 The Chinese remainder theorem 953

0 1 2 3 4 5 6 7 8 9 10 11 12

0 0 40 15 55 30 5 45 20 60 35 10 50 25

1 26 1 41 16 56 31 6 46 21 61 36 11 51

2 52 27 2 42 17 57 32 7 47 22 62 37 12

3 13 53 28 3 43 18 58 33 8 48 23 63 38

4 39 14 54 29 4 44 19 59 34 9 49 24 64

Figure 31.3 An illustration of the Chinese remainder theorem for n1 D 5 and n2 D 13. For this

example, c1 D 26 and c2 D 40. In row i, column j is shown the value of a, modulo 65, such

that a mod 5 D i and a mod 13 D j . Note that row 0, column 0 contains a 0. Similarly, row 4,

column 12 contains a 64 (equivalent to 1). Since c1 D 26, moving down a row increases a by 26.

Similarly, c2 D 40 means that moving right by a column increases a by 40. Increasing a by 1

corresponds to moving diagonally downward and to the right, wrapping around from the bottom to

the top and from the right to the left.

As an example of the application of the Chinese remainder theorem, suppose we

are given the two equations

a 2 .mod 5/ ;

a 3 .mod 13/ ;

so that a 1 D 2, n 1 D m 2 D 5, a 2 D 3, and n 2 D m 1 D 13, and we wish

to compute a mod 65, since n D n 1 n 2 D 65. Because 13 1 2 .mod 5/ and

5 1 8 .mod 13/, we have

c 1 D 13.2 mod 5/ D 26 ;

c 2 D 5.8 mod 13/ D 40 ;

and

a 2 26 C 3 40 .mod 65/

52 C 120 .mod 65/

42 .mod 65/ :

See Figure 31.3 for an illustration of the Chinese remainder theorem, modulo 65.

Thus, we can work modulo n by working modulo n directly or by working in the

transformed representation using separate modulo n i computations, as convenient.

The computations are entirely equivalent.

Exercises

31.5-1

Find all solutions to the equations x 4 .mod 5/ and x 5 .mod 11/.

954 Chapter 31 Number-Theoretic Algorithms

31.5-2

Find all integers x that leave remainders 1, 2, 3 when divided by 9, 8, 7 respectively.

31.5-3

Argue that, under the deﬁnitions of Theorem 31.27, if gcd.a; n/ D 1, then

.a

1

mod n/ $ ..a

1

1

mod n 1 /; .a

1

2

mod n 2 /; : : : ; .a

1

k

mod n k // :

31.5-4

Under the deﬁnitions of Theorem 31.27, prove that for any polynomial f , the num-

ber of roots of the equation f .x/ 0 .mod n/ equals the product of the number

of roots of each of the equations f .x/ 0 .mod n 1 /, f .x/ 0 .mod n 2 /, . . . ,

f .x/ 0 .mod n k /.

31.6 Powers of an element

Just as we often consider the multiples of a given element a, modulo n, we consider

the sequence of powers of a, modulo n, where a 2 Z

n

:

a

0

; a

1

; a

2

; a

3

; : : : ; (31.33)

modulo n. Indexing from 0, the 0th value in this sequence is a 0

mod n D 1, and

the ith value is a i

mod n. For example, the powers of 3 modulo 7 are

i 0 1 2 3 4 5 6 7 8 9 10 11

3 i

mod 7 1 3 2 6 4 5 1 3 2 6 4 5

whereas the powers of 2 modulo 7 are

i 0 1 2 3 4 5 6 7 8 9 10 11

2 i

mod 7 1 2 4 1 2 4 1 2 4 1 2 4

In this section, let hai denote the subgroup of Z

n

generated by a by repeated

multiplication, and let ord n .a/ (the “order of a, modulo n”) denote the order of a

in Z

n

. For example, h2i D f1; 2; 4g in Z

7

, and ord 7 .2/ D 3. Using the deﬁnition of

the Euler phi function .n/ as the size of Z

n

(see Section 31.3), we now translate

Corollary 31.19 into the notation of Z

n

to obtain Euler’s theorem and specialize it

to Z

p

, where p is prime, to obtain Fermat’s theorem.

Theorem 31.30 (Euler’s theorem)

For any integer n > 1,

a

.n/

1 .mod n/ for all a 2 Z

n

:

31.6 Powers of an element 955

Theorem 31.31 (Fermat’s theorem)

If p is prime, then

a

p1

1 .mod p/ for all a 2 Z

p

:

Proof By equation (31.21), .p/ D p 1 if p is prime.

Fermat’s theorem applies to every element in Z p except 0, since 0 62 Z

p

. For all

a 2 Z p , however, we have a p a .mod p/ if p is prime.

If ord n .g/ D j Z

n

j, then every element in Z

n

is a power of g, modulo n, and

g is a primitive root or a generator of Z

n

. For example, 3 is a primitive root,

modulo 7, but 2 is not a primitive root, modulo 7. If Z

n

possesses a primitive

root, the group Z

n

is cyclic. We omit the proof of the following theorem, which is

proven by Niven and Zuckerman [265].

Theorem 31.32

The values of n > 1 for which Z

n

is cyclic are 2, 4, p e

, and 2p e

, for all primes

p > 2 and all positive integers e.

If g is a primitive root of Z

n

and a is any element of Z

n

, then there exists a ´ such

that g ´ a .mod n/. This ´ is a discrete logarithm or an index of a, modulo n,

to the base g; we denote this value as ind n;g .a/.

Theorem 31.33 (Discrete logarithm theorem)

If g is a primitive root of Z

n

, then the equation g x g y .mod n/ holds if and

only if the equation x y .mod .n// holds.

Proof Suppose ﬁrst that x y .mod .n//. Then, x D y C k.n/ for some

integer k. Therefore,

g

x

g

yCk.n/

.mod n/

g

y

.g

.n/

/

k

.mod n/

g

y

1

k

.mod n/ (by Euler’s theorem)

g

y

.mod n/ :

Conversely, suppose that g x g y .mod n/. Because the sequence of powers of g

generates every element of hgi and jhgij D .n/, Corollary 31.18 implies that

the sequence of powers of g is periodic with period .n/. Therefore, if g x g y

.mod n/, then we must have x y .mod .n//.

We now turn our attention to the square roots of 1, modulo a prime power. The

following theorem will be useful in our development of a primality-testing algo-

rithm in Section 31.8.

956 Chapter 31 Number-Theoretic Algorithms

Theorem 31.34

If p is an odd prime and e 1, then the equation

x

2

1 .mod p

e

/ (31.34)

has only two solutions, namely x D 1 and x D 1.

Proof Equation (31.34) is equivalent to

p

e

j .x 1/.x C 1/ :

Since p > 2, we can have p j .x 1/ or p j .x C 1/, but not both. (Otherwise,

by property (31.3), p would also divide their difference .x C 1/ .x 1/ D 2.)

If p − .x 1/, then gcd.p e ; x 1/ D 1, and by Corollary 31.5, we would have

p e j .x C 1/. That is, x 1 .mod p e /. Symmetrically, if p − .x C 1/,

then gcd.p e ; x C 1/ D 1, and Corollary 31.5 implies that p e j .x 1/, so that

x 1 .mod p e /. Therefore, either x 1 .mod p e / or x 1 .mod p e /.

A number x is a nontrivial square root of 1, modulo n, if it satisﬁes the equation

x 2 1 .mod n/ but x is equivalent to neither of the two “trivial” square roots:

1 or 1, modulo n. For example, 6 is a nontrivial square root of 1, modulo 35.

We shall use the following corollary to Theorem 31.34 in the correctness proof in

Section 31.8 for the Miller-Rabin primality-testing procedure.

Corollary 31.35

If there exists a nontrivial square root of 1, modulo n, then n is composite.

Proof By the contrapositive of Theorem 31.34, if there exists a nontrivial square

root of 1, modulo n, then n cannot be an odd prime or a power of an odd prime.

If x 2 1 .mod 2/, then x 1 .mod 2/, and so all square roots of 1, modulo 2,

are trivial. Thus, n cannot be prime. Finally, we must have n > 1 for a nontrivial

square root of 1 to exist. Therefore, n must be composite.

Raising to powers with repeated squaring

A frequently occurring operation in number-theoretic computations is raising one

number to a power modulo another number, also known as modular exponentia-

tion. More precisely, we would like an efﬁcient way to compute a b

mod n, where

a and b are nonnegative integers and n is a positive integer. Modular exponenti-

ation is an essential operation in many primality-testing routines and in the RSA

public-key cryptosystem. The method of repeated squaring solves this problem

efﬁciently using the binary representation of b.

Let hb k ; b k1 ; : : : ; b 1 ; b 0 i be the binary representation of b. (That is, the binary

representation is k C 1 bits long, b k is the most signiﬁcant bit, and b 0 is the least

31.6 Powers of an element 957

i 9 8 7 6 5 4 3 2 1 0

bi 1 0 0 0 1 1 0 0 0 0

c 1 2 4 8 17 35 70 140 280 560

d 7 49 157 526 160 241 298 166 67 1

Figure 31.4 The results of MODULAR-EXPONENTIATION when computing a

b

.mod n/, where

a D 7, b D 560 D h1000110000i, and n D 561. The values are shown after each execution of the

for loop. The ﬁnal result is 1.

signiﬁcant bit.) The following procedure computes a c

mod n as c is increased by

doublings and incrementations from 0 to b.

MODULAR-EXPONENTIATION.a; b; n/

1 c D 0

2 d D 1

3 let hb k ; b k1 ; : : : ; b 0 i be the binary representation of b

4 for i D k downto 0

5 c D 2c

6 d D .d d/ mod n

7 if b i == 1

8 c D c C 1

9 d D .d a/ mod n

10 return d

The essential use of squaring in line 6 of each iteration explains the name “repeated

squaring.” As an example, for a D 7, b D 560, and n D 561, the algorithm

computes the sequence of values modulo 561 shown in Figure 31.4; the sequence

of exponents used appears in the row of the table labeled by c.

The variable c is not really needed by the algorithm but is included for the fol-

lowing two-part loop invariant:

Just prior to each iteration of the for loop of lines 4–9,

1. The value of c is the same as the preﬁx hb k ; b k1 ; : : : ; b iC1 i of the binary

representation of b, and

2. d D a c

mod n.

We use this loop invariant as follows:

Initialization: Initially, i D k, so that the preﬁx hb k ; b k1 ; : : : ; b iC1 i is empty,

which corresponds to c D 0. Moreover, d D 1 D a 0

mod n.

958 Chapter 31 Number-Theoretic Algorithms

Maintenance: Let c 0

and d 0

denote the values of c and d at the end of an iteration

of the for loop, and thus the values prior to the next iteration. Each iteration

updates c 0 D 2c (if b i D 0) or c 0 D 2c C1 (if b i D 1), so that c will be correct

prior to the next iteration. If b i D 0, then d 0 D d 2

mod n D .a c / 2

mod n D

a 2c

mod n D a c

0

mod n. If b i D 1, then d 0 D d 2 a mod n D .a c / 2 a mod n D

a 2cC1

mod n D a c

0

mod n. In either case, d D a c

mod n prior to the next

iteration.

Termination: At termination, i D 1. Thus, c D b, since c has the value of the

preﬁx hb k ; b k1 ; : : : ; b 0 i of b’s binary representation. Hence d D a c

mod n D

a b

mod n.

If the inputs a, b, and n are ˇ-bit numbers, then the total number of arith-

metic operations required is O.ˇ/ and the total number of bit operations required

is O.ˇ 3 /.

Exercises

31.6-1

Draw a table showing the order of every element in Z

11

. Pick the smallest primitive

root g and compute a table giving ind 11;g .x/ for all x 2 Z

11

.

31.6-2

Give a modular exponentiation algorithm that examines the bits of b from right to

left instead of left to right.

31.6-3

Assuming that you know .n/, explain how to compute a 1

mod n for any a 2 Z

n

using the procedure MODULAR-EXPONENTIATION.

31.7 The RSA public-key cryptosystem

With a public-key cryptosystem, we can encrypt messages sent between two com-

municating parties so that an eavesdropper who overhears the encrypted messages

will not be able to decode them. A public-key cryptosystem also enables a party

to append an unforgeable “digital signature” to the end of an electronic message.

Such a signature is the electronic version of a handwritten signature on a paper doc-

ument. It can be easily checked by anyone, forged by no one, yet loses its validity

if any bit of the message is altered. It therefore provides authentication of both the

identity of the signer and the contents of the signed message. It is the perfect tool

31.7 The RSA public-key cryptosystem 959

for electronically signed business contracts, electronic checks, electronic purchase

orders, and other electronic communications that parties wish to authenticate.

The RSA public-key cryptosystem relies on the dramatic difference between the

ease of ﬁnding large prime numbers and the difﬁculty of factoring the product of

two large prime numbers. Section 31.8 describes an efﬁcient procedure for ﬁnding

large prime numbers, and Section 31.9 discusses the problem of factoring large

integers.

Public-key cryptosystems

In a public-key cryptosystem, each participant has both a public key and a secret

key. Each key is a piece of information. For example, in the RSA cryptosystem,

each key consists of a pair of integers. The participants “Alice” and “Bob” are

traditionally used in cryptography examples; we denote their public and secret

keys as P A , S A for Alice and P B , S B for Bob.

Each participant creates his or her own public and secret keys. Secret keys are

kept secret, but public keys can be revealed to anyone or even published. In fact,

it is often convenient to assume that everyone’s public key is available in a pub-

lic directory, so that any participant can easily obtain the public key of any other

participant.

The public and secret keys specify functions that can be applied to any message.

Let D denote the set of permissible messages. For example, D might be the set of

all ﬁnite-length bit sequences. In the simplest, and original, formulation of public-

key cryptography, we require that the public and secret keys specify one-to-one

functions from D to itself. We denote the function corresponding to Alice’s public

key P A by P A ./ and the function corresponding to her secret key S A by S A ./. The

functions P A ./ and S A ./ are thus permutations of D . We assume that the functions

P A ./ and S A ./ are efﬁciently computable given the corresponding key P A or S A .

The public and secret keys for any participant are a “matched pair” in that they

specify functions that are inverses of each other. That is,

M D S A .P A .M // ; (31.35)

M D P A .S A .M // (31.36)

for any message M 2 D . Transforming M with the two keys P A and S A succes-

sively, in either order, yields the message M back.

In a public-key cryptosystem, we require that no one but Alice be able to com-

pute the function S A ./ in any practical amount of time. This assumption is crucial

to keeping encrypted mail sent to Alice private and to knowing that Alice’s digi-

tal signatures are authentic. Alice must keep S A secret; if she does not, she loses

her uniqueness and the cryptosystem cannot provide her with unique capabilities.

The assumption that only Alice can compute S A ./ must hold even though everyone

960 Chapter 31 Number-Theoretic Algorithms

decrypt

communication channel

encrypt

Bob Alice

eavesdropper

M M P A S A

C

C D P A .M /

Figure 31.5 Encryption in a public key system. Bob encrypts the message M using Alice’s public

key PA and transmits the resulting ciphertext C D PA.M/ over a communication channel to Al-

ice. An eavesdropper who captures the transmitted ciphertext gains no information about M. Alice

receives C and decrypts it using her secret key to obtain the original message M D SA.C /.

knows P A and can compute P A ./, the inverse function to S A ./, efﬁciently. In order

to design a workable public-key cryptosystem, we must ﬁgure out how to create

a system in which we can reveal a transformation P A ./ without thereby revealing

how to compute the corresponding inverse transformation S A ./. This task appears

formidable, but we shall see how to accomplish it.

In a public-key cryptosystem, encryption works as shown in Figure 31.5. Sup-

pose Bob wishes to send Alice a message M encrypted so that it will look like

unintelligible gibberish to an eavesdropper. The scenario for sending the message

goes as follows.

Bob obtains Alice’s public key P A (from a public directory or directly from

Alice).

Bob computes the ciphertext C D P A .M / corresponding to the message M

and sends C to Alice.

When Alice receives the ciphertext C , she applies her secret key S A to retrieve

the original message: S A .C / D S A .P A .M // D M .

Because S A ./ and P A ./ are inverse functions, Alice can compute M from C . Be-

cause only Alice is able to compute S A ./, Alice is the only one who can compute M

from C . Because Bob encrypts M using P A ./, only Alice can understand the trans-

mitted message.

We can just as easily implement digital signatures within our formulation of a

public-key cryptosystem. (There are other ways of approaching the problem of

constructing digital signatures, but we shall not go into them here.) Suppose now

that Alice wishes to send Bob a digitally signed response M 0

. Figure 31.6 shows

how the digital-signature scenario proceeds.

Alice computes her digital signature for the message M 0

using her secret

key S A and the equation D S A .M 0 /.

31.7 The RSA public-key cryptosystem 961

sign

communication channel

verify

=? accept

Bob Alice

M 0

M 0

P A S A

.M 0 ; /

D S A .M 0 /

Figure 31.6 Digital signatures in a public-key system. Alice signs the message M

0

by appending

her digital signature D SA.M

0

/ to it. She transmits the message/signature pair .M

0

; / to Bob,

who veriﬁes it by checking the equation M

0

D PA. /. If the equation holds, he accepts .M

0

; / as

a message that Alice has signed.

Alice sends the message/signature pair .M 0 ; / to Bob.

When Bob receives .M 0 ; /, he can verify that it originated from Alice by us-

ing Alice’s public key to verify the equation M 0 D P A . /. (Presumably, M 0

contains Alice’s name, so Bob knows whose public key to use.) If the equation

holds, then Bob concludes that the message M 0

was actually signed by Alice.

If the equation fails to hold, Bob concludes either that the message M 0

or the

digital signature was corrupted by transmission errors or that the pair .M 0 ; /

is an attempted forgery.

Because a digital signature provides both authentication of the signer’s identity and

authentication of the contents of the signed message, it is analogous to a handwrit-

ten signature at the end of a written document.

A digital signature must be veriﬁable by anyone who has access to the signer’s

public key. A signed message can be veriﬁed by one party and then passed on to

other parties who can also verify the signature. For example, the message might

be an electronic check from Alice to Bob. After Bob veriﬁes Alice’s signature on

the check, he can give the check to his bank, who can then also verify the signature

and effect the appropriate funds transfer.

A signed message is not necessarily encrypted; the message can be “in the clear”

and not protected from disclosure. By composing the above protocols for encryp-

tion and for signatures, we can create messages that are both signed and encrypted.

The signer ﬁrst appends his or her digital signature to the message and then en-

crypts the resulting message/signature pair with the public key of the intended re-

cipient. The recipient decrypts the received message with his or her secret key to

obtain both the original message and its digital signature. The recipient can then

verify the signature using the public key of the signer. The corresponding com-

bined process using paper-based systems would be to sign the paper document and

962 Chapter 31 Number-Theoretic Algorithms

then seal the document inside a paper envelope that is opened only by the intended

recipient.

The RSA cryptosystem

In the RSA public-key cryptosystem, a participant creates his or her public and

secret keys with the following procedure:

1. Select at random two large prime numbers p and q such that p ¤ q. The primes

p and q might be, say, 1024 bits each.

2. Compute n D pq.

3. Select a small odd integer e that is relatively prime to .n/, which, by equa-

tion (31.20), equals .p 1/.q 1/.

4. Compute d as the multiplicative inverse of e, modulo .n/. (Corollary 31.26

guarantees that d exists and is uniquely deﬁned. We can use the technique of

Section 31.4 to compute d, given e and .n/.)

5. Publish the pair P D .e; n/ as the participant’s RSA public key.

6. Keep secret the pair S D .d; n/ as the participant’s RSA secret key.

For this scheme, the domain D is the set Z n . To transform a message M asso-

ciated with a public key P D .e; n/, compute

P.M / D M

e

mod n : (31.37)

To transform a ciphertext C associated with a secret key S D .d; n/, compute

S.C / D C

d

mod n : (31.38)

These equations apply to both encryption and signatures. To create a signature, the

signer applies his or her secret key to the message to be signed, rather than to a

ciphertext. To verify a signature, the public key of the signer is applied to it, rather

than to a message to be encrypted.

We can implement the public-key and secret-key operations using the procedure

MODULAR-EXPONENTIATION described in Section 31.6. To analyze the running

time of these operations, assume that the public key .e; n/ and secret key .d; n/

satisfy lg e D O.1/, lg d ˇ, and lg n ˇ. Then, applying a public key requires

O.1/ modular multiplications and uses O.ˇ 2 / bit operations. Applying a secret

key requires O.ˇ/ modular multiplications, using O.ˇ 3 / bit operations.

Theorem 31.36 (Correctness of RSA)

The RSA equations (31.37) and (31.38) deﬁne inverse transformations of Z n satis-

fying equations (31.35) and (31.36).

31.7 The RSA public-key cryptosystem 963

Proof From equations (31.37) and (31.38), we have that for any M 2 Z n ,

P.S.M // D S.P.M // D M

ed

.mod n/ :

Since e and d are multiplicative inverses modulo .n/ D .p 1/.q 1/,

ed D 1 C k.p 1/.q 1/

for some integer k. But then, if M 6 0 .mod p/, we have

M

ed

M.M

p1

/

k.q1/

.mod p/

M..M mod p/

p1

/

k.q1/

.mod p/

M.1/

k.q1/

.mod p/ (by Theorem 31.31)

M .mod p/ :

Also, M ed M .mod p/ if M 0 .mod p/. Thus,

M

ed

M .mod p/

for all M . Similarly,

M

ed

M .mod q/

for all M . Thus, by Corollary 31.29 to the Chinese remainder theorem,

M

ed

M .mod n/

for all M .

The security of the RSA cryptosystem rests in large part on the difﬁculty of fac-

toring large integers. If an adversary can factor the modulus n in a public key, then

the adversary can derive the secret key from the public key, using the knowledge

of the factors p and q in the same way that the creator of the public key used them.

Therefore, if factoring large integers is easy, then breaking the RSA cryptosystem

is easy. The converse statement, that if factoring large integers is hard, then break-

ing RSA is hard, is unproven. After two decades of research, however, no easier

method has been found to break the RSA public-key cryptosystem than to factor

the modulus n. And as we shall see in Section 31.9, factoring large integers is sur-

prisingly difﬁcult. By randomly selecting and multiplying together two 1024-bit

primes, we can create a public key that cannot be “broken” in any feasible amount

of time with current technology. In the absence of a fundamental breakthrough in

the design of number-theoretic algorithms, and when implemented with care fol-

lowing recommended standards, the RSA cryptosystem is capable of providing a

high degree of security in applications.

In order to achieve security with the RSA cryptosystem, however, we should

use integers that are quite long—hundreds or even more than one thousand bits

964 Chapter 31 Number-Theoretic Algorithms

long—to resist possible advances in the art of factoring. At the time of this

writing (2009), RSA moduli were commonly in the range of 768 to 2048 bits.

To create moduli of such sizes, we must be able to ﬁnd large primes efﬁciently.

Section 31.8 addresses this problem.

For efﬁciency, RSA is often used in a “hybrid” or “key-management” mode

with fast non-public-key cryptosystems. With such a system, the encryption and

decryption keys are identical. If Alice wishes to send a long message M to Bob

privately, she selects a random key K for the fast non-public-key cryptosystem and

encrypts M using K, obtaining ciphertext C . Here, C is as long as M , but K

is quite short. Then, she encrypts K using Bob’s public RSA key. Since K is

short, computing P B .K/ is fast (much faster than computing P B .M /). She then

transmits .C; P B .K// to Bob, who decrypts P B .K/ to obtain K and then uses K

to decrypt C , obtaining M .

We can use a similar hybrid approach to make digital signatures efﬁciently.

This approach combines RSA with a public collision-resistant hash function h—a

function that is easy to compute but for which it is computationally infeasible to

ﬁnd two messages M and M 0

such that h.M / D h.M 0 /. The value h.M / is

a short (say, 256-bit) “ﬁngerprint” of the message M . If Alice wishes to sign a

message M , she ﬁrst applies h to M to obtain the ﬁngerprint h.M /, which she

then encrypts with her secret key. She sends .M; S A .h.M /// to Bob as her signed

version of M . Bob can verify the signature by computing h.M / and verifying

that P A applied to S A .h.M // as received equals h.M /. Because no one can create

two messages with the same ﬁngerprint, it is computationally infeasible to alter a

signed message and preserve the validity of the signature.

Finally, we note that the use of certiﬁcates makes distributing public keys much

easier. For example, assume there is a “trusted authority” T whose public key

is known by everyone. Alice can obtain from T a signed message (her certiﬁcate)

stating that “Alice’s public key is P A .” This certiﬁcate is “self-authenticating” since

everyone knows P T . Alice can include her certiﬁcate with her signed messages,

so that the recipient has Alice’s public key immediately available in order to verify

her signature. Because her key was signed by T , the recipient knows that Alice’s

key is really Alice’s.

Exercises

31.7-1

Consider an RSA key set with p D 11, q D 29, n D 319, and e D 3. What

value of d should be used in the secret key? What is the encryption of the message

M D 100?

31.8 Primality testing 965

31.7-2

Prove that if Alice’s public exponent e is 3 and an adversary obtains Alice’s secret

exponent d, where 0 < d < .n/, then the adversary can factor Alice’s modulus n

in time polynomial in the number of bits in n. (Although you are not asked to prove

it, you may be interested to know that this result remains true even if the condition

e D 3 is removed. See Miller [255].)

31.7-3 ?

Prove that RSA is multiplicative in the sense that

P A .M 1 /P A .M 2 / P A .M 1 M 2 / .mod n/ :

Use this fact to prove that if an adversary had a procedure that could efﬁciently

decrypt 1 percent of messages from Z n encrypted with P A , then he could employ

a probabilistic algorithm to decrypt every message encrypted with P A with high

probability.

? 31.8 Primality testing

In this section, we consider the problem of ﬁnding large primes. We begin with a

discussion of the density of primes, proceed to examine a plausible, but incomplete,

approach to primality testing, and then present an effective randomized primality

test due to Miller and Rabin.

The density of prime numbers

For many applications, such as cryptography, we need to ﬁnd large “random”

primes. Fortunately, large primes are not too rare, so that it is feasible to test

random integers of the appropriate size until we ﬁnd a prime. The prime distribu-

tion function .n/ speciﬁes the number of primes that are less than or equal to n.

For example, .10/ D 4, since there are 4 prime numbers less than or equal to 10,

namely, 2, 3, 5, and 7. The prime number theorem gives a useful approximation

to .n/.

Theorem 31.37 (Prime number theorem)

lim

n!1

.n/

n= ln n

D 1 :

The approximation n= ln n gives reasonably accurate estimates of .n/ even

for small n. For example, it is off by less than 6% at n D 10 9

, where .n/ D

966 Chapter 31 Number-Theoretic Algorithms

50,847,534 and n= ln n 48,254,942. (To a number theorist, 10 9

is a small num-

ber.)

We can view the process of randomly selecting an integer n and determining

whether it is prime as a Bernoulli trial (see Section C.4). By the prime number

theorem, the probability of a success—that is, the probability that n is prime—is

approximately 1= ln n. The geometric distribution tells us how many trials we need

to obtain a success, and by equation (C.32), the expected number of trials is ap-

proximately ln n. Thus, we would expect to examine approximately ln n integers

chosen randomly near n in order to ﬁnd a prime that is of the same length as n.

For example, we expect that ﬁnding a 1024-bit prime would require testing ap-

proximately ln 2 1024 710 randomly chosen 1024-bit numbers for primality. (Of

course, we can cut this ﬁgure in half by choosing only odd integers.)

In the remainder of this section, we consider the problem of determining whether

or not a large odd integer n is prime. For notational convenience, we assume that n

has the prime factorization

n D p

e 1

1

p

e 2

2

p

e r

r

; (31.39)

where r 1, p 1 ; p 2 ; : : : ; p r are the prime factors of n, and e 1 ; e 2 ; : : : ; e r are posi-

tive integers. The integer n is prime if and only if r D 1 and e 1 D 1.

One simple approach to the problem of testing for primality is trial division. We

try dividing n by each integer 2; 3; : : : ; b

p

nc. (Again, we may skip even integers

greater than 2.) It is easy to see that n is prime if and only if none of the trial divi-

sors divides n. Assuming that each trial division takes constant time, the worst-case

running time is ‚.

p

n/, which is exponential in the length of n. (Recall that if n

is encoded in binary using ˇ bits, then ˇ D dlg.n C 1/e, and so

p

n D ‚.2 ˇ=2 /.)

Thus, trial division works well only if n is very small or happens to have a small

prime factor. When it works, trial division has the advantage that it not only de-

termines whether n is prime or composite, but also determines one of n’s prime

factors if n is composite.

In this section, we are interested only in ﬁnding out whether a given number n

is prime; if n is composite, we are not concerned with ﬁnding its prime factor-

ization. As we shall see in Section 31.9, computing the prime factorization of a

number is computationally expensive. It is perhaps surprising that it is much easier

to tell whether or not a given number is prime than it is to determine the prime

factorization of the number if it is not prime.

Pseudoprimality testing

We now consider a method for primality testing that “almost works” and in fact

is good enough for many practical applications. Later on, we shall present a re-

31.8 Primality testing 967

ﬁnement of this method that removes the small defect. Let Z C

n

denote the nonzero

elements of Z n :

Z C

n

D f1; 2; : : : ; n 1g :

If n is prime, then Z C

n

D Z

n

.

We say that n is a base-a pseudoprime if n is composite and

a

n1

1 .mod n/ : (31.40)

Fermat’s theorem (Theorem 31.31) implies that if n is prime, then n satisﬁes equa-

tion (31.40) for every a in Z C

n

. Thus, if we can ﬁnd any a 2 Z C

n

such that n does

not satisfy equation (31.40), then n is certainly composite. Surprisingly, the con-

verse almost holds, so that this criterion forms an almost perfect test for primality.

We test to see whether n satisﬁes equation (31.40) for a D 2. If not, we declare n

to be composite by returning COMPOSITE. Otherwise, we return PRIME, guessing

that n is prime (when, in fact, all we know is that n is either prime or a base-2

pseudoprime).

The following procedure pretends in this manner to be checking the primality

of n. It uses the procedure MODULAR-EXPONENTIATION from Section 31.6. We

assume that the input n is an odd integer greater than 2.

PSEUDOPRIME.n/

1 if MODULAR-EXPONENTIATION.2; n 1; n/ 6 1 .mod n/

2 return COMPOSITE // deﬁnitely

3 else return PRIME // we hope!

This procedure can make errors, but only of one type. That is, if it says that n

is composite, then it is always correct. If it says that n is prime, however, then it

makes an error only if n is a base-2 pseudoprime.

How often does this procedure err? Surprisingly rarely. There are only 22 values

of n less than 10,000 for which it errs; the ﬁrst four such values are 341, 561,

645, and 1105. We won’t prove it, but the probability that this program makes an

error on a randomly chosen ˇ-bit number goes to zero as ˇ ! 1. Using more

precise estimates due to Pomerance [279] of the number of base-2 pseudoprimes of

a given size, we may estimate that a randomly chosen 512-bit number that is called

prime by the above procedure has less than one chance in 10 20

of being a base-2

pseudoprime, and a randomly chosen 1024-bit number that is called prime has less

than one chance in 10 41

of being a base-2 pseudoprime. So if you are merely

trying to ﬁnd a large prime for some application, for all practical purposes you

almost never go wrong by choosing large numbers at random until one of them

causes PSEUDOPRIME to return PRIME. But when the numbers being tested for

primality are not randomly chosen, we need a better approach for testing primality.

968 Chapter 31 Number-Theoretic Algorithms

As we shall see, a little more cleverness, and some randomization, will yield a

primality-testing routine that works well on all inputs.

Unfortunately, we cannot entirely eliminate all the errors by simply checking

equation (31.40) for a second base number, say a D 3, because there exist com-

posite integers n, known as Carmichael numbers, that satisfy equation (31.40) for

all a 2 Z

n

. (We note that equation (31.40) does fail when gcd.a; n/ > 1—that

is, when a 62 Z

n

—but hoping to demonstrate that n is composite by ﬁnding such

an a can be difﬁcult if n has only large prime factors.) The ﬁrst three Carmichael

numbers are 561, 1105, and 1729. Carmichael numbers are extremely rare; there

are, for example, only 255 of them less than 100,000,000. Exercise 31.8-2 helps

explain why they are so rare.

We next show how to improve our primality test so that it won’t be fooled by

Carmichael numbers.

The Miller-Rabin randomized primality test

The Miller-Rabin primality test overcomes the problems of the simple test PSEU-

DOPRIME with two modiﬁcations:

It tries several randomly chosen base values a instead of just one base value.

While computing each modular exponentiation, it looks for a nontrivial square

root of 1, modulo n, during the ﬁnal set of squarings. If it ﬁnds one, it stops

and returns COMPOSITE. Corollary 31.35 from Section 31.6 justiﬁes detecting

composites in this manner.

The pseudocode for the Miller-Rabin primality test follows. The input n > 2 is

the odd number to be tested for primality, and s is the number of randomly cho-

sen base values from Z C

n

to be tried. The code uses the random-number generator

RANDOM described on page 117: RANDOM.1; n 1/ returns a randomly chosen

integer a satisfying 1 a n1. The code uses an auxiliary procedure WITNESS

such that WITNESS.a; n/ is TRUE if and only if a is a “witness” to the composite-

ness of n—that is, if it is possible using a to prove (in a manner that we shall see)

that n is composite. The test WITNESS.a; n/ is an extension of, but more effective

than, the test

a

n1

6 1 .mod n/

that formed the basis (using a D 2) for PSEUDOPRIME. We ﬁrst present and

justify the construction of WITNESS, and then we shall show how we use it in the

Miller-Rabin primality test. Let n 1 D 2 t u where t 1 and u is odd; i.e.,

the binary representation of n 1 is the binary representation of the odd integer u

followed by exactly t zeros. Therefore, a n1 .a u / 2

t

.mod n/, so that we can

31.8 Primality testing 969

compute a n1

mod n by ﬁrst computing a u

mod n and then squaring the result t

times successively.

WITNESS.a; n/

1 let t and u be such that t 1, u is odd, and n 1 D 2 t u

2 x 0 D MODULAR-EXPONENTIATION.a; u; n/

3 for i D 1 to t

4 x i D x 2

i1

mod n

5 if x i == 1 and x i1 ¤ 1 and x i1 ¤ n 1

6 return TRUE

7 if x t ¤ 1

8 return TRUE

9 return FALSE

This pseudocode for WITNESS computes a n1

mod n by ﬁrst computing the

value x 0 D a u

mod n in line 2 and then squaring the result t times in a row in the

for loop of lines 3–6. By induction on i, the sequence x 0 , x 1 , . . . , x t of values

computed satisﬁes the equation x i a 2

i

u .mod n/ for i D 0; 1; : : : ; t, so that in

particular x t a n1 .mod n/. After line 4 performs a squaring step, however,

the loop may terminate early if lines 5–6 detect that a nontrivial square root of 1

has just been discovered. (We shall explain these tests shortly.) If so, the algo-

rithm stops and returns TRUE. Lines 7–8 return TRUE if the value computed for

x t a n1 .mod n/ is not equal to 1, just as the PSEUDOPRIME procedure returns

COMPOSITE in this case. Line 9 returns FALSE if we haven’t returned TRUE in

lines 6 or 8.

We now argue that if WITNESS.a; n/ returns TRUE, then we can construct a

proof that n is composite using a as a witness.

If WITNESS returns TRUE from line 8, then it has discovered that x t D

a n1

mod n ¤ 1. If n is prime, however, we have by Fermat’s theorem (Theo-

rem 31.31) that a n1 1 .mod n/ for all a 2 Z C

n

. Therefore, n cannot be prime,

and the equation a n1

mod n ¤ 1 proves this fact.

If WITNESS returns TRUE from line 6, then it has discovered that x i1 is a non-

trivial square root of 1, modulo n, since we have that x i1 6 ˙1 .mod n/ yet

x i x 2

i1

1 .mod n/. Corollary 31.35 states that only if n is composite can

there exist a nontrivial square root of 1 modulo n, so that demonstrating that x i1

is a nontrivial square root of 1 modulo n proves that n is composite.

This completes our proof of the correctness of WITNESS. If we ﬁnd that the call

WITNESS.a; n/ returns TRUE, then n is surely composite, and the witness a, along

with the reason that the procedure returns TRUE (did it return from line 6 or from

line 8?), provides a proof that n is composite.

970 Chapter 31 Number-Theoretic Algorithms

At this point, we brieﬂy present an alternative description of the behavior of

WITNESS as a function of the sequence X D hx 0 ; x 1 ; : : : ; x t i, which we shall ﬁnd

useful later on, when we analyze the efﬁciency of the Miller-Rabin primality test.

Note that if x i D 1 for some 0 i < t, WITNESS might not compute the rest

of the sequence. If it were to do so, however, each value x iC1 ; x iC2 ; : : : ; x t would

be 1, and we consider these positions in the sequence X as being all 1s. We have

four cases:

1. X D h: : : ; di, where d ¤ 1: the sequence X does not end in 1. Return TRUE

in line 8; a is a witness to the compositeness of n (by Fermat’s Theorem).

2. X D h1; 1; : : : ; 1i: the sequence X is all 1s. Return FALSE; a is not a witness

to the compositeness of n.

3. X D h: : : ; 1; 1; : : : ; 1i: the sequence X ends in 1, and the last non-1 is equal

to 1. Return FALSE; a is not a witness to the compositeness of n.

4. X D h: : : ; d; 1; : : : ; 1i, where d ¤ ˙1: the sequence X ends in 1, but the last

non-1 is not 1. Return TRUE in line 6; a is a witness to the compositeness

of n, since d is a nontrivial square root of 1.

We now examine the Miller-Rabin primality test based on the use of WITNESS.

Again, we assume that n is an odd integer greater than 2.

MILLER-RABIN.n; s/

1 for j D 1 to s

2 a D RANDOM.1; n 1/

3 if WITNESS.a; n/

4 return COMPOSITE // deﬁnitely

5 return PRIME // almost surely

The procedure MILLER-RABIN is a probabilistic search for a proof that n is

composite. The main loop (beginning on line 1) picks up to s random values of a

from Z C

n

(line 2). If one of the a’s picked is a witness to the compositeness of n,

then MILLER-RABIN returns COMPOSITE on line 4. Such a result is always cor-

rect, by the correctness of WITNESS. If MILLER-RABIN ﬁnds no witness in s

trials, then the procedure assumes that this is because no witnesses exist, and there-

fore it assumes that n is prime. We shall see that this result is likely to be correct

if s is large enough, but that there is still a tiny chance that the procedure may be

unlucky in its choice of a’s and that witnesses do exist even though none has been

found.

To illustrate the operation of MILLER-RABIN, let n be the Carmichael num-

ber 561, so that n 1 D 560 D 2 4 35, t D 4, and u D 35. If the pro-

cedure chooses a D 7 as a base, Figure 31.4 in Section 31.6 shows that WIT-

NESS computes x 0 a 35 241 .mod 561/ and thus computes the sequence

31.8 Primality testing 971

X D h241; 298; 166; 67; 1i. Thus, WITNESS discovers a nontrivial square root

of 1 in the last squaring step, since a 280 67 .mod n/ and a 560 1 .mod n/.

Therefore, a D 7 is a witness to the compositeness of n, WITNESS.7; n/ returns

TRUE, and MILLER-RABIN returns COMPOSITE.

If n is a ˇ-bit number, MILLER-RABIN requires O.sˇ/ arithmetic operations

and O.sˇ 3 / bit operations, since it requires asymptotically no more work than s

modular exponentiations.

Error rate of the Miller-Rabin primality test

If MILLER-RABIN returns PRIME, then there is a very slim chance that it has made

an error. Unlike PSEUDOPRIME, however, the chance of error does not depend

on n; there are no bad inputs for this procedure. Rather, it depends on the size of s

and the “luck of the draw” in choosing base values a. Moreover, since each test is

more stringent than a simple check of equation (31.40), we can expect on general

principles that the error rate should be small for randomly chosen integers n. The

following theorem presents a more precise argument.

Theorem 31.38

If n is an odd composite number, then the number of witnesses to the composite-

ness of n is at least .n 1/=2.

Proof The proof shows that the number of nonwitnesses is at most .n 1/=2,

which implies the theorem.

We start by claiming that any nonwitness must be a member of Z

n

. Why?

Consider any nonwitness a. It must satisfy a n1 1 .mod n/ or, equivalently,

a a n2 1 .mod n/. Thus, the equation ax 1 .mod n/ has a solution,

namely a n2

. By Corollary 31.21, gcd.a; n/ j 1, which in turn implies that

gcd.a; n/ D 1. Therefore, a is a member of Z

n

; all nonwitnesses belong to Z

n

.

To complete the proof, we show that not only are all nonwitnesses contained

in Z

n

, they are all contained in a proper subgroup B of Z

n

(recall that we say B

is a proper subgroup of Z

n

when B is subgroup of Z

n

but B is not equal to Z

n

).

By Corollary 31.16, we then have jBj j Z

n

j =2. Since j Z

n

j n 1, we obtain

jBj .n 1/=2. Therefore, the number of nonwitnesses is at most .n 1/=2, so

that the number of witnesses must be at least .n 1/=2.

We now show how to ﬁnd a proper subgroup B of Z

n

containing all of the

nonwitnesses. We break the proof into two cases.

Case 1: There exists an x 2 Z

n

such that

x

n1

6 1 .mod n/ :

972 Chapter 31 Number-Theoretic Algorithms

In other words, n is not a Carmichael number. Because, as we noted earlier,

Carmichael numbers are extremely rare, case 1 is the main case that arises “in

practice” (e.g., when n has been chosen randomly and is being tested for primal-

ity).

Let B D fb 2 Z

n

W b n1 1 .mod n/g. Clearly, B is nonempty, since 1 2 B.

Since B is closed under multiplication modulo n, we have that B is a subgroup

of Z

n

by Theorem 31.14. Note that every nonwitness belongs to B, since a non-

witness a satisﬁes a n1 1 .mod n/. Since x 2 Z

n

B, we have that B is a

proper subgroup of Z

n

.

Case 2: For all x 2 Z

n

,

x

n1

1 .mod n/ : (31.41)

In other words, n is a Carmichael number. This case is extremely rare in prac-

tice. However, the Miller-Rabin test (unlike a pseudo-primality test) can efﬁciently

determine that Carmichael numbers are composite, as we now show.

In this case, n cannot be a prime power. To see why, let us suppose to the

contrary that n D p e

, where p is a prime and e > 1. We derive a contradiction

as follows. Since we assume that n is odd, p must also be odd. Theorem 31.32

implies that Z

n

is a cyclic group: it contains a generator g such that ord n .g/ D

j Z

n

j D .n/ D p e .1 1=p/ D .p 1/p e1

. (The formula for .n/ comes from

equation (31.20).) By equation (31.41), we have g n1 1 .mod n/. Then the

discrete logarithm theorem (Theorem 31.33, taking y D 0) implies that n 1 0

.mod .n//, or

.p 1/p

e1

j p

e

1 :

This is a contradiction for e > 1, since .p 1/p e1

is divisible by the prime p

but p e 1 is not. Thus, n is not a prime power.

Since the odd composite number n is not a prime power, we decompose it into

a product n 1 n 2 , where n 1 and n 2 are odd numbers greater than 1 that are relatively

prime to each other. (There may be several ways to decompose n, and it does not

matter which one we choose. For example, if n D p

e 1

1 p

e 2

2 p e r

r

, then we can

choose n 1 D p

e 1

1 and n 2 D p

e 2

2 p

e 3

3 p e r

r

.)

Recall that we deﬁne t and u so that n 1 D 2 t u, where t 1 and u is odd, and

that for an input a, the procedure WITNESS computes the sequence

X D ha

u

; a

2u

; a

2

2

u

; : : : ; a

2

t

u

i

(all computations are performed modulo n).

Let us call a pair .; j / of integers acceptable if 2 Z

n

, j 2 f0; 1; : : : ; tg, and

2

j

u

1 .mod n/ :

31.8 Primality testing 973

Acceptable pairs certainly exist since u is odd; we can choose D n 1 and

j D 0, so that .n1; 0/ is an acceptable pair. Now pick the largest possible j such

that there exists an acceptable pair .; j /, and ﬁx so that .; j / is an acceptable

pair. Let

B D fx 2 Z

n

W x

2

j

u

˙1 .mod n/g :

Since B is closed under multiplication modulo n, it is a subgroup of Z

n

. By Theo-

rem 31.15, therefore, jBj divides j Z

n

j. Every nonwitness must be a member of B,

since the sequence X produced by a nonwitness must either be all 1s or else contain

a 1 no later than the j th position, by the maximality of j . (If .a; j 0 / is acceptable,

where a is a nonwitness, we must have j 0 j by how we chose j .)

We now use the existence of to demonstrate that there exists a w 2 Z

n

B,

and hence that B is a proper subgroup of Z

n

. Since 2

j

u 1 .mod n/, we have

2

j

u 1 .mod n 1 / by Corollary 31.29 to the Chinese remainder theorem. By

Corollary 31.28, there exists a w simultaneously satisfying the equations

w .mod n 1 / ;

w 1 .mod n 2 / :

Therefore,

w

2

j

u

1 .mod n 1 / ;

w

2

j

u

1 .mod n 2 / :

By Corollary 31.29, w 2

j

u 6 1 .mod n 1 / implies w 2

j

u 6 1 .mod n/, and

w 2

j

u 6 1 .mod n 2 / implies w 2

j

u 6 1 .mod n/. Hence, we conclude that

w 2

j

u 6 ˙1 .mod n/, and so w 62 B.

It remains to show that w 2 Z

n

, which we do by ﬁrst working separately mod-

ulo n 1 and modulo n 2 . Working modulo n 1 , we observe that since 2 Z

n

, we

have that gcd.; n/ D 1, and so also gcd.; n 1 / D 1; if does not have any com-

mon divisors with n, then it certainly does not have any common divisors with n 1 .

Since w .mod n 1 /, we see that gcd.w; n 1 / D 1. Working modulo n 2 , we

observe that w 1 .mod n 2 / implies gcd.w; n 2 / D 1. To combine these results,

we use Theorem 31.6, which implies that gcd.w; n 1 n 2 / D gcd.w; n/ D 1. That is,

w 2 Z

n

.

Therefore w 2 Z

n

B, and we ﬁnish case 2 with the conclusion that B is a

proper subgroup of Z

n

.

In either case, we see that the number of witnesses to the compositeness of n is

at least .n 1/=2.

Theorem 31.39

For any odd integer n > 2 and positive integer s, the probability that MILLER-

RABIN.n; s/ errs is at most 2 s

.

974 Chapter 31 Number-Theoretic Algorithms

Proof Using Theorem 31.38, we see that if n is composite, then each execution of

the for loop of lines 1–4 has a probability of at least 1=2 of discovering a witness x

to the compositeness of n. MILLER-RABIN makes an error only if it is so unlucky

as to miss discovering a witness to the compositeness of n on each of the s iterations

of the main loop. The probability of such a sequence of misses is at most 2 s

.

If n is prime, MILLER-RABIN always reports PRIME, and if n is composite, the

chance that MILLER-RABIN reports PRIME is at most 2 s

.

When applying MILLER-RABIN to a large randomly chosen integer n, however,

we need to consider as well the prior probability that n is prime, in order to cor-

rectly interpret MILLER-RABIN’s result. Suppose that we ﬁx a bit length ˇ and

choose at random an integer n of length ˇ bits to be tested for primality. Let A

denote the event that n is prime. By the prime number theorem (Theorem 31.37),

the probability that n is prime is approximately

Pr fAg 1= ln n

1:443=ˇ :

Now let B denote the event that MILLER-RABIN returns PRIME. We have that

Pr

˚

B j A

D 0 (or equivalently, that Pr fB j Ag D 1) and Pr

˚

B j A

2 s

(or

equivalently, that Pr

˚

B j A

> 1 2 s

).

But what is Pr fA j Bg, the probability that n is prime, given that MILLER-

RABIN has returned PRIME? By the alternate form of Bayes’s theorem (equa-

tion (C.18)) we have

Pr fA j Bg D

Pr fAg Pr fB j Ag

Pr fAg Pr fB j Ag C Pr

˚

A

Pr

˚

B j A

1

1 C 2 s .ln n 1/

:

This probability does not exceed 1=2 until s exceeds lg.ln n 1/. Intuitively, that

many initial trials are needed just for the conﬁdence derived from failing to ﬁnd a

witness to the compositeness of n to overcome the prior bias in favor of n being

composite. For a number with ˇ D 1024 bits, this initial testing requires about

lg.ln n 1/ lg.ˇ=1:443/

9

trials. In any case, choosing s D 50 should sufﬁce for almost any imaginable

application.

In fact, the situation is much better. If we are trying to ﬁnd large primes by

applying MILLER-RABIN to large randomly chosen odd integers, then choosing

a small value of s (say 3) is very unlikely to lead to erroneous results, though

31.9 Integer factorization 975

we won’t prove it here. The reason is that for a randomly chosen odd composite

integer n, the expected number of nonwitnesses to the compositeness of n is likely

to be very much smaller than .n 1/=2.

If the integer n is not chosen randomly, however, the best that can be proven is

that the number of nonwitnesses is at most .n 1/=4, using an improved version

of Theorem 31.38. Furthermore, there do exist integers n for which the number of

nonwitnesses is .n 1/=4.

Exercises

31.8-1

Prove that if an odd integer n > 1 is not a prime or a prime power, then there exists

a nontrivial square root of 1 modulo n.

31.8-2 ?

It is possible to strengthen Euler’s theorem slightly to the form

a

.n/

1 .mod n/ for all a 2 Z

n

;

where n D p

e 1

1

p e r

r

and .n/ is deﬁned by

.n/ D lcm..p

e 1

1 /; : : : ; .p

e r

r

// : (31.42)

Prove that .n/ j .n/. A composite number n is a Carmichael number if

.n/ j n 1. The smallest Carmichael number is 561 D 3 11 17; here,

.n/ D lcm.2; 10; 16/ D 80, which divides 560. Prove that Carmichael num-

bers must be both “square-free” (not divisible by the square of any prime) and the

product of at least three primes. (For this reason, they are not very common.)

31.8-3

Prove that if x is a nontrivial square root of 1, modulo n, then gcd.x 1; n/ and

gcd.x C 1; n/ are both nontrivial divisors of n.

? 31.9 Integer factorization

Suppose we have an integer n that we wish to factor, that is, to decompose into a

product of primes. The primality test of the preceding section may tell us that n is

composite, but it does not tell us the prime factors of n. Factoring a large integer n

seems to be much more difﬁcult than simply determining whether n is prime or

composite. Even with today’s supercomputers and the best algorithms to date, we

cannot feasibly factor an arbitrary 1024-bit number.

976 Chapter 31 Number-Theoretic Algorithms

Pollard’s rho heuristic

Trial division by all integers up to R is guaranteed to factor completely any number

up to R 2

. For the same amount of work, the following procedure, POLLARD-RHO,

factors any number up to R 4

(unless we are unlucky). Since the procedure is only

a heuristic, neither its running time nor its success is guaranteed, although the

procedure is highly effective in practice. Another advantage of the POLLARD-

RHO procedure is that it uses only a constant number of memory locations. (If you

wanted to, you could easily implement POLLARD-RHO on a programmable pocket

calculator to ﬁnd factors of small numbers.)

POLLARD-RHO.n/

1 i D 1

2 x 1 D RANDOM.0; n 1/

3 y D x 1

4 k D 2

5 while TRUE

6 i D i C 1

7 x i D .x 2

i1

1/ mod n

8 d D gcd.y x i ; n/

9 if d ¤ 1 and d ¤ n

10 print d

11 if i == k

12 y D x i

13 k D 2k

The procedure works as follows. Lines 1–2 initialize i to 1 and x 1 to a randomly

chosen value in Z n . The while loop beginning on line 5 iterates forever, searching

for factors of n. During each iteration of the while loop, line 7 uses the recurrence

x i D .x

2

i1

1/ mod n (31.43)

to produce the next value of x i in the inﬁnite sequence

x 1 ; x 2 ; x 3 ; x 4 ; : : : ; (31.44)

with line 6 correspondingly incrementing i. The pseudocode is written using sub-

scripted variables x i for clarity, but the program works the same if all of the sub-

scripts are dropped, since only the most recent value of x i needs to be maintained.

With this modiﬁcation, the procedure uses only a constant number of memory lo-

cations.

Every so often, the program saves the most recently generated x i value in the

variable y. Speciﬁcally, the values that are saved are the ones whose subscripts are

powers of 2:

31.9 Integer factorization 977

x 1 ; x 2 ; x 4 ; x 8 ; x 16 ; : : : :

Line 3 saves the value x 1 , and line 12 saves x k whenever i is equal to k. The

variable k is initialized to 2 in line 4, and line 13 doubles it whenever line 12

updates y. Therefore, k follows the sequence 1; 2; 4; 8; : : : and always gives the

subscript of the next value x k to be saved in y.

Lines 8–10 try to ﬁnd a factor of n, using the saved value of y and the cur-

rent value of x i . Speciﬁcally, line 8 computes the greatest common divisor

d D gcd.y x i ; n/. If line 9 ﬁnds d to be a nontrivial divisor of n, then line 10

prints d.

This procedure for ﬁnding a factor may seem somewhat mysterious at ﬁrst.

Note, however, that POLLARD-RHO never prints an incorrect answer; any num-

ber it prints is a nontrivial divisor of n. POLLARD-RHO might not print anything

at all, though; it comes with no guarantee that it will print any divisors. We shall

see, however, that we have good reason to expect POLLARD-RHO to print a fac-

tor p of n after ‚.

p

p/ iterations of the while loop. Thus, if n is composite, we

can expect this procedure to discover enough divisors to factor n completely after

approximately n 1=4

updates, since every prime factor p of n except possibly the

largest one is less than

p

n.

We begin our analysis of how this procedure behaves by studying how long

it takes a random sequence modulo n to repeat a value. Since Z n is ﬁnite, and

since each value in the sequence (31.44) depends only on the previous value, the

sequence (31.44) eventually repeats itself. Once we reach an x i such that x i D x j

for some j < i, we are in a cycle, since x iC1 D x j C1 , x iC2 D x j C2 , and so on.

The reason for the name “rho heuristic” is that, as Figure 31.7 shows, we can draw

the sequence x 1 ; x 2 ; : : : ; x j 1 as the “tail” of the rho and the cycle x j ; x j C1 ; : : : ; x i

as the “body” of the rho.

Let us consider the question of how long it takes for the sequence of x i to repeat.

This information is not exactly what we need, but we shall see later how to modify

the argument. For the purpose of this estimation, let us assume that the function

f n .x/ D .x

2

1/ mod n

behaves like a “random” function. Of course, it is not really random, but this as-

sumption yields results consistent with the observed behavior of POLLARD-RHO.

We can then consider each x i to have been independently drawn from Z n according

to a uniform distribution on Z n . By the birthday-paradox analysis of Section 5.4.1,

we expect ‚.

p

n/ steps to be taken before the sequence cycles.

Now for the required modiﬁcation. Let p be a nontrivial factor of n such that

gcd.p; n=p/ D 1. For example, if n has the factorization n D p

e 1

1 p

e 2

2 p e r

r

, then

we may take p to be p

e 1

1 . (If e 1 D 1, then p is just the smallest prime factor of n,

a good example to keep in mind.)

978 Chapter 31 Number-Theoretic Algorithms

996 310

396

84

120

529

1053 595

339

814

1194

63

8

3

2

(b) (c) (a)

3

2

18

26

8

31

11

47

177

1186

mod 1387 mod 19 mod 73

8

6

16

63

3

2 x 1

x 2

x 3

x 4

x 5

x 6

x 7

x 0

1

x 0

2

x 0

3

x 0

4

x 0

5

x 0

6

x 0

7

x 00

1

x 00

2

x 00

3

x 00

4

x 00

5

x 00

6

x 00

7

Figure 31.7 Pollard’s rho heuristic. (a) The values produced by the recurrence xiC1 D

.x

2

i

1/ mod 1387, starting with x1 D 2. The prime factorization of 1387 is 19 73. The heavy

arrows indicate the iteration steps that are executed before the factor 19 is discovered. The light

arrows point to unreached values in the iteration, to illustrate the “rho” shape. The shaded values are

the y values stored by POLLARD-RHO. The factor 19 is discovered upon reaching x7 D 177, when

gcd.63 177; 1387/ D 19 is computed. The ﬁrst x value that would be repeated is 1186, but the

factor 19 is discovered before this value is repeated. (b) The values produced by the same recurrence,

modulo 19. Every value xi given in part (a) is equivalent, modulo 19, to the value x

0

i

shown here.

For example, both x4 D 63 and x7 D 177 are equivalent to 6, modulo 19. (c) The values produced

by the same recurrence, modulo 73. Every value xi given in part (a) is equivalent, modulo 73, to the

value x

00

i

shown here. By the Chinese remainder theorem, each node in part (a) corresponds to a pair

of nodes, one from part (b) and one from part (c).

The sequence hx i i induces a corresponding sequence hx 0

i

i modulo p, where

x

0

i

D x i mod p

for all i.

Furthermore, because f n is deﬁned using only arithmetic operations (squaring

and subtraction) modulo n, we can compute x 0

iC1

from x 0

i

; the “modulo p” view of

31.9 Integer factorization 979

the sequence is a smaller version of what is happening modulo n:

x

0

iC1

D x iC1 mod p

D f n .x i / mod p

D ..x

2

i

1/ mod n/ mod p

D .x

2

i

1/ mod p (by Exercise 31.1-7)

D ..x i mod p/

2

1/ mod p

D ..x

0

i

/

2

1/ mod p

D f p .x

0

i

/ :

Thus, although we are not explicitly computing the sequence hx 0

i

i, this sequence is

well deﬁned and obeys the same recurrence as the sequence hx i i.

Reasoning as before, we ﬁnd that the expected number of steps before the se-

quence hx 0

i

i repeats is ‚.

p

p/. If p is small compared to n, the sequence hx 0

i

i might

repeat much more quickly than the sequence hx i i. Indeed, as parts (b) and (c) of

Figure 31.7 show, the hx 0

i

i sequence repeats as soon as two elements of the se-

quence hx i i are merely equivalent modulo p, rather than equivalent modulo n.

Let t denote the index of the ﬁrst repeated value in the hx 0

i

i sequence, and let

u > 0 denote the length of the cycle that has been thereby produced. That is, t

and u > 0 are the smallest values such that x 0

tCi

D x 0

tCuCi

for all i 0. By the

above arguments, the expected values of t and u are both ‚.

p

p/. Note that if

x 0

tCi

D x 0

tCuCi

, then p j .x tCuCi x tCi /. Thus, gcd.x tCuCi x tCi ; n/ > 1.

Therefore, once POLLARD-RHO has saved as y any value x k such that k t,

then y mod p is always on the cycle modulo p. (If a new value is saved as y,

that value is also on the cycle modulo p.) Eventually, k is set to a value that

is greater than u, and the procedure then makes an entire loop around the cycle

modulo p without changing the value of y. The procedure then discovers a factor

of n when x i “runs into” the previously stored value of y, modulo p, that is, when

x i y .mod p/.

Presumably, the factor found is the factor p, although it may occasionally hap-

pen that a multiple of p is discovered. Since the expected values of both t and u are

‚.

p

p/, the expected number of steps required to produce the factor p is ‚.

p

p/.

This algorithm might not perform quite as expected, for two reasons. First, the

heuristic analysis of the running time is not rigorous, and it is possible that the cycle

of values, modulo p, could be much larger than

p

p. In this case, the algorithm

performs correctly but much more slowly than desired. In practice, this issue seems

to be moot. Second, the divisors of n produced by this algorithm might always be

one of the trivial factors 1 or n. For example, suppose that n D pq, where p

and q are prime. It can happen that the values of t and u for p are identical with

the values of t and u for q, and thus the factor p is always revealed in the same

gcd operation that reveals the factor q. Since both factors are revealed at the same

980 Chapter 31 Number-Theoretic Algorithms

time, the trivial factor pq D n is revealed, which is useless. Again, this problem

seems to be insigniﬁcant in practice. If necessary, we can restart the heuristic with

a different recurrence of the form x iC1 D .x 2

i

c/ mod n. (We should avoid the

values c D 0 and c D 2 for reasons we will not go into here, but other values are

ﬁne.)

Of course, this analysis is heuristic and not rigorous, since the recurrence is

not really “random.” Nonetheless, the procedure performs well in practice, and

it seems to be as efﬁcient as this heuristic analysis indicates. It is the method of

choice for ﬁnding small prime factors of a large number. To factor a ˇ-bit compos-

ite number n completely, we only need to ﬁnd all prime factors less than bn 1=2 c,

and so we expect POLLARD-RHO to require at most n 1=4 D 2 ˇ=4

arithmetic opera-

tions and at most n 1=4 ˇ 2 D 2 ˇ=4 ˇ 2

bit operations. POLLARD-RHO’s ability to ﬁnd

a small factor p of n with an expected number ‚.

p

p/ of arithmetic operations is

often its most appealing feature.

Exercises

31.9-1

Referring to the execution history shown in Figure 31.7(a), when does POLLARD-

RHO print the factor 73 of 1387?

31.9-2

Suppose that we are given a function f W Z n ! Z n and an initial value x 0 2 Z n .

Deﬁne x i D f .x i1 / for i D 1; 2; : : :. Let t and u > 0 be the smallest values such

that x tCi D x tCuCi for i D 0; 1; : : :. In the terminology of Pollard’s rho algorithm,

t is the length of the tail and u is the length of the cycle of the rho. Give an efﬁcient

algorithm to determine t and u exactly, and analyze its running time.

31.9-3

How many steps would you expect POLLARD-RHO to require to discover a factor

of the form p e

, where p is prime and e > 1?

31.9-4 ?

One disadvantage of POLLARD-RHO as written is that it requires one gcd compu-

tation for each step of the recurrence. Instead, we could batch the gcd computa-

tions by accumulating the product of several x i values in a row and then using this

product instead of x i in the gcd computation. Describe carefully how you would

implement this idea, why it works, and what batch size you would pick as the most

effective when working on a ˇ-bit number n.

Problems for Chapter 31 981

Problems

31-1 Binary gcd algorithm

Most computers can perform the operations of subtraction, testing the parity (odd

or even) of a binary integer, and halving more quickly than computing remainders.

This problem investigates the binary gcd algorithm, which avoids the remainder

computations used in Euclid’s algorithm.

a. Prove that if a and b are both even, then gcd.a; b/ D 2 gcd.a=2; b=2/.

b. Prove that if a is odd and b is even, then gcd.a; b/ D gcd.a; b=2/.

c. Prove that if a and b are both odd, then gcd.a; b/ D gcd..a b/=2; b/.

d. Design an efﬁcient binary gcd algorithm for input integers a and b, where

a b, that runs in O.lg a/ time. Assume that each subtraction, parity test,

and halving takes unit time.

31-2 Analysis of bit operations in Euclid’s algorithm

a. Consider the ordinary “paper and pencil” algorithm for long division: dividing

a by b, which yields a quotient q and remainder r. Show that this method

requires O..1 C lg q/ lg b/ bit operations.

b. Deﬁne .a; b/ D .1 C lg a/.1 C lg b/. Show that the number of bit operations

performed by EUCLID in reducing the problem of computing gcd.a; b/ to that

of computing gcd.b; a mod b/ is at most c..a; b/ .b; a mod b// for some

sufﬁciently large constant c > 0.

c. Show that EUCLID.a; b/ requires O..a; b// bit operations in general and

O.ˇ 2 / bit operations when applied to two ˇ-bit inputs.

31-3 Three algorithms for Fibonacci numbers

This problem compares the efﬁciency of three methods for computing the nth Fi-

bonacci number F n , given n. Assume that the cost of adding, subtracting, or mul-

tiplying two numbers is O.1/, independent of the size of the numbers.

a. Show that the running time of the straightforward recursive method for com-

puting F n based on recurrence (3.22) is exponential in n. (See, for example, the

FIB procedure on page 775.)

b. Show how to compute F n in O.n/ time using memoization.

982 Chapter 31 Number-Theoretic Algorithms

c. Show how to compute F n in O.lg n/ time using only integer addition and mul-

tiplication. (Hint: Consider the matrix

0 1

1 1

and its powers.)

d. Assume now that adding two ˇ-bit numbers takes ‚.ˇ/ time and that multi-

plying two ˇ-bit numbers takes ‚.ˇ 2 / time. What is the running time of these

three methods under this more reasonable cost measure for the elementary arith-

metic operations?

31-4 Quadratic residues

Let p be an odd prime. A number a 2 Z

p

is a quadratic residue if the equation

x 2 D a .mod p/ has a solution for the unknown x.

a. Show that there are exactly .p 1/=2 quadratic residues, modulo p.

b. If p is prime, we deﬁne the Legendre symbol .

a

p

/, for a 2 Z

p

, to be 1 if a is a

quadratic residue modulo p and 1 otherwise. Prove that if a 2 Z

p

, then

a

p

a

.p1/=2

.mod p/ :

Give an efﬁcient algorithm that determines whether a given number a is a qua-

dratic residue modulo p. Analyze the efﬁciency of your algorithm.

c. Prove that if p is a prime of the form 4k C 3 and a is a quadratic residue in Z

p

,

then a kC1

mod p is a square root of a, modulo p. How much time is required

to ﬁnd the square root of a quadratic residue a modulo p?

d. Describe an efﬁcient randomized algorithm for ﬁnding a nonquadratic residue,

modulo an arbitrary prime p, that is, a member of Z

p

that is not a quadratic

residue. How many arithmetic operations does your algorithm require on aver-

age?

Chapter notes

Niven and Zuckerman [265] provide an excellent introduction to elementary num-

ber theory. Knuth [210] contains a good discussion of algorithms for ﬁnding the

Notes for Chapter 31 983

greatest common divisor, as well as other basic number-theoretic algorithms. Bach

[30] and Riesel [295] provide more recent surveys of computational number the-

ory. Dixon [91] gives an overview of factorization and primality testing. The

conference proceedings edited by Pomerance [280] contains several excellent sur-

vey articles. More recently, Bach and Shallit [31] have provided an exceptional

overview of the basics of computational number theory.

Knuth [210] discusses the origin of Euclid’s algorithm. It appears in Book 7,

Propositions 1 and 2, of the Greek mathematician Euclid’s Elements, which was

written around 300 B.C. Euclid’s description may have been derived from an al-

gorithm due to Eudoxus around 375 B.C. Euclid’s algorithm may hold the honor

of being the oldest nontrivial algorithm; it is rivaled only by an algorithm for mul-

tiplication known to the ancient Egyptians. Shallit [312] chronicles the history of

the analysis of Euclid’s algorithm.

Knuth attributes a special case of the Chinese remainder theorem (Theo-

rem 31.27) to the Chinese mathematician Sun-Ts˘u, who lived sometime between

200 B.C. and A.D. 200—the date is quite uncertain. The same special case was

given by the Greek mathematician Nichomachus around A.D. 100. It was gener-

alized by Chhin Chiu-Shao in 1247. The Chinese remainder theorem was ﬁnally

stated and proved in its full generality by L. Euler in 1734.

The randomized primality-testing algorithm presented here is due to Miller [255]

and Rabin [289]; it is the fastest randomized primality-testing algorithm known,

to within constant factors. The proof of Theorem 31.39 is a slight adaptation of

one suggested by Bach [29]. A proof of a stronger result for MILLER-RABIN

was given by Monier [258, 259]. For many years primality-testing was the classic

example of a problem where randomization appeared to be necessary to obtain

an efﬁcient (polynomial-time) algorithm. In 2002, however, Agrawal, Kayal, and

Saxema [4] surprised everyone with their deterministic polynomial-time primality-

testing algorithm. Until then, the fastest deterministic primality testing algorithm

known, due to Cohen and Lenstra [73], ran in time .lg n/ O.lg lg lg n/

on input n, which

is just slightly superpolynomial. Nonetheless, for practical purposes randomized

primality-testing algorithms remain more efﬁcient and are preferred.

The problem of ﬁnding large “random” primes is nicely discussed in an article

by Beauchemin, Brassard, Cr´epeau, Goutier, and Pomerance [36].

The concept of a public-key cryptosystem is due to Difﬁe and Hellman [87].

The RSA cryptosystem was proposed in 1977 by Rivest, Shamir, and Adleman

[296]. Since then, the ﬁeld of cryptography has blossomed. Our understanding

of the RSA cryptosystem has deepened, and modern implementations use signif-

icant reﬁnements of the basic techniques presented here. In addition, many new

techniques have been developed for proving cryptosystems to be secure. For ex-

ample, Goldwasser and Micali [142] show that randomization can be an effective

tool in the design of secure public-key encryption schemes. For signature schemes,

984 Chapter 31 Number-Theoretic Algorithms

Goldwasser, Micali, and Rivest [143] present a digital-signature scheme for which

every conceivable type of forgery is provably as difﬁcult as factoring. Menezes,

van Oorschot, and Vanstone [254] provide an overview of applied cryptography.

The rho heuristic for integer factorization was invented by Pollard [277]. The

version presented here is a variant proposed by Brent [56].

The best algorithms for factoring large numbers have a running time that grows

roughly exponentially with the cube root of the length of the number n to be fac-

tored. The general number-ﬁeld sieve factoring algorithm (as developed by Buh-

ler, Lenstra, and Pomerance [57] as an extension of the ideas in the number-ﬁeld

sieve factoring algorithm by Pollard [278] and Lenstra et al. [232] and reﬁned by

Coppersmith [77] and others) is perhaps the most efﬁcient such algorithm in gen-

eral for large inputs. Although it is difﬁcult to give a rigorous analysis of this

algorithm, under reasonable assumptions we can derive a running-time estimate of

L.1=3; n/ 1:902Co.1/

, where L.˛; n/ D e .ln n/

˛

.ln ln n/

1˛

.

The elliptic-curve method due to Lenstra [233] may be more effective for some

inputs than the number-ﬁeld sieve method, since, like Pollard’s rho method, it can

ﬁnd a small prime factor p quite quickly. With this method, the time to ﬁnd p is

estimated to be L.1=2; p/

p

2Co.1/

.

32 String Matching

Text-editing programs frequently need to ﬁnd all occurrences of a pattern in the

text. Typically, the text is a document being edited, and the pattern searched for is a

particular word supplied by the user. Efﬁcient algorithms for this problem—called

“string matching”—can greatly aid the responsiveness of the text-editing program.

Among their many other applications, string-matching algorithms search for par-

ticular patterns in DNA sequences. Internet search engines also use them to ﬁnd

Web pages relevant to queries.

We formalize the string-matching problem as follows. We assume that the

text is an array T Œ1 : : n of length n and that the pattern is an array P Œ1 : : m

of length m n. We further assume that the elements of P and T are char-

acters drawn from a ﬁnite alphabet †. For example, we may have † D f0,1g

or † D fa; b; : : : ; zg. The character arrays P and T are often called strings of

characters.

Referring to Figure 32.1, we say that pattern P occurs with shift s in text T

(or, equivalently, that pattern P occurs beginning at position s C 1 in text T ) if

0 s n m and T Œs C 1 : : s C m D P Œ1 : : m (that is, if T Œs C j D P Œj , for

1 j m). If P occurs with shift s in T , then we call s a valid shift; otherwise,

we call s an invalid shift. The string-matching problem is the problem of ﬁnding

all valid shifts with which a given pattern P occurs in a given text T .

a b c a b a a b c a b a c

a b a a pattern P

text T

s = 3

Figure 32.1 An example of the string-matching problem, where we want to ﬁnd all occurrences of

the pattern P D abaa in the text T D abcabaabcabac. The pattern occurs only once in the text,

at shift s D 3, which we call a valid shift. A vertical line connects each character of the pattern to its

matching character in the text, and all matched characters are shaded.

986 Chapter 32 String Matching

Algorithm Preprocessing time Matching time

Naive 0 O..n m C 1/m/

Rabin-Karp ‚.m/ O..n m C 1/m/

Finite automaton O.m j†j/ ‚.n/

Knuth-Morris-Pratt ‚.m/ ‚.n/

Figure 32.2 The string-matching algorithms in this chapter and their preprocessing and matching

times.

Except for the naive brute-force algorithm, which we review in Section 32.1,

each string-matching algorithm in this chapter performs some preprocessing based

on the pattern and then ﬁnds all valid shifts; we call this latter phase “matching.”

Figure 32.2 shows the preprocessing and matching times for each of the algorithms

in this chapter. The total running time of each algorithm is the sum of the prepro-

cessing and matching times. Section 32.2 presents an interesting string-matching

algorithm, due to Rabin and Karp. Although the ‚..n m C 1/m/ worst-case

running time of this algorithm is no better than that of the naive method, it works

much better on average and in practice. It also generalizes nicely to other pattern-

matching problems. Section 32.3 then describes a string-matching algorithm that

begins by constructing a ﬁnite automaton speciﬁcally designed to search for occur-

rences of the given pattern P in a text. This algorithm takes O.m j†j/ preprocess-

ing time, but only ‚.n/ matching time. Section 32.4 presents the similar, but much

cleverer, Knuth-Morris-Pratt (or KMP) algorithm; it has the same ‚.n/ matching

time, and it reduces the preprocessing time to only ‚.m/.

Notation and terminology

We denote by †

(read “sigma-star”) the set of all ﬁnite-length strings formed

using characters from the alphabet †. In this chapter, we consider only ﬁnite-

length strings. The zero-length empty string, denoted ", also belongs to †

. The

length of a string x is denoted jxj. The concatenation of two strings x and y,

denoted xy, has length jxj C jyj and consists of the characters from x followed by

the characters from y.

We say that a string w is a preﬁx of a string x, denoted w x, if x D wy for

some string y 2 †

. Note that if w x, then jwj jxj. Similarly, we say that a

string w is a sufﬁx of a string x, denoted w x, if x D yw for some y 2 †

. As

with a preﬁx, w x implies jwj jxj. For example, we have ab abcca and

cca abcca. The empty string " is both a sufﬁx and a preﬁx of every string. For

any strings x and y and any character a, we have x y if and only if xa ya.

Chapter 32 String Matching 987

x

z

x

y

y

(a)

x

z

x

y

y

(b)

x

z

x

y

y

(c)

Figure 32.3 A graphical proof of Lemma 32.1. We suppose that x ´ and y ´. The three parts

of the ﬁgure illustrate the three cases of the lemma. Vertical lines connect matching regions (shown

shaded) of the strings. (a) If jxj jyj, then x y. (b) If jxj jyj, then y x. (c) If jxj D jyj,

then x D y.

Also note that and are transitive relations. The following lemma will be useful

later.

Lemma 32.1 (Overlapping-sufﬁx lemma)

Suppose that x, y, and ´ are strings such that x ´ and y ´. If jxj jyj,

then x y. If jxj jyj, then y x. If jxj D jyj, then x D y.

Proof See Figure 32.3 for a graphical proof.

For brevity of notation, we denote the k-character preﬁx P Œ1 : : k of the pattern

P Œ1 : : m by P k . Thus, P 0 D " and P m D P D P Œ1 : : m. Similarly, we denote

the k-character preﬁx of the text T by T k . Using this notation, we can state the

string-matching problem as that of ﬁnding all shifts s in the range 0 s n m

such that P T sCm .

In our pseudocode, we allow two equal-length strings to be compared for equal-

ity as a primitive operation. If the strings are compared from left to right and the

comparison stops when a mismatch is discovered, we assume that the time taken

by such a test is a linear function of the number of matching characters discovered.

To be precise, the test “x == y” is assumed to take time ‚.t C 1/, where t is the

length of the longest string ´ such that ´ x and ´ y. (We write ‚.t C 1/

rather than ‚.t/ to handle the case in which t D 0; the ﬁrst characters compared

do not match, but it takes a positive amount of time to perform this comparison.)

988 Chapter 32 String Matching

32.1 The naive string-matching algorithm

The naive algorithm ﬁnds all valid shifts using a loop that checks the condition

P Œ1 : : m D T Œs C 1 : : s C m for each of the n m C 1 possible values of s.

NAIVE-STRING-MATCHER.T; P /

1 n D T:length

2 m D P:length

3 for s D 0 to n m

4 if P Œ1 : : m == T Œs C 1 : : s C m

5 print “Pattern occurs with shift” s

Figure 32.4 portrays the naive string-matching procedure as sliding a “template”

containing the pattern over the text, noting for which shifts all of the characters

on the template equal the corresponding characters in the text. The for loop of

lines 3–5 considers each possible shift explicitly. The test in line 4 determines

whether the current shift is valid; this test implicitly loops to check corresponding

character positions until all positions match successfully or a mismatch is found.

Line 5 prints out each valid shift s.

Procedure NAIVE-STRING-MATCHER takes time O..n m C 1/m/, and this

bound is tight in the worst case. For example, consider the text string a

n

(a string

of n a’s) and the pattern a

m

. For each of the nmC1 possible values of the shift s,

the implicit loop on line 4 to compare corresponding characters must execute m

times to validate the shift. The worst-case running time is thus ‚..n m C 1/m/,

which is ‚.n 2 / if m D bn=2c. Because it requires no preprocessing, NAIVE-

STRING-MATCHER’s running time equals its matching time.

a c a a b c

a a b

s = 0

(a)

a c a a b c

a a b

s = 1

(b)

a c a a b c

a a b

s = 2

(c)

a c a a b c

a a b

s = 3

(d)

Figure 32.4 The operation of the naive string matcher for the pattern P D aab and the text

T D acaabc. We can imagine the pattern P as a template that we slide next to the text. (a)–(d) The

four successive alignments tried by the naive string matcher. In each part, vertical lines connect cor-

responding regions found to match (shown shaded), and a jagged line connects the ﬁrst mismatched

character found, if any. The algorithm ﬁnds one occurrence of the pattern, at shift s D 2, shown in

part (c).

32.1 The naive string-matching algorithm 989

As we shall see, NAIVE-STRING-MATCHER is not an optimal procedure for this

problem. Indeed, in this chapter we shall see that the Knuth-Morris-Pratt algorithm

is much better in the worst case. The naive string-matcher is inefﬁcient because

it entirely ignores information gained about the text for one value of s when it

considers other values of s. Such information can be quite valuable, however. For

example, if P D aaab and we ﬁnd that s D 0 is valid, then none of the shifts 1, 2,

or 3 are valid, since T Œ4 D b. In the following sections, we examine several ways

to make effective use of this sort of information.

Exercises

32.1-1

Show the comparisons the naive string matcher makes for the pattern P D 0001

in the text T D 000010001010001.

32.1-2

Suppose that all characters in the pattern P are different. Show how to accelerate

NAIVE-STRING-MATCHER to run in time O.n/ on an n-character text T .

32.1-3

Suppose that pattern P and text T are randomly chosen strings of length m and n,

respectively, from the d-ary alphabet † d D f0; 1; : : : ; d 1g, where d 2. Show

that the expected number of character-to-character comparisons made by the im-

plicit loop in line 4 of the naive algorithm is

.n m C 1/

1 d m

1 d 1

2.n m C 1/

over all executions of this loop. (Assume that the naive algorithm stops comparing

characters for a given shift once it ﬁnds a mismatch or matches the entire pattern.)

Thus, for randomly chosen strings, the naive algorithm is quite efﬁcient.

32.1-4

Suppose we allow the pattern P to contain occurrences of a gap character } that

can match an arbitrary string of characters (even one of zero length). For example,

the pattern ab}ba}c occurs in the text cabccbacbacab as

c ab ’

ab

cc ’

}

ba ’

ba

cba “

}

c ’

c

ab

and as

c ab ’

ab

ccbac —

}

ba ’

ba

’

}

c ’

c

ab :

990 Chapter 32 String Matching

Note that the gap character may occur an arbitrary number of times in the pattern

but not at all in the text. Give a polynomial-time algorithm to determine whether

such a pattern P occurs in a given text T , and analyze the running time of your

algorithm.

32.2 The Rabin-Karp algorithm

Rabin and Karp proposed a string-matching algorithm that performs well in prac-

tice and that also generalizes to other algorithms for related problems, such as

two-dimensional pattern matching. The Rabin-Karp algorithm uses ‚.m/ prepro-

cessing time, and its worst-case running time is ‚..nmC1/m/. Based on certain

assumptions, however, its average-case running time is better.

This algorithm makes use of elementary number-theoretic notions such as the

equivalence of two numbers modulo a third number. You might want to refer to

Section 31.1 for the relevant deﬁnitions.

For expository purposes, let us assume that † D f0; 1; 2; : : : ; 9g, so that each

character is a decimal digit. (In the general case, we can assume that each charac-

ter is a digit in radix-d notation, where d D j†j.) We can then view a string of k

consecutive characters as representing a length-k decimal number. The character

string 31415 thus corresponds to the decimal number 31,415. Because we inter-

pret the input characters as both graphical symbols and digits, we ﬁnd it convenient

in this section to denote them as we would digits, in our standard text font.

Given a pattern P Œ1 : : m, let p denote its corresponding decimal value. In a sim-

ilar manner, given a text T Œ1 : : n, let t s denote the decimal value of the length-m

substring T Œs C 1 : : s C m, for s D 0; 1; : : : ; n m. Certainly, t s D p if and only

if T Œs C 1 : : s C m D P Œ1 : : m; thus, s is a valid shift if and only if t s D p. If we

could compute p in time ‚.m/ and all the t s values in a total of ‚.nmC1/ time, 1

then we could determine all valid shifts s in time ‚.m/ C ‚.n m C 1/ D ‚.n/

by comparing p with each of the t s values. (For the moment, let’s not worry about

the possibility that p and the t s values might be very large numbers.)

We can compute p in time ‚.m/ using Horner’s rule (see Section 30.1):

p D P Œm C 10 .P Œm 1 C 10.P Œm 2 C C 10.P Œ2 C 10P Œ1/ // :

Similarly, we can compute t 0 from T Œ1 : : m in time ‚.m/.

1

We write ‚.n m C 1/ instead of ‚.n m/ because s takes on n m C 1 different values. The

“C1” is signiﬁcant in an asymptotic sense because when m D n, computing the lone ts value takes

‚.1/ time, not ‚.0/ time.

32.2 The Rabin-Karp algorithm 991

To compute the remaining values t 1 ; t 2 ; : : : ; t nm in time ‚.n m/, we observe

that we can compute t sC1 from t s in constant time, since

t sC1 D 10.t s 10

m1

T Œs C 1/ C T Œs C m C 1 : (32.1)

Subtracting 10 m1 T Œs C 1 removes the high-order digit from t s , multiplying the

result by 10 shifts the number left by one digit position, and adding T Œs C m C 1

brings in the appropriate low-order digit. For example, if m D 5 and t s D 31415,

then we wish to remove the high-order digit T Œs C 1 D 3 and bring in the new

low-order digit (suppose it is T Œs C 5 C 1 D 2) to obtain

t sC1 D 10.31415 10000 3/ C 2

D 14152 :

If we precompute the constant 10 m1

(which we can do in time O.lg m/ using the

techniques of Section 31.6, although for this application a straightforward O.m/-

time method sufﬁces), then each execution of equation (32.1) takes a constant num-

ber of arithmetic operations. Thus, we can compute p in time ‚.m/, and we can

compute all of t 0 ; t 1 ; : : : ; t nm in time ‚.n m C 1/. Therefore, we can ﬁnd all

occurrences of the pattern P Œ1 : : m in the text T Œ1 : : n with ‚.m/ preprocessing

time and ‚.n m C 1/ matching time.

Until now, we have intentionally overlooked one problem: p and t s may be

too large to work with conveniently. If P contains m characters, then we cannot

reasonably assume that each arithmetic operation on p (which is m digits long)

takes “constant time.” Fortunately, we can solve this problem easily, as Figure 32.5

shows: compute p and the t s values modulo a suitable modulus q. We can compute

p modulo q in ‚.m/ time and all the t s values modulo q in ‚.n m C 1/ time.

If we choose the modulus q as a prime such that 10q just ﬁts within one computer

word, then we can perform all the necessary computations with single-precision

arithmetic. In general, with a d-ary alphabet f0; 1; : : : ; d 1g, we choose q so

that dq ﬁts within a computer word and adjust the recurrence equation (32.1) to

work modulo q, so that it becomes

t sC1 D .d.t s T Œs C 1h/ C T Œs C m C 1/ mod q ; (32.2)

where h d m1 .mod q/ is the value of the digit “1” in the high-order position

of an m-digit text window.

The solution of working modulo q is not perfect, however: t s p .mod q/

does not imply that t s D p. On the other hand, if t s 6 p .mod q/, then we

deﬁnitely have that t s ¤ p, so that shift s is invalid. We can thus use the test

t s p .mod q/ as a fast heuristic test to rule out invalid shifts s. Any shift s for

which t s p .mod q/ must be tested further to see whether s is really valid or

we just have a spurious hit. This additional test explicitly checks the condition

992 Chapter 32 String Matching

2 3 5 9 0 2 3 1 4 1 5 2 6 7 3 9 9 2 1

7

(a)

mod 13

2 3 5 9 0 2 3 1 4 1 5 2 6 7 3 9 9 2 1

7

(b)

mod 13

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

8 9 3 11 0 1 8 5 11 9 11 7 10 4

valid

match

spurious

hit

… … …

3 1 4 1 5 2

7 8

old

high-order

digit

new

low-order

digit

≡ (31415 – 3·10000)·10 + 2 (mod 13)

old

high-order

digit

new

low-order

digit shift

≡ (7 – 3·3)·10 + 2 (mod 13)

≡ 8 (mod 13)

(c)

14152

Figure 32.5 The Rabin-Karp algorithm. Each character is a decimal digit, and we compute values

modulo 13. (a) A text string. A window of length 5 is shown shaded. The numerical value of the

shaded number, computed modulo 13, yields the value 7. (b) The same text string with values com-

puted modulo 13 for each possible position of a length-5 window. Assuming the pattern P D 31415,

we look for windows whose value modulo 13 is 7, since 31415 7 .mod 13/. The algorithm ﬁnds

two such windows, shown shaded in the ﬁgure. The ﬁrst, beginning at text position 7, is indeed an

occurrence of the pattern, while the second, beginning at text position 13, is a spurious hit. (c) How

to compute the value for a window in constant time, given the value for the previous window. The

ﬁrst window has value 31415. Dropping the high-order digit 3, shifting left (multiplying by 10), and

then adding in the low-order digit 2 gives us the new value 14152. Because all computations are

performed modulo 13, the value for the ﬁrst window is 7, and the value for the new window is 8.

32.2 The Rabin-Karp algorithm 993

P Œ1 : : m D T Œs C 1 : : s C m. If q is large enough, then we hope that spurious

hits occur infrequently enough that the cost of the extra checking is low.

The following procedure makes these ideas precise. The inputs to the procedure

are the text T , the pattern P , the radix d to use (which is typically taken to be j†j),

and the prime q to use.

RABIN-KARP-MATCHER.T; P; d; q/

1 n D T:length

2 m D P:length

3 h D d m1

mod q

4 p D 0

5 t 0 D 0

6 for i D 1 to m // preprocessing

7 p D .dp C P Œi/ mod q

8 t 0 D .dt 0 C T Œi/ mod q

9 for s D 0 to n m // matching

10 if p == t s

11 if P Œ1 : : m == T Œs C 1 : : s C m

12 print “Pattern occurs with shift” s

13 if s < n m

14 t sC1 D .d.t s T Œs C 1h/ C T Œs C m C 1/ mod q

The procedure RABIN-KARP-MATCHER works as follows. All characters are

interpreted as radix-d digits. The subscripts on t are provided only for clarity; the

program works correctly if all the subscripts are dropped. Line 3 initializes h to the

value of the high-order digit position of an m-digit window. Lines 4–8 compute p

as the value of P Œ1 : : m mod q and t 0 as the value of T Œ1 : : m mod q. The for

loop of lines 9–14 iterates through all possible shifts s, maintaining the following

invariant:

Whenever line 10 is executed, t s D T Œs C 1 : : s C m mod q.

If p D t s in line 10 (a “hit”), then line 11 checks to see whether P Œ1 : : m D

T Œs C1 : : s Cm in order to rule out the possibility of a spurious hit. Line 12 prints

out any valid shifts that are found. If s < n m (checked in line 13), then the for

loop will execute at least one more time, and so line 14 ﬁrst executes to ensure that

the loop invariant holds when we get back to line 10. Line 14 computes the value

of t sC1 mod q from the value of t s mod q in constant time using equation (32.2)

directly.

RABIN-KARP-MATCHER takes ‚.m/ preprocessing time, and its matching time

is ‚..n m C 1/m/ in the worst case, since (like the naive string-matching algo-

rithm) the Rabin-Karp algorithm explicitly veriﬁes every valid shift. If P D a

m

994 Chapter 32 String Matching

and T D a

n

, then verifying takes time ‚..nmC1/m/, since each of the nmC1

possible shifts is valid.

In many applications, we expect few valid shifts—perhaps some constant c of

them. In such applications, the expected matching time of the algorithm is only

O..n m C 1/ C cm/ D O.n C m/, plus the time required to process spurious

hits. We can base a heuristic analysis on the assumption that reducing values mod-

ulo q acts like a random mapping from †

to Z q . (See the discussion on the use of

division for hashing in Section 11.3.1. It is difﬁcult to formalize and prove such an

assumption, although one viable approach is to assume that q is chosen randomly

from integers of the appropriate size. We shall not pursue this formalization here.)

We can then expect that the number of spurious hits is O.n=q/, since we can es-

timate the chance that an arbitrary t s will be equivalent to p, modulo q, as 1=q.

Since there are O.n/ positions at which the test of line 10 fails and we spend O.m/

time for each hit, the expected matching time taken by the Rabin-Karp algorithm

is

O.n/ C O.m. C n=q// ;

where is the number of valid shifts. This running time is O.n/ if D O.1/ and

we choose q m. That is, if the expected number of valid shifts is small (O.1/)

and we choose the prime q to be larger than the length of the pattern, then we

can expect the Rabin-Karp procedure to use only O.n C m/ matching time. Since

m n, this expected matching time is O.n/.

Exercises

32.2-1

Working modulo q D 11, how many spurious hits does the Rabin-Karp matcher en-

counter in the text T D 3141592653589793 when looking for the pattern P D 26?

32.2-2

How would you extend the Rabin-Karp method to the problem of searching a text

string for an occurrence of any one of a given set of k patterns? Start by assuming

that all k patterns have the same length. Then generalize your solution to allow the

patterns to have different lengths.

32.2-3

Show how to extend the Rabin-Karp method to handle the problem of looking for

a given m m pattern in an n n array of characters. (The pattern may be shifted

vertically and horizontally, but it may not be rotated.)

32.3 String matching with ﬁnite automata 995

32.2-4

Alice has a copy of a long n-bit ﬁle A D ha n1 ; a n2 ; : : : ; a 0 i, and Bob similarly

has an n-bit ﬁle B D hb n1 ; b n2 ; : : : ; b 0 i. Alice and Bob wish to know if their

ﬁles are identical. To avoid transmitting all of A or B, they use the following fast

probabilistic check. Together, they select a prime q > 1000n and randomly select

an integer x from f0; 1; : : : ; q 1g. Then, Alice evaluates

A.x/ D

n1 X

iD0

a i x

i

!

mod q

and Bob similarly evaluates B.x/. Prove that if A ¤ B, there is at most one

chance in 1000 that A.x/ D B.x/, whereas if the two ﬁles are the same, A.x/ is

necessarily the same as B.x/. (Hint: See Exercise 31.4-4.)

32.3 String matching with ﬁnite automata

Many string-matching algorithms build a ﬁnite automaton—a simple machine for

processing information—that scans the text string T for all occurrences of the pat-

tern P . This section presents a method for building such an automaton. These

string-matching automata are very efﬁcient: they examine each text character ex-

actly once, taking constant time per text character. The matching time used—after

preprocessing the pattern to build the automaton—is therefore ‚.n/. The time to

build the automaton, however, can be large if † is large. Section 32.4 describes a

clever way around this problem.

We begin this section with the deﬁnition of a ﬁnite automaton. We then examine

a special string-matching automaton and show how to use it to ﬁnd occurrences

of a pattern in a text. Finally, we shall show how to construct the string-matching

automaton for a given input pattern.

Finite automata

A ﬁnite automaton M , illustrated in Figure 32.6, is a 5-tuple .Q; q 0 ; A; †; ı/,

where

Q is a ﬁnite set of states,

q 0 2 Q is the start state,

A Q is a distinguished set of accepting states,

† is a ﬁnite input alphabet,

ı is a function from Q † into Q, called the transition function of M .

996 Chapter 32 String Matching

1 0

0 0

a b

input

state

0

1

(a)

a

a

b

b

(b)

0 1

Figure 32.6 A simple two-state ﬁnite automaton with state set Q D f0; 1g, start state q0 D 0,

and input alphabet † D fa; bg. (a) A tabular representation of the transition function ı. (b) An

equivalent state-transition diagram. State 1, shown blackend, is the only accepting state. Directed

edges represent transitions. For example, the edge from state 1 to state 0 labeled b indicates that

ı.1; b/ D 0. This automaton accepts those strings that end in an odd number of a’s. More precisely,

it accepts a string x if and only if x D y´, where y D " or y ends with a b, and ´ D a

k

, where k is

odd. For example, on input abaaa, including the start state, this automaton enters the sequence of

states h0; 1; 0; 1; 0; 1i, and so it accepts this input. For input abbaa, it enters the sequence of states

h0; 1; 0; 0; 1; 0i, and so it rejects this input.

The ﬁnite automaton begins in state q 0 and reads the characters of its input string

one at a time. If the automaton is in state q and reads input character a, it moves

(“makes a transition”) from state q to state ı.q; a/. Whenever its current state q is

a member of A, the machine M has accepted the string read so far. An input that

is not accepted is rejected.

A ﬁnite automaton M induces a function , called the ﬁnal-state function,

from †

to Q such that .w/ is the state M ends up in after scanning the string w.

Thus, M accepts a string w if and only if .w/ 2 A. We deﬁne the function

recursively, using the transition function:

."/ D q 0 ;

.wa/ D ı..w/; a/ for w 2 † ; a 2 † .

String-matching automata

For a given pattern P , we construct a string-matching automaton in a preprocess-

ing step before using it to search the text string. Figure 32.7 illustrates how we

construct the automaton for the pattern P D ababaca. From now on, we shall

assume that P is a given ﬁxed pattern string; for brevity, we shall not indicate the

dependence upon P in our notation.

In order to specify the string-matching automaton corresponding to a given pat-

tern P Œ1 : : m, we ﬁrst deﬁne an auxiliary function , called the sufﬁx function

corresponding to P . The function maps †

to f0; 1; : : : ; mg such that .x/ is the

length of the longest preﬁx of P that is also a sufﬁx of x:

.x/ D max fk W P k xg : (32.3)

32.3 String matching with ﬁnite automata 997

0 1 2 3 4 5 6 7

a b a b a c a

b

a

a

a

a

b

(a)

1 0 0

1 2 0

3 0 0

1 4 0

5 0 0

1 4 6

7 0 0

1 2 0

0

1

2

3

4

5

6

7

state

input

a b c

a

b

a

b

a

c

a

P

(b)

1 2 3 4 5 6 7 8 9 10 11

a b a b a b a c a b a

0 1 2 3 4 5 4 5 6 7 2 3

—

—

(c)

i

T Œi

state .T i /

Figure 32.7 (a) A state-transition diagram for the string-matching automaton that accepts all

strings ending in the string ababaca. State 0 is the start state, and state 7 (shown blackened) is

the only accepting state. A directed edge from state i to state j labeled a represents ı.i; a/ D j . The

right-going edges forming the “spine” of the automaton, shown heavy in the ﬁgure, correspond to

successful matches between pattern and input characters. The left-going edges correspond to failing

matches. Some edges corresponding to failing matches are omitted; by convention, if a state i has

no outgoing edge labeled a for some a 2 †, then ı.i; a/ D 0. (b) The corresponding transition

function ı, and the pattern string P D ababaca. The entries corresponding to successful matches

between pattern and input characters are shown shaded. (c) The operation of the automaton on the

text T D abababacaba. Under each text character T Œi appears the state .Ti / that the automa-

ton is in after processing the preﬁx Ti . The automaton ﬁnds one occurrence of the pattern, ending in

position 9.

The sufﬁx function is well deﬁned since the empty string P 0 D " is a suf-

ﬁx of every string. As examples, for the pattern P D ab, we have ."/ D 0,

.ccaca/ D 1, and .ccab/ D 2. For a pattern P of length m, we have

.x/ D m if and only if P x. From the deﬁnition of the sufﬁx function,

x y implies .x/ .y/.

We deﬁne the string-matching automaton that corresponds to a given pattern

P Œ1 : : m as follows:

998 Chapter 32 String Matching

The state set Q is f0; 1; : : : ; mg. The start state q 0 is state 0, and state m is the

only accepting state.

The transition function ı is deﬁned by the following equation, for any state q

and character a:

ı.q; a/ D .P q a/ : (32.4)

We deﬁne ı.q; a/ D .P q a/ because we want to keep track of the longest pre-

ﬁx of the pattern P that has matched the text string T so far. We consider the

most recently read characters of T . In order for a substring of T —let’s say the

substring ending at T Œi—to match some preﬁx P j of P , this preﬁx P j must be a

sufﬁx of T i . Suppose that q D .T i /, so that after reading T i , the automaton is in

state q. We design the transition function ı so that this state number, q, tells us the

length of the longest preﬁx of P that matches a sufﬁx of T i . That is, in state q,

P q T i and q D .T i /. (Whenever q D m, all m characters of P match a sufﬁx

of T i , and so we have found a match.) Thus, since .T i / and .T i / both equal q,

we shall see (in Theorem 32.4, below) that the automaton maintains the following

invariant:

.T i / D .T i / : (32.5)

If the automaton is in state q and reads the next character T Œi C 1 D a, then we

want the transition to lead to the state corresponding to the longest preﬁx of P that

is a sufﬁx of T i a, and that state is .T i a/. Because P q is the longest preﬁx of P

that is a sufﬁx of T i , the longest preﬁx of P that is a sufﬁx of T i a is not only .T i a/,

but also .P q a/. (Lemma 32.3, on page 1000, proves that .T i a/ D .P q a/.)

Thus, when the automaton is in state q, we want the transition function on charac-

ter a to take the automaton to state .P q a/.

There are two cases to consider. In the ﬁrst case, a D P Œq C 1, so that the

character a continues to match the pattern; in this case, because ı.q; a/ D qC1, the

transition continues to go along the “spine” of the automaton (the heavy edges in

Figure 32.7). In the second case, a ¤ P ŒqC1, so that a does not continue to match

the pattern. Here, we must ﬁnd a smaller preﬁx of P that is also a sufﬁx of T i .

Because the preprocessing step matches the pattern against itself when creating the

string-matching automaton, the transition function quickly identiﬁes the longest

such smaller preﬁx of P .

Let’s look at an example. The string-matching automaton of Figure 32.7 has

ı.5; c/ D 6, illustrating the ﬁrst case, in which the match continues. To illus-

trate the second case, observe that the automaton of Figure 32.7 has ı.5; b/ D 4.

We make this transition because if the automaton reads a b in state q D 5, then

P qb D ababab, and the longest preﬁx of P that is also a sufﬁx of ababab is

P 4 D abab.

32.3 String matching with ﬁnite automata 999

x

a

P

r

P

r–1

Figure 32.8 An illustration for the proof of Lemma 32.2. The ﬁgure shows that r .x/ C 1,

where r D .xa/.

To clarify the operation of a string-matching automaton, we now give a simple,

efﬁcient program for simulating the behavior of such an automaton (represented

by its transition function ı) in ﬁnding occurrences of a pattern P of length m in an

input text T Œ1 : : n. As for any string-matching automaton for a pattern of length m,

the state set Q is f0; 1; : : : ; mg, the start state is 0, and the only accepting state is

state m.

FINITE-AUTOMATON-MATCHER.T; ı; m/

1 n D T:length

2 q D 0

3 for i D 1 to n

4 q D ı.q; T Œi/

5 if q == m

6 print “Pattern occurs with shift” i m

From the simple loop structure of FINITE-AUTOMATON-MATCHER, we can easily

see that its matching time on a text string of length n is ‚.n/. This matching

time, however, does not include the preprocessing time required to compute the

transition function ı. We address this problem later, after ﬁrst proving that the

procedure FINITE-AUTOMATON-MATCHER operates correctly.

Consider how the automaton operates on an input text T Œ1 : : n. We shall prove

that the automaton is in state .T i / after scanning character T Œi. Since .T i / D m

if and only if P T i , the machine is in the accepting state m if and only if it has

just scanned the pattern P . To prove this result, we make use of the following two

lemmas about the sufﬁx function .

Lemma 32.2 (Sufﬁx-function inequality)

For any string x and character a, we have .xa/ .x/ C 1.

Proof Referring to Figure 32.8, let r D .xa/. If r D 0, then the conclusion

.xa/ D r .x/ C 1 is trivially satisﬁed, by the nonnegativity of .x/. Now

assume that r > 0. Then, P r xa, by the deﬁnition of . Thus, P r1 x, by

1000 Chapter 32 String Matching

x

a

a P

q

P

r

Figure 32.9 An illustration for the proof of Lemma 32.3. The ﬁgure shows that r D .Pqa/,

where q D .x/ and r D .xa/.

dropping the a from the end of P r and from the end of xa. Therefore, r1 .x/,

since .x/ is the largest k such that P k x, and thus .xa/ D r .x/ C 1.

Lemma 32.3 (Sufﬁx-function recursion lemma)

For any string x and character a, if q D .x/, then .xa/ D .P q a/.

Proof From the deﬁnition of , we have P q x. As Figure 32.9 shows, we

also have P q a xa. If we let r D .xa/, then P r xa and, by Lemma 32.2,

r q C 1. Thus, we have jP r j D r q C 1 D jP q aj. Since P q a xa, P r xa,

and jP r j jP q aj, Lemma 32.1 implies that P r P q a. Therefore, r .P q a/,

that is, .xa/ .P q a/. But we also have .P q a/ .xa/, since P q a xa.

Thus, .xa/ D .P q a/.

We are now ready to prove our main theorem characterizing the behavior of a

string-matching automaton on a given input text. As noted above, this theorem

shows that the automaton is merely keeping track, at each step, of the longest

preﬁx of the pattern that is a sufﬁx of what has been read so far. In other words,

the automaton maintains the invariant (32.5).

Theorem 32.4

If is the ﬁnal-state function of a string-matching automaton for a given pattern P

and T Œ1 : : n is an input text for the automaton, then

.T i / D .T i /

for i D 0; 1; : : : ; n.

Proof The proof is by induction on i. For i D 0, the theorem is trivially true,

since T 0 D ". Thus, .T 0 / D 0 D .T 0 /.

32.3 String matching with ﬁnite automata 1001

Now, we assume that .T i / D .T i / and prove that .T iC1 / D .T iC1 /. Let q

denote .T i /, and let a denote T Œi C 1. Then,

.T iC1 / D .T i a/ (by the deﬁnitions of T iC1 and a)

D ı..T i /; a/ (by the deﬁnition of )

D ı.q; a/ (by the deﬁnition of q)

D .P q a/ (by the deﬁnition (32.4) of ı)

D .T i a/ (by Lemma 32.3 and induction)

D .T iC1 / (by the deﬁnition of T iC1 ) .

By Theorem 32.4, if the machine enters state q on line 4, then q is the largest

value such that P q T i . Thus, we have q D m on line 5 if and only if the ma-

chine has just scanned an occurrence of the pattern P . We conclude that FINITE-

AUTOMATON-MATCHER operates correctly.

Computing the transition function

The following procedure computes the transition function ı from a given pattern

P Œ1 : : m.

COMPUTE-TRANSITION-FUNCTION.P; †/

1 m D P:length

2 for q D 0 to m

3 for each character a 2 †

4 k D min.m C 1; q C 2/

5 repeat

6 k D k 1

7 until P k P q a

8 ı.q; a/ D k

9 return ı

This procedure computes ı.q; a/ in a straightforward manner according to its def-

inition in equation (32.4). The nested loops beginning on lines 2 and 3 consider

all states q and all characters a, and lines 4–8 set ı.q; a/ to be the largest k such

that P k P q a. The code starts with the largest conceivable value of k, which is

min.m; q C 1/. It then decreases k until P k P q a, which must eventually occur,

since P 0 D " is a sufﬁx of every string.

The running time of COMPUTE-TRANSITION-FUNCTION is O.m 3 j†j/, be-

cause the outer loops contribute a factor of m j†j, the inner repeat loop can run

at most m C 1 times, and the test P k P q a on line 7 can require comparing up

1002 Chapter 32 String Matching

to m characters. Much faster procedures exist; by utilizing some cleverly com-

puted information about the pattern P (see Exercise 32.4-8), we can improve the

time required to compute ı from P to O.m j†j/. With this improved procedure for

computing ı, we can ﬁnd all occurrences of a length-m pattern in a length-n text

over an alphabet † with O.m j†j/ preprocessing time and ‚.n/ matching time.

Exercises

32.3-1

Construct the string-matching automaton for the pattern P D aabab and illustrate

its operation on the text string T D aaababaabaababaab.

32.3-2

Draw a state-transition diagram for a string-matching automaton for the pattern

ababbabbababbababbabb over the alphabet † D fa; bg.

32.3-3

We call a pattern P nonoverlappable if P k P q implies k D 0 or k D q. De-

scribe the state-transition diagram of the string-matching automaton for a nonover-

lappable pattern.

32.3-4 ?

Given two patterns P and P 0

, describe how to construct a ﬁnite automaton that

determines all occurrences of either pattern. Try to minimize the number of states

in your automaton.

32.3-5

Given a pattern P containing gap characters (see Exercise 32.1-4), show how to

build a ﬁnite automaton that can ﬁnd an occurrence of P in a text T in O.n/

matching time, where n D jT j.

? 32.4 The Knuth-Morris-Pratt algorithm

We now present a linear-time string-matching algorithm due to Knuth, Morris, and

Pratt. This algorithm avoids computing the transition function ı altogether, and its

matching time is ‚.n/ using just an auxiliary function , which we precompute

from the pattern in time ‚.m/ and store in an array Œ1 : : m. The array allows

us to compute the transition function ı efﬁciently (in an amortized sense) “on the

ﬂy” as needed. Loosely speaking, for any state q D 0; 1; : : : ; m and any character

32.4 The Knuth-Morris-Pratt algorithm 1003

a 2 †, the value Œq contains the information we need to compute ı.q; a/ but

that does not depend on a. Since the array has only m entries, whereas ı has

‚.m j†j/ entries, we save a factor of j†j in the preprocessing time by computing

rather than ı.

The preﬁx function for a pattern

The preﬁx function for a pattern encapsulates knowledge about how the pat-

tern matches against shifts of itself. We can take advantage of this information to

avoid testing useless shifts in the naive pattern-matching algorithm and to avoid

precomputing the full transition function ı for a string-matching automaton.

Consider the operation of the naive string matcher. Figure 32.10(a) shows a

particular shift s of a template containing the pattern P D ababaca against a

text T . For this example, q D 5 of the characters have matched successfully, but

the 6th pattern character fails to match the corresponding text character. The infor-

mation that q characters have matched successfully determines the corresponding

text characters. Knowing these q text characters allows us to determine immedi-

ately that certain shifts are invalid. In the example of the ﬁgure, the shift s C 1 is

necessarily invalid, since the ﬁrst pattern character (a) would be aligned with a text

character that we know does not match the ﬁrst pattern character, but does match

the second pattern character (b). The shift s 0 D s C 2 shown in part (b) of the ﬁg-

ure, however, aligns the ﬁrst three pattern characters with three text characters that

must necessarily match. In general, it is useful to know the answer to the following

question:

Given that pattern characters P Œ1 : : q match text characters T ŒsC1 : : sCq,

what is the least shift s 0 > s such that for some k < q,

P Œ1 : : k D T Œs

0

C 1 : : s

0

C k ; (32.6)

where s 0 C k D s C q?

In other words, knowing that P q T sCq , we want the longest proper preﬁx P k

of P q that is also a sufﬁx of T sCq . (Since s 0 C k D s C q, if we are given s

and q, then ﬁnding the smallest shift s 0

is tantamount to ﬁnding the longest preﬁx

length k.) We add the difference q k in the lengths of these preﬁxes of P to the

shift s to arrive at our new shift s 0

, so that s 0 D s C.q k/. In the best case, k D 0,

so that s 0 D s C q, and we immediately rule out shifts s C 1; s C 2; : : : ; s C q 1.

In any case, at the new shift s 0

we don’t need to compare the ﬁrst k characters of P

with the corresponding characters of T , since equation (32.6) guarantees that they

match.

We can precompute the necessary information by comparing the pattern against

itself, as Figure 32.10(c) demonstrates. Since T Œs 0 C 1 : : s 0 C k is part of the

1004 Chapter 32 String Matching

b a c b a b

a b a

(a)

a b a a b c b a b

b a c a

s

T

P

q

b a c b a b

a b a

(b)

a b a a b c b a b

b a c a

s ′ = s + 2

T

P

k

a b

a b a

a b a

(c)

P

q

P

k

Figure 32.10 The preﬁx function . (a) The pattern P D ababaca aligns with a text T so that

the ﬁrst q D 5 characters match. Matching characters, shown shaded, are connected by vertical lines.

(b) Using only our knowledge of the 5 matched characters, we can deduce that a shift of s C 1 is

invalid, but that a shift of s

0

D sC2 is consistent with everything we know about the text and therefore

is potentially valid. (c) We can precompute useful information for such deductions by comparing the

pattern with itself. Here, we see that the longest preﬁx of P that is also a proper sufﬁx of P5 is P3.

We represent this precomputed information in the array , so that Œ5 D 3. Given that q characters

have matched successfully at shift s, the next potentially valid shift is at s

0

D s C.q Œq/ as shown

in part (b).

known portion of the text, it is a sufﬁx of the string P q . Therefore, we can interpret

equation (32.6) as asking for the greatest k < q such that P k P q . Then, the new

shift s 0 D sC.qk/ is the next potentially valid shift. We will ﬁnd it convenient to

store, for each value of q, the number k of matching characters at the new shift s 0

,

rather than storing, say, s 0 s.

We formalize the information that we precompute as follows. Given a pattern

P Œ1 : : m, the preﬁx function for the pattern P is the function W f1; 2; : : : ; mg !

f0; 1; : : : ; m 1g such that

Œq D max fk W k < q and P k P q g :

That is, Œq is the length of the longest preﬁx of P that is a proper sufﬁx of P q .

Figure 32.11(a) gives the complete preﬁx function for the pattern ababaca.

32.4 The Knuth-Morris-Pratt algorithm 1005

1 2 3 4 5 6 7

0 0 1 2 3 0 1

a b a b a c a

(a)

a b a b a c a

a b a b a c a

a b a b a c a

a b a b a c a

(b)

"

i

P Œi

Œi

P 5

P 3

P 1

P 0

Œ5 D 3

Œ3 D 1

Œ1 D 0

Figure 32.11 An illustration of Lemma 32.5 for the pattern P D ababaca and q D 5. (a) The

function for the given pattern. Since Œ5 D 3, Œ3 D 1, and Œ1 D 0, by iterating we obtain

Œ5 D f3; 1; 0g. (b) We slide the template containing the pattern P to the right and note when some

preﬁx Pk of P matches up with some proper sufﬁx of P5; we get matches when k D 3, 1, and 0. In

the ﬁgure, the ﬁrst row gives P , and the dotted vertical line is drawn just after P5. Successive rows

show all the shifts of P that cause some preﬁx Pk of P to match some sufﬁx of P5. Successfully

matched characters are shown shaded. Vertical lines connect aligned matching characters. Thus,

fk W k < 5 and Pk P5g D f3; 1; 0g. Lemma 32.5 claims that

Œq D fk W k < q and Pk Pqg

for all q.

The pseudocode below gives the Knuth-Morris-Pratt matching algorithm as

the procedure KMP-MATCHER. For the most part, the procedure follows from

FINITE-AUTOMATON-MATCHER, as we shall see. KMP-MATCHER calls the aux-

iliary procedure COMPUTE-PREFIX-FUNCTION to compute .

KMP-MATCHER.T; P /

1 n D T:length

2 m D P:length

3 D COMPUTE-PREFIX-FUNCTION.P /

4 q D 0 // number of characters matched

5 for i D 1 to n // scan the text from left to right

6 while q > 0 and P Œq C 1 ¤ T Œi

7 q D Œq // next character does not match

8 if P Œq C 1 == T Œi

9 q D q C 1 // next character matches

10 if q == m // is all of P matched?

11 print “Pattern occurs with shift” i m

12 q D Œq // look for the next match

1006 Chapter 32 String Matching

COMPUTE-PREFIX-FUNCTION.P /

1 m D P:length

2 let Œ1 : : m be a new array

3 Œ1 D 0

4 k D 0

5 for q D 2 to m

6 while k > 0 and P Œk C 1 ¤ P Œq

7 k D Œk

8 if P Œk C 1 == P Œq

9 k D k C 1

10 Œq D k

11 return

These two procedures have much in common, because both match a string against

the pattern P : KMP-MATCHER matches the text T against P , and COMPUTE-

PREFIX-FUNCTION matches P against itself.

We begin with an analysis of the running times of these procedures. Proving

these procedures correct will be more complicated.

Running-time analysis

The running time of COMPUTE-PREFIX-FUNCTION is ‚.m/, which we show by

using the aggregate method of amortized analysis (see Section 17.1). The only

tricky part is showing that the while loop of lines 6–7 executes O.m/ times alto-

gether. We shall show that it makes at most m 1 iterations. We start by making

some observations about k. First, line 4 starts k at 0, and the only way that k

increases is by the increment operation in line 9, which executes at most once per

iteration of the for loop of lines 5–10. Thus, the total increase in k is at most m1.

Second, since k < q upon entering the for loop and each iteration of the loop in-

crements q, we always have k < q. Therefore, the assignments in lines 3 and 10

ensure that Œq < q for all q D 1; 2; : : : ; m, which means that each iteration of

the while loop decreases k. Third, k never becomes negative. Putting these facts

together, we see that the total decrease in k from the while loop is bounded from

above by the total increase in k over all iterations of the for loop, which is m 1.

Thus, the while loop iterates at most m 1 times in all, and COMPUTE-PREFIX-

FUNCTION runs in time ‚.m/.

Exercise 32.4-4 asks you to show, by a similar aggregate analysis, that the match-

ing time of KMP-MATCHER is ‚.n/.

Compared with FINITE-AUTOMATON-MATCHER, by using rather than ı, we

have reduced the time for preprocessing the pattern from O.m j†j/ to ‚.m/, while

keeping the actual matching time bounded by ‚.n/.

32.4 The Knuth-Morris-Pratt algorithm 1007

Correctness of the preﬁx-function computation

We shall see a little later that the preﬁx function helps us simulate the transition

function ı in a string-matching automaton. But ﬁrst, we need to prove that the

procedure COMPUTE-PREFIX-FUNCTION does indeed compute the preﬁx func-

tion correctly. In order to do so, we will need to ﬁnd all preﬁxes P k that are proper

sufﬁxes of a given preﬁx P q . The value of Œq gives us the longest such preﬁx, but

the following lemma, illustrated in Figure 32.11, shows that by iterating the preﬁx

function , we can indeed enumerate all the preﬁxes P k that are proper sufﬁxes

of P q . Let

Œq D fŒq;

.2/

Œq;

.3/

Œq; : : : ;

.t/

Œqg ;

where .i/ Œq is deﬁned in terms of functional iteration, so that .0/ Œq D q and

.i/ Œq D Œ .i1/ Œq for i 1, and where the sequence in Œq stops upon

reaching .t/ Œq D 0.

Lemma 32.5 (Preﬁx-function iteration lemma)

Let P be a pattern of length m with preﬁx function . Then, for q D 1; 2; : : : ; m,

we have Œq D fk W k < q and P k P q g.

Proof We ﬁrst prove that Œq fk W k < q and P k P q g or, equivalently,

i 2

Œq implies P i P q : (32.7)

If i 2 Œq, then i D .u/ Œq for some u > 0. We prove equation (32.7) by

induction on u. For u D 1, we have i D Œq, and the claim follows since i < q

and P Œq P q by the deﬁnition of . Using the relations Œi < i and P Œi P i

and the transitivity of < and establishes the claim for all i in Œq. Therefore,

Œq fk W k < q and P k P q g.

We now prove that fk W k < q and P k P q g Œq by contradiction. Sup-

pose to the contrary that the set fk W k < q and P k P q g Œq is nonempty,

and let j be the largest number in the set. Because Œq is the largest value in

fk W k < q and P k P q g and Œq 2 Œq, we must have j < Œq, and so we

let j 0

denote the smallest integer in Œq that is greater than j . (We can choose

j 0 D Œq if no other number in Œq is greater than j .) We have P j P q because

j 2 fk W k < q and P k P q g, and from j 0 2 Œq and equation (32.7), we have

P j 0 P q . Thus, P j P j 0 by Lemma 32.1, and j is the largest value less than j 0

with this property. Therefore, we must have Œj 0 D j and, since j 0 2 Œq, we

must have j 2 Œq as well. This contradiction proves the lemma.

The algorithm COMPUTE-PREFIX-FUNCTION computes Œq, in order, for q D

1; 2; : : : ; m. Setting Œ1 to 0 in line 3 of COMPUTE-PREFIX-FUNCTION is cer-

tainly correct, since Œq < q for all q. We shall use the following lemma and

1008 Chapter 32 String Matching

its corollary to prove that COMPUTE-PREFIX-FUNCTION computes Œq correctly

for q > 1.

Lemma 32.6

Let P be a pattern of length m, and let be the preﬁx function for P . For q D

1; 2; : : : ; m, if Œq > 0, then Œq 1 2 Œq 1.

Proof Let r D Œq > 0, so that r < q and P r P q ; thus, r 1 < q 1 and

P r1 P q1 (by dropping the last character from P r and P q , which we can do

because r > 0). By Lemma 32.5, therefore, r 1 2 Œq 1. Thus, we have

Œq 1 D r 1 2 Œq 1.

For q D 2; 3; : : : ; m, deﬁne the subset E q1 Œq 1 by

E q1 D fk 2

Œq 1 W P Œk C 1 D P Œqg

D fk W k < q 1 and P k P q1 and P Œk C 1 D P Œqg (by Lemma 32.5)

D fk W k < q 1 and P kC1 P q g :

The set E q1 consists of the values k < q 1 for which P k P q1 and for which,

because P Œk C 1 D P Œq, we have P kC1 P q . Thus, E q1 consists of those

values k 2 Œq 1 such that we can extend P k to P kC1 and get a proper sufﬁx

of P q .

Corollary 32.7

Let P be a pattern of length m, and let be the preﬁx function for P . For q D

2; 3; : : : ; m,

Œq D

(

0 if E q1 D ; ;

1 C max fk 2 E q1 g if E q1 ¤ ; :

Proof If E q1 is empty, there is no k 2 Œq 1 (including k D 0) for which

we can extend P k to P kC1 and get a proper sufﬁx of P q . Therefore Œq D 0.

If E q1 is nonempty, then for each k 2 E q1 we have kC1 < q and P kC1 P q .

Therefore, from the deﬁnition of Œq, we have

Œq 1 C max fk 2 E q1 g : (32.8)

Note that Œq > 0. Let r D Œq 1, so that r C 1 D Œq and there-

fore P rC1 P q . Since r C 1 > 0, we have P Œr C 1 D P Œq. Furthermore,

by Lemma 32.6, we have r 2 Œq 1. Therefore, r 2 E q1 , and so r

max fk 2 E q1 g or, equivalently,

Œq 1 C max fk 2 E q1 g : (32.9)

Combining equations (32.8) and (32.9) completes the proof.

32.4 The Knuth-Morris-Pratt algorithm 1009

We now ﬁnish the proof that COMPUTE-PREFIX-FUNCTION computes cor-

rectly. In the procedure COMPUTE-PREFIX-FUNCTION, at the start of each iter-

ation of the for loop of lines 5–10, we have that k D Œq 1. This condition

is enforced by lines 3 and 4 when the loop is ﬁrst entered, and it remains true in

each successive iteration because of line 10. Lines 6–9 adjust k so that it becomes

the correct value of Œq. The while loop of lines 6–7 searches through all values

k 2 Œq 1 until it ﬁnds a value of k for which P Œk C 1 D P Œq; at that point,

k is the largest value in the set E q1 , so that, by Corollary 32.7, we can set Œq

to k C 1. If the while loop cannot ﬁnd a k 2 Œq 1 such that P Œk C 1 D P Œq,

then k equals 0 at line 8. If P Œ1 D P Œq, then we should set both k and Œq to 1;

otherwise we should leave k alone and set Œq to 0. Lines 8–10 set k and Œq

correctly in either case. This completes our proof of the correctness of COMPUTE-

PREFIX-FUNCTION.

Correctness of the Knuth-Morris-Pratt algorithm

We can think of the procedure KMP-MATCHER as a reimplemented version of

the procedure FINITE-AUTOMATON-MATCHER, but using the preﬁx function

to compute state transitions. Speciﬁcally, we shall prove that in the ith iteration of

the for loops of both KMP-MATCHER and FINITE-AUTOMATON-MATCHER, the

state q has the same value when we test for equality with m (at line 10 in KMP-

MATCHER and at line 5 in FINITE-AUTOMATON-MATCHER). Once we have

argued that KMP-MATCHER simulates the behavior of FINITE-AUTOMATON-

MATCHER, the correctness of KMP-MATCHER follows from the correctness of

FINITE-AUTOMATON-MATCHER (though we shall see a little later why line 12 in

KMP-MATCHER is necessary).

Before we formally prove that KMP-MATCHER correctly simulates FINITE-

AUTOMATON-MATCHER, let’s take a moment to understand how the preﬁx func-

tion replaces the ı transition function. Recall that when a string-matching

automaton is in state q and it scans a character a D T Œi, it moves to a new

state ı.q; a/. If a D P Œq C 1, so that a continues to match the pattern, then

ı.q; a/ D q C 1. Otherwise, a ¤ P Œq C 1, so that a does not continue to match

the pattern, and 0 ı.q; a/ q. In the ﬁrst case, when a continues to match,

KMP-MATCHER moves to state q C 1 without referring to the function: the

while loop test in line 6 comes up false the ﬁrst time, the test in line 8 comes up

true, and line 9 increments q.

The function comes into play when the character a does not continue to match

the pattern, so that the new state ı.q; a/ is either q or to the left of q along the spine

of the automaton. The while loop of lines 6–7 in KMP-MATCHER iterates through

the states in Œq, stopping either when it arrives in a state, say q 0

, such that a

matches P Œq 0 C 1 or q 0

has gone all the way down to 0. If a matches P Œq 0 C 1,

1010 Chapter 32 String Matching

then line 9 sets the new state to q 0 C1, which should equal ı.q; a/ for the simulation

to work correctly. In other words, the new state ı.q; a/ should be either state 0 or

one greater than some state in Œq.

Let’s look at the example in Figures 32.7 and 32.11, which are for the pattern

P D ababaca. Suppose that the automaton is in state q D 5; the states in

Œ5 are, in descending order, 3, 1, and 0. If the next character scanned is c, then

we can easily see that the automaton moves to state ı.5; c/ D 6 in both FINITE-

AUTOMATON-MATCHER and KMP-MATCHER. Now suppose that the next char-

acter scanned is instead b, so that the automaton should move to state ı.5; b/ D 4.

The while loop in KMP-MATCHER exits having executed line 7 once, and it ar-

rives in state q 0 D Œ5 D 3. Since P Œq 0 C 1 D P Œ4 D b, the test in line 8

comes up true, and KMP-MATCHER moves to the new state q 0 C 1 D 4 D ı.5; b/.

Finally, suppose that the next character scanned is instead a, so that the automa-

ton should move to state ı.5; a/ D 1. The ﬁrst three times that the test in line 6

executes, the test comes up true. The ﬁrst time, we ﬁnd that P Œ6 D c ¤ a, and

KMP-MATCHER moves to state Œ5 D 3 (the ﬁrst state in Œ5). The second

time, we ﬁnd that P Œ4 D b ¤ a and move to state Œ3 D 1 (the second state

in Œ5). The third time, we ﬁnd that P Œ2 D b ¤ a and move to state Œ1 D 0

(the last state in Œ5). The while loop exits once it arrives in state q 0 D 0. Now,

line 8 ﬁnds that P Œq 0 C 1 D P Œ1 D a, and line 9 moves the automaton to the new

state q 0 C 1 D 1 D ı.5; a/.

Thus, our intuition is that KMP-MATCHER iterates through the states in Œq in

decreasing order, stopping at some state q 0

and then possibly moving to state q 0 C1.

Although that might seem like a lot of work just to simulate computing ı.q; a/,

bear in mind that asymptotically, KMP-MATCHER is no slower than FINITE-

AUTOMATON-MATCHER.

We are now ready to formally prove the correctness of the Knuth-Morris-Pratt

algorithm. By Theorem 32.4, we have that q D .T i / after each time we execute

line 4 of FINITE-AUTOMATON-MATCHER. Therefore, it sufﬁces to show that the

same property holds with regard to the for loop in KMP-MATCHER. The proof

proceeds by induction on the number of loop iterations. Initially, both procedures

set q to 0 as they enter their respective for loops for the ﬁrst time. Consider itera-

tion i of the for loop in KMP-MATCHER, and let q 0

be state at the start of this loop

iteration. By the inductive hypothesis, we have q 0 D .T i1 /. We need to show

that q D .T i / at line 10. (Again, we shall handle line 12 separately.)

When we consider the character T Œi, the longest preﬁx of P that is a sufﬁx of T i

is either P q 0 C1 (if P Œq 0 C 1 D T Œi) or some preﬁx (not necessarily proper, and

possibly empty) of P q 0 . We consider separately the three cases in which .T i / D 0,

.T i / D q 0 C 1, and 0 < .T i / q 0

.

32.4 The Knuth-Morris-Pratt algorithm 1011

If .T i / D 0, then P 0 D " is the only preﬁx of P that is a sufﬁx of T i . The while

loop of lines 6–7 iterates through the values in Œq 0 , but although P q T i for

every q 2 Œq 0 , the loop never ﬁnds a q such that P Œq C 1 D T Œi. The loop

terminates when q reaches 0, and of course line 9 does not execute. Therefore,

q D 0 at line 10, so that q D .T i /.

If .T i / D q 0 C 1, then P Œq 0 C 1 D T Œi, and the while loop test in line 6

fails the ﬁrst time through. Line 9 executes, incrementing q so that afterward

we have q D q 0 C 1 D .T i /.

If 0 < .T i / q 0

, then the while loop of lines 6–7 iterates at least once,

checking in decreasing order each value q 2 Œq 0 until it stops at some q < q 0

.

Thus, P q is the longest preﬁx of P q 0 for which P ŒqC1 D T Œi, so that when the

while loop terminates, q C 1 D .P q 0 T Œi/. Since q 0 D .T i1 /, Lemma 32.3

implies that .T i1 T Œi/ D .P q 0 T Œi/. Thus, we have

q C 1 D .P q 0 T Œi/

D .T i1 T Œi/

D .T i /

when the while loop terminates. After line 9 increments q, we have q D .T i /.

Line 12 is necessary in KMP-MATCHER, because otherwise, we might refer-

ence P Œm C 1 on line 6 after ﬁnding an occurrence of P . (The argument that

q D .T i1 / upon the next execution of line 6 remains valid by the hint given in

Exercise 32.4-8: ı.m; a/ D ı.Œm; a/ or, equivalently, .P a/ D .P Œm a/ for

any a 2 †.) The remaining argument for the correctness of the Knuth-Morris-

Pratt algorithm follows from the correctness of FINITE-AUTOMATON-MATCHER,

since we have shown that KMP-MATCHER simulates the behavior of FINITE-

AUTOMATON-MATCHER.

Exercises

32.4-1

Compute the preﬁx function for the pattern ababbabbabbababbabb.

32.4-2

Give an upper bound on the size of Œq as a function of q. Give an example to

show that your bound is tight.

32.4-3

Explain how to determine the occurrences of pattern P in the text T by examining

the function for the string P T (the string of length mCn that is the concatenation

of P and T ).

1012 Chapter 32 String Matching

32.4-4

Use an aggregate analysis to show that the running time of KMP-MATCHER

is ‚.n/.

32.4-5

Use a potential function to show that the running time of KMP-MATCHER is ‚.n/.

32.4-6

Show how to improve KMP-MATCHER by replacing the occurrence of in line 7

(but not line 12) by 0

, where 0

is deﬁned recursively for q D 1; 2; : : : ; m 1 by

the equation

0

Œq D

0 if Œq D 0 ;

0 ŒŒq if Œq ¤ 0 and P ŒŒq C 1 D P Œq C 1 ;

Œq if Œq ¤ 0 and P ŒŒq C 1 ¤ P Œq C 1 :

Explain why the modiﬁed algorithm is correct, and explain in what sense this

change constitutes an improvement.

32.4-7

Give a linear-time algorithm to determine whether a text T is a cyclic rotation of

another string T 0

. For example, arc and car are cyclic rotations of each other.

32.4-8 ?

Give an O.m j†j/-time algorithm for computing the transition function ı for the

string-matching automaton corresponding to a given pattern P . (Hint: Prove that

ı.q; a/ D ı.Œq; a/ if q D m or P Œq C 1 ¤ a.)

Problems

32-1 String matching based on repetition factors

Let y i

denote the concatenation of string y with itself i times. For example,

.ab/ 3 D ababab. We say that a string x 2 †

has repetition factor r if x D y r

for some string y 2 †

and some r > 0. Let .x/ denote the largest r such that x

has repetition factor r.

a. Give an efﬁcient algorithm that takes as input a pattern P Œ1 : : m and computes

the value .P i / for i D 1; 2; : : : ; m. What is the running time of your algo-

rithm?

Notes for Chapter 32 1013

b. For any pattern P Œ1 : : m, let .P / be deﬁned as max 1im .P i /. Prove that if

the pattern P is chosen randomly from the set of all binary strings of length m,

then the expected value of .P / is O.1/.

c. Argue that the following string-matching algorithm correctly ﬁnds all occur-

rences of pattern P in a text T Œ1 : : n in time O. .P /n C m/:

REPETITION-MATCHER.P; T /

1 m D P:length

2 n D T:length

3 k D 1 C .P /

4 q D 0

5 s D 0

6 while s n m

7 if T Œs C q C 1 == P Œq C 1

8 q D q C 1

9 if q == m

10 print “Pattern occurs with shift” s

11 if q == m or T Œs C q C 1 ¤ P Œq C 1

12 s D s C max.1; dq=ke/

13 q D 0

This algorithm is due to Galil and Seiferas. By extending these ideas greatly,

they obtained a linear-time string-matching algorithm that uses only O.1/ stor-

age beyond what is required for P and T .

Chapter notes

The relation of string matching to the theory of ﬁnite automata is discussed by

Aho, Hopcroft, and Ullman [5]. The Knuth-Morris-Pratt algorithm [214] was

invented independently by Knuth and Pratt and by Morris; they published their

work jointly. Reingold, Urban, and Gries [294] give an alternative treatment of the

Knuth-Morris-Pratt algorithm. The Rabin-Karp algorithm was proposed by Karp

and Rabin [201]. Galil and Seiferas [126] give an interesting deterministic linear-

time string-matching algorithm that uses only O.1/ space beyond that required to

store the pattern and text.

33 Computational Geometry

Computational geometry is the branch of computer science that studies algorithms

for solving geometric problems. In modern engineering and mathematics, com-

putational geometry has applications in such diverse ﬁelds as computer graphics,

robotics, VLSI design, computer-aided design, molecular modeling, metallurgy,

manufacturing, textile layout, forestry, and statistics. The input to a computational-

geometry problem is typically a description of a set of geometric objects, such as

a set of points, a set of line segments, or the vertices of a polygon in counterclock-

wise order. The output is often a response to a query about the objects, such as

whether any of the lines intersect, or perhaps a new geometric object, such as the

convex hull (smallest enclosing convex polygon) of the set of points.

In this chapter, we look at a few computational-geometry algorithms in two

dimensions, that is, in the plane. We represent each input object by a set of

points fp 1 ; p 2 ; p 3 ; : : :g, where each p i D .x i ; y i / and x i ; y i 2 R . For exam-

ple, we represent an n-vertex polygon P by a sequence hp 0 ; p 1 ; p 2 ; : : : ; p n1 i

of its vertices in order of their appearance on the boundary of P . Computational

geometry can also apply to three dimensions, and even higher-dimensional spaces,

but such problems and their solutions can be very difﬁcult to visualize. Even in

two dimensions, however, we can see a good sample of computational-geometry

techniques.

Section 33.1 shows how to answer basic questions about line segments efﬁ-

ciently and accurately: whether one segment is clockwise or counterclockwise

from another that shares an endpoint, which way we turn when traversing two

adjoining line segments, and whether two line segments intersect. Section 33.2

presents a technique called “sweeping” that we use to develop an O.n lg n/-time

algorithm for determining whether a set of n line segments contains any inter-

sections. Section 33.3 gives two “rotational-sweep” algorithms that compute the

convex hull (smallest enclosing convex polygon) of a set of n points: Graham’s

scan, which runs in time O.n lg n/, and Jarvis’s march, which takes O.nh/ time,

where h is the number of vertices of the convex hull. Finally, Section 33.4 gives

33.1 Line-segment properties 1015

an O.n lg n/-time divide-and-conquer algorithm for ﬁnding the closest pair of

points in a set of n points in the plane.

33.1 Line-segment properties

Several of the computational-geometry algorithms in this chapter require answers

to questions about the properties of line segments. A convex combination of two

distinct points p 1 D .x 1 ; y 1 / and p 2 D .x 2 ; y 2 / is any point p 3 D .x 3 ; y 3 / such

that for some ˛ in the range 0 ˛ 1, we have x 3 D ˛x 1 C .1 ˛/x 2 and

y 3 D ˛y 1 C .1 ˛/y 2 . We also write that p 3 D ˛p 1 C .1 ˛/p 2 . Intuitively, p 3

is any point that is on the line passing through p 1 and p 2 and is on or between p 1

and p 2 on the line. Given two distinct points p 1 and p 2 , the line segment p 1 p 2

is the set of convex combinations of p 1 and p 2 . We call p 1 and p 2 the endpoints

of segment p 1 p 2 . Sometimes the ordering of p 1 and p 2 matters, and we speak of

the directed segment

!

p 1 p 2 . If p 1 is the origin .0; 0/, then we can treat the directed

segment

!

p 1 p 2 as the vector p 2 .

In this section, we shall explore the following questions:

1. Given two directed segments

!

p 0 p 1 and

!

p 0 p 2 , is

!

p 0 p 1 clockwise from

!

p 0 p 2

with respect to their common endpoint p 0 ?

2. Given two line segments p 0 p 1 and p 1 p 2 , if we traverse p 0 p 1 and then p 1 p 2 ,

do we make a left turn at point p 1 ?

3. Do line segments p 1 p 2 and p 3 p 4 intersect?

There are no restrictions on the given points.

We can answer each question in O.1/ time, which should come as no surprise

since the input size of each question is O.1/. Moreover, our methods use only ad-

ditions, subtractions, multiplications, and comparisons. We need neither division

nor trigonometric functions, both of which can be computationally expensive and

prone to problems with round-off error. For example, the “straightforward” method

of determining whether two segments intersect—compute the line equation of the

form y D mx C b for each segment (m is the slope and b is the y-intercept),

ﬁnd the point of intersection of the lines, and check whether this point is on both

segments—uses division to ﬁnd the point of intersection. When the segments are

nearly parallel, this method is very sensitive to the precision of the division opera-

tion on real computers. The method in this section, which avoids division, is much

more accurate.

1016 Chapter 33 Computational Geometry

p

2

x

y

(0,0)

p

1

p

1

+ p

2

(a) (b)

y

x

(0,0)

p

Figure 33.1 (a) The cross product of vectors p1 and p2 is the signed area of the parallelogram.

(b) The lightly shaded region contains vectors that are clockwise from p. The darkly shaded region

contains vectors that are counterclockwise from p.

Cross products

Computing cross products lies at the heart of our line-segment methods. Consider

vectors p 1 and p 2 , shown in Figure 33.1(a). We can interpret the cross product

p 1 p 2 as the signed area of the parallelogram formed by the points .0; 0/, p 1 , p 2 ,

and p 1 Cp 2 D .x 1 Cx 2 ; y 1 Cy 2 /. An equivalent, but more useful, deﬁnition gives

the cross product as the determinant of a matrix: 1

p 1 p 2 D det

x 1 x 2

y 1 y 2

D x 1 y 2 x 2 y 1

D p 2 p 1 :

If p 1 p 2 is positive, then p 1 is clockwise from p 2 with respect to the origin .0; 0/;

if this cross product is negative, then p 1 is counterclockwise from p 2 . (See Exer-

cise 33.1-1.) Figure 33.1(b) shows the clockwise and counterclockwise regions

relative to a vector p. A boundary condition arises if the cross product is 0; in this

case, the vectors are colinear, pointing in either the same or opposite directions.

To determine whether a directed segment

!

p 0 p 1 is closer to a directed seg-

ment

!

p 0 p 2 in a clockwise direction or in a counterclockwise direction with respect

to their common endpoint p 0 , we simply translate to use p 0 as the origin. That

is, we let p 1 p 0 denote the vector p 0

1

D .x 0

1

; y 0

1

/, where x 0

1

D x 1 x 0 and

y 0

1

D y 1 y 0 , and we deﬁne p 2 p 0 similarly. We then compute the cross product

1

Actually, the cross product is a three-dimensional concept. It is a vector that is perpendicular to

both p1 and p2 according to the “right-hand rule” and whose magnitude is jx1y2 x2y1j. In this

chapter, however, we ﬁnd it convenient to treat the cross product simply as the value x1y2 x2y1.

33.1 Line-segment properties 1017

p

0

p

1

p

2

p

0

p

1

p

2

counterclockwise

(a) (b)

clockwise

Figure 33.2 Using the cross product to determine how consecutive line segments p0p1 and p1p2

turn at point p1. We check whether the directed segment

!

p0p2 is clockwise or counterclockwise

relative to the directed segment

!

p0p1. (a) If counterclockwise, the points make a left turn. (b) If

clockwise, they make a right turn.

.p 1 p 0 / .p 2 p 0 / D .x 1 x 0 /.y 2 y 0 / .x 2 x 0 /.y 1 y 0 / :

If this cross product is positive, then

!

p 0 p 1 is clockwise from

!

p 0 p 2 ; if negative, it

is counterclockwise.

Determining whether consecutive segments turn left or right

Our next question is whether two consecutive line segments p 0 p 1 and p 1 p 2 turn

left or right at point p 1 . Equivalently, we want a method to determine which way a

given angle †p 0 p 1 p 2 turns. Cross products allow us to answer this question with-

out computing the angle. As Figure 33.2 shows, we simply check whether directed

segment

!

p 0 p 2 is clockwise or counterclockwise relative to directed segment

!

p 0 p 1 .

To do so, we compute the cross product .p 2 p 0 / .p 1 p 0 /. If the sign of

this cross product is negative, then

!

p 0 p 2 is counterclockwise with respect to

!

p 0 p 1 ,

and thus we make a left turn at p 1 . A positive cross product indicates a clockwise

orientation and a right turn. A cross product of 0 means that points p 0 , p 1 , and p 2

are colinear.

Determining whether two line segments intersect

To determine whether two line segments intersect, we check whether each segment

straddles the line containing the other. A segment p 1 p 2 straddles a line if point p 1

lies on one side of the line and point p 2 lies on the other side. A boundary case

arises if p 1 or p 2 lies directly on the line. Two line segments intersect if and only

if either (or both) of the following conditions holds:

1. Each segment straddles the line containing the other.

2. An endpoint of one segment lies on the other segment. (This condition comes

from the boundary case.)

1018 Chapter 33 Computational Geometry

The following procedures implement this idea. SEGMENTS-INTERSECT returns

TRUE if segments p 1 p 2 and p 3 p 4 intersect and FALSE if they do not. It calls

the subroutines DIRECTION, which computes relative orientations using the cross-

product method above, and ON-SEGMENT, which determines whether a point

known to be colinear with a segment lies on that segment.

SEGMENTS-INTERSECT.p 1 ; p 2 ; p 3 ; p 4 /

1 d 1 D DIRECTION.p 3 ; p 4 ; p 1 /

2 d 2 D DIRECTION.p 3 ; p 4 ; p 2 /

3 d 3 D DIRECTION.p 1 ; p 2 ; p 3 /

4 d 4 D DIRECTION.p 1 ; p 2 ; p 4 /

5 if ..d 1 > 0 and d 2 < 0/ or .d 1 < 0 and d 2 > 0// and

..d 3 > 0 and d 4 < 0/ or .d 3 < 0 and d 4 > 0//

6 return TRUE

7 elseif d 1 == 0 and ON-SEGMENT.p 3 ; p 4 ; p 1 /

8 return TRUE

9 elseif d 2 == 0 and ON-SEGMENT.p 3 ; p 4 ; p 2 /

10 return TRUE

11 elseif d 3 == 0 and ON-SEGMENT.p 1 ; p 2 ; p 3 /

12 return TRUE

13 elseif d 4 == 0 and ON-SEGMENT.p 1 ; p 2 ; p 4 /

14 return TRUE

15 else return FALSE

DIRECTION.p i ; p j ; p k /

1 return .p k p i / .p j p i /

ON-SEGMENT.p i ; p j ; p k /

1 if min.x i ; x j / x k max.x i ; x j / and min.y i ; y j / y k max.y i ; y j /

2 return TRUE

3 else return FALSE

SEGMENTS-INTERSECT works as follows. Lines 1–4 compute the relative ori-

entation d i of each endpoint p i with respect to the other segment. If all the relative

orientations are nonzero, then we can easily determine whether segments p 1 p 2

and p 3 p 4 intersect, as follows. Segment p 1 p 2 straddles the line containing seg-

ment p 3 p 4 if directed segments

!

p 3 p 1 and

!

p 3 p 2 have opposite orientations relative

to

!

p 3 p 4 . In this case, the signs of d 1 and d 2 differ. Similarly, p 3 p 4 straddles

the line containing p 1 p 2 if the signs of d 3 and d 4 differ. If the test of line 5 is

true, then the segments straddle each other, and SEGMENTS-INTERSECT returns

TRUE. Figure 33.3(a) shows this case. Otherwise, the segments do not straddle

33.1 Line-segment properties 1019

p

1

p

2

p

3

p

4

(p

1

–p

3

) × (p

4

–p

3

) < 0

(p

4

–p

1

) × (p

2

–p

1

) < 0

(p

2

–p

3

) × (p

4

–p

3

) > 0

(p

3

–p

1

) × (p

2

–p

1

) > 0

(a)

p

1

p

2

p

3

p

4

(p

1

–p

3

) × (p

4

–p

3

) < 0

(p

4

–p

1

) × (p

2

–p

1

) < 0

(p

2

–p

3

) × (p

4

–p

3

) < 0

(p

3

–p

1

) × (p

2

–p

1

) > 0

(b)

p

1

p

2

p

3

p

4

(c)

p

1

p

2

p

3

p

4

(d)

Figure 33.3 Cases in the procedure SEGMENTS-INTERSECT. (a) The segments p1p2 and p3p4

straddle each other’s lines. Because p3p4 straddles the line containing p1p2, the signs of the cross

products .p3 p1/ .p2 p1/ and .p4 p1/ .p2 p1/ differ. Because p1p2 straddles the line

containing p3p4, the signs of the cross products .p1 p3/ .p4 p3/ and .p2 p3/ .p4 p3/

differ. (b) Segment p3p4 straddles the line containing p1p2, but p1p2 does not straddle the line

containing p3p4. The signs of the cross products .p1 p3/ .p4 p3/ and .p2 p3/ .p4 p3/

are the same. (c) Point p3 is colinear with p1p2 and is between p1 and p2. (d) Point p3 is colinear

with p1p2, but it is not between p1 and p2. The segments do not intersect.

each other’s lines, although a boundary case may apply. If all the relative orienta-

tions are nonzero, no boundary case applies. All the tests against 0 in lines 7–13

then fail, and SEGMENTS-INTERSECT returns FALSE in line 15. Figure 33.3(b)

shows this case.

A boundary case occurs if any relative orientation d k is 0. Here, we know that p k

is colinear with the other segment. It is directly on the other segment if and only

if it is between the endpoints of the other segment. The procedure ON-SEGMENT

returns whether p k is between the endpoints of segment p i p j , which will be the

other segment when called in lines 7–13; the procedure assumes that p k is colinear

with segment p i p j . Figures 33.3(c) and (d) show cases with colinear points. In

Figure 33.3(c), p 3 is on p 1 p 2 , and so SEGMENTS-INTERSECT returns TRUE in

line 12. No endpoints are on other segments in Figure 33.3(d), and so SEGMENTS-

INTERSECT returns FALSE in line 15.

1020 Chapter 33 Computational Geometry

Other applications of cross products

Later sections of this chapter introduce additional uses for cross products. In Sec-

tion 33.3, we shall need to sort a set of points according to their polar angles with

respect to a given origin. As Exercise 33.1-3 asks you to show, we can use cross

products to perform the comparisons in the sorting procedure. In Section 33.2, we

shall use red-black trees to maintain the vertical ordering of a set of line segments.

Rather than keeping explicit key values which we compare to each other in the

red-black tree code, we shall compute a cross-product to determine which of two

segments that intersect a given vertical line is above the other.

Exercises

33.1-1

Prove that if p 1 p 2 is positive, then vector p 1 is clockwise from vector p 2 with

respect to the origin .0; 0/ and that if this cross product is negative, then p 1 is

counterclockwise from p 2 .

33.1-2

Professor van Pelt proposes that only the x-dimension needs to be tested in line 1

of ON-SEGMENT. Show why the professor is wrong.

33.1-3

The polar angle of a point p 1 with respect to an origin point p 0 is the angle of the

vector p 1 p 0 in the usual polar coordinate system. For example, the polar angle

of .3; 5/ with respect to .2; 4/ is the angle of the vector .1; 1/, which is 45 degrees

or =4 radians. The polar angle of .3; 3/ with respect to .2; 4/ is the angle of the

vector .1; 1/, which is 315 degrees or 7=4 radians. Write pseudocode to sort a

sequence hp 1 ; p 2 ; : : : ; p n i of n points according to their polar angles with respect

to a given origin point p 0 . Your procedure should take O.n lg n/ time and use cross

products to compare angles.

33.1-4

Show how to determine in O.n 2

lg n/ time whether any three points in a set of n

points are colinear.

33.1-5

A polygon is a piecewise-linear, closed curve in the plane. That is, it is a curve

ending on itself that is formed by a sequence of straight-line segments, called the

sides of the polygon. A point joining two consecutive sides is a vertex of the poly-

gon. If the polygon is simple, as we shall generally assume, it does not cross itself.

The set of points in the plane enclosed by a simple polygon forms the interior of

33.2 Determining whether any pair of segments intersects 1021

the polygon, the set of points on the polygon itself forms its boundary, and the set

of points surrounding the polygon forms its exterior. A simple polygon is convex

if, given any two points on its boundary or in its interior, all points on the line

segment drawn between them are contained in the polygon’s boundary or interior.

A vertex of a convex polygon cannot be expressed as a convex combination of any

two distinct points on the boundary or in the interior of the polygon.

Professor Amundsen proposes the following method to determine whether a se-

quence hp 0 ; p 1 ; : : : ; p n1 i of n points forms the consecutive vertices of a convex

polygon. Output “yes” if the set f†p i p iC1 p iC2 W i D 0; 1; : : : ; n 1g, where sub-

script addition is performed modulo n, does not contain both left turns and right

turns; otherwise, output “no.” Show that although this method runs in linear time,

it does not always produce the correct answer. Modify the professor’s method so

that it always produces the correct answer in linear time.

33.1-6

Given a point p 0 D .x 0 ; y 0 /, the right horizontal ray from p 0 is the set of points

fp i D .x i ; y i / W x i x 0 and y i D y 0 g, that is, it is the set of points due right of p 0

along with p 0 itself. Show how to determine whether a given right horizontal ray

from p 0 intersects a line segment p 1 p 2 in O.1/ time by reducing the problem to

that of determining whether two line segments intersect.

33.1-7

One way to determine whether a point p 0 is in the interior of a simple, but not

necessarily convex, polygon P is to look at any ray from p 0 and check that the ray

intersects the boundary of P an odd number of times but that p 0 itself is not on

the boundary of P . Show how to compute in ‚.n/ time whether a point p 0 is in

the interior of an n-vertex polygon P . (Hint: Use Exercise 33.1-6. Make sure your

algorithm is correct when the ray intersects the polygon boundary at a vertex and

when the ray overlaps a side of the polygon.)

33.1-8

Show how to compute the area of an n-vertex simple, but not necessarily convex,

polygon in ‚.n/ time. (See Exercise 33.1-5 for deﬁnitions pertaining to polygons.)

33.2 Determining whether any pair of segments intersects

This section presents an algorithm for determining whether any two line segments

in a set of segments intersect. The algorithm uses a technique known as “sweep-

ing,” which is common to many computational-geometry algorithms. Moreover, as

1022 Chapter 33 Computational Geometry

the exercises at the end of this section show, this algorithm, or simple variations of

it, can help solve other computational-geometry problems.

The algorithm runs in O.n lg n/ time, where n is the number of segments we are

given. It determines only whether or not any intersection exists; it does not print

all the intersections. (By Exercise 33.2-1, it takes .n 2 / time in the worst case to

ﬁnd all the intersections in a set of n line segments.)

In sweeping, an imaginary vertical sweep line passes through the given set of

geometric objects, usually from left to right. We treat the spatial dimension that

the sweep line moves across, in this case the x-dimension, as a dimension of

time. Sweeping provides a method for ordering geometric objects, usually by plac-

ing them into a dynamic data structure, and for taking advantage of relationships

among them. The line-segment-intersection algorithm in this section considers all

the line-segment endpoints in left-to-right order and checks for an intersection each

time it encounters an endpoint.

To describe and prove correct our algorithm for determining whether any two

of n line segments intersect, we shall make two simplifying assumptions. First, we

assume that no input segment is vertical. Second, we assume that no three input

segments intersect at a single point. Exercises 33.2-8 and 33.2-9 ask you to show

that the algorithm is robust enough that it needs only a slight modiﬁcation to work

even when these assumptions do not hold. Indeed, removing such simplifying

assumptions and dealing with boundary conditions often present the most difﬁcult

challenges when programming computational-geometry algorithms and proving

their correctness.

Ordering segments

Because we assume that there are no vertical segments, we know that any input

segment intersecting a given vertical sweep line intersects it at a single point. Thus,

we can order the segments that intersect a vertical sweep line according to the y-

coordinates of the points of intersection.

To be more precise, consider two segments s 1 and s 2 . We say that these segments

are comparable at x if the vertical sweep line with x-coordinate x intersects both of

them. We say that s 1 is above s 2 at x, written s 1 < x s 2 , if s 1 and s 2 are comparable

at x and the intersection of s 1 with the sweep line at x is higher than the intersection

of s 2 with the same sweep line, or if s 1 and s 2 intersect at the sweep line. In

Figure 33.4(a), for example, we have the relationships a < r c, a < t b, b < t c,

a < t c, and b < u c. Segment d is not comparable with any other segment.

For any given x, the relation “ < x ” is a total preorder (see Section B.2) for all

segments that intersect the sweep line at x. That is, the relation is transitive, and

if segments s 1 and s 2 each intersect the sweep line at x, then either s 1 < x s 2

or s 2 < x s 1 , or both (if s 1 and s 2 intersect at the sweep line). (The relation < x is

33.2 Determining whether any pair of segments intersects 1023

r t u

a

c

b

d

(a) (b)

v w

e

f

g

h

i

z

Figure 33.4 The ordering among line segments at various vertical sweep lines. (a) We have a < r c,

a < t b, b < t c, a < t c, and b < u c. Segment d is comparable with no other segment shown.

(b) When segments e and f intersect, they reverse their orders: we have e < f but f < w e. Any

sweep line (such as ´) that passes through the shaded region has e and f consecutive in the ordering

given by the relation < ´.

also reﬂexive, but neither symmetric nor antisymmetric.) The total preorder may

differ for differing values of x, however, as segments enter and leave the ordering.

A segment enters the ordering when its left endpoint is encountered by the sweep,

and it leaves the ordering when its right endpoint is encountered.

What happens when the sweep line passes through the intersection of two seg-

ments? As Figure 33.4(b) shows, the segments reverse their positions in the total

preorder. Sweep lines and w are to the left and right, respectively, of the point

of intersection of segments e and f , and we have e < f and f < w e. Note

that because we assume that no three segments intersect at the same point, there

must be some vertical sweep line x for which intersecting segments e and f are

consecutive in the total preorder < x . Any sweep line that passes through the shaded

region of Figure 33.4(b), such as ´, has e and f consecutive in its total preorder.

Moving the sweep line

Sweeping algorithms typically manage two sets of data:

1. The sweep-line status gives the relationships among the objects that the sweep

line intersects.

2. The event-point schedule is a sequence of points, called event points, which

we order from left to right according to their x-coordinates. As the sweep

progresses from left to right, whenever the sweep line reaches the x-coordinate

of an event point, the sweep halts, processes the event point, and then resumes.

Changes to the sweep-line status occur only at event points.

For some algorithms (the algorithm asked for in Exercise 33.2-7, for example),

the event-point schedule develops dynamically as the algorithm progresses. The al-

gorithm at hand, however, determines all the event points before the sweep, based

1024 Chapter 33 Computational Geometry

solely on simple properties of the input data. In particular, each segment endpoint

is an event point. We sort the segment endpoints by increasing x-coordinate and

proceed from left to right. (If two or more endpoints are covertical, i.e., they have

the same x-coordinate, we break the tie by putting all the covertical left endpoints

before the covertical right endpoints. Within a set of covertical left endpoints, we

put those with lower y-coordinates ﬁrst, and we do the same within a set of cover-

tical right endpoints.) When we encounter a segment’s left endpoint, we insert the

segment into the sweep-line status, and we delete the segment from the sweep-line

status upon encountering its right endpoint. Whenever two segments ﬁrst become

consecutive in the total preorder, we check whether they intersect.

The sweep-line status is a total preorder T , for which we require the following

operations:

INSERT.T; s/: insert segment s into T .

DELETE.T; s/: delete segment s from T .

ABOVE.T; s/: return the segment immediately above segment s in T .

BELOW.T; s/: return the segment immediately below segment s in T .

It is possible for segments s 1 and s 2 to be mutually above each other in the total

preorder T ; this situation can occur if s 1 and s 2 intersect at the sweep line whose

total preorder is given by T . In this case, the two segments may appear in either

order in T .

If the input contains n segments, we can perform each of the operations INSERT,

DELETE, ABOVE, and BELOW in O.lg n/ time using red-black trees. Recall that

the red-black-tree operations in Chapter 13 involve comparing keys. We can re-

place the key comparisons by comparisons that use cross products to determine the

relative ordering of two segments (see Exercise 33.2-2).

Segment-intersection pseudocode

The following algorithm takes as input a set S of n line segments, returning the

boolean value TRUE if any pair of segments in S intersects, and FALSE otherwise.

A red-black tree maintains the total preorder T .

33.2 Determining whether any pair of segments intersects 1025

ANY-SEGMENTS-INTERSECT.S/

1 T D ;

2 sort the endpoints of the segments in S from left to right,

breaking ties by putting left endpoints before right endpoints

and breaking further ties by putting points with lower

y-coordinates ﬁrst

3 for each point p in the sorted list of endpoints

4 if p is the left endpoint of a segment s

5 INSERT.T; s/

6 if (ABOVE.T; s/ exists and intersects s)

or (BELOW.T; s/ exists and intersects s)

7 return TRUE

8 if p is the right endpoint of a segment s

9 if both ABOVE.T; s/ and BELOW.T; s/ exist

and ABOVE.T; s/ intersects BELOW.T; s/

10 return TRUE

11 DELETE.T; s/

12 return FALSE

Figure 33.5 illustrates how the algorithm works. Line 1 initializes the total preorder

to be empty. Line 2 determines the event-point schedule by sorting the 2n segment

endpoints from left to right, breaking ties as described above. One way to perform

line 2 is by lexicographically sorting the endpoints on .x; e; y/, where x and y are

the usual coordinates, e D 0 for a left endpoint, and e D 1 for a right endpoint.

Each iteration of the for loop of lines 3–11 processes one event point p. If p is

the left endpoint of a segment s, line 5 adds s to the total preorder, and lines 6–7

return TRUE if s intersects either of the segments it is consecutive with in the total

preorder deﬁned by the sweep line passing through p. (A boundary condition

occurs if p lies on another segment s 0

. In this case, we require only that s and s 0

be placed consecutively into T .) If p is the right endpoint of a segment s, then

we need to delete s from the total preorder. But ﬁrst, lines 9–10 return TRUE if

there is an intersection between the segments surrounding s in the total preorder

deﬁned by the sweep line passing through p. If these segments do not intersect,

line 11 deletes segment s from the total preorder. If the segments surrounding

segment s intersect, they would have become consecutive after deleting s had the

return statement in line 10 not prevented line 11 from executing. The correctness

argument, which follows, will make it clear why it sufﬁces to check the segments

surrounding s. Finally, if we never ﬁnd any intersections after having processed

all 2n event points, line 12 returns FALSE.

1026 Chapter 33 Computational Geometry

a a

b

a

c

b

d

a

c

b

d

c

b

e

d

c

b

a

b

c

d

e

f

time

Figure 33.5 The execution of ANY-SEGMENTS-INTERSECT. Each dashed line is the sweep line at

an event point. Except for the rightmost sweep line, the ordering of segment names below each sweep

line corresponds to the total preorder T at the end of the for loop processing the corresponding event

point. The rightmost sweep line occurs when processing the right endpoint of segment c; because

segments d and b surround c and intersect each other, the procedure returns TRUE.

Correctness

To show that ANY-SEGMENTS-INTERSECT is correct, we will prove that the call

ANY-SEGMENTS-INTERSECT.S/ returns TRUE if and only if there is an intersec-

tion among the segments in S.

It is easy to see that ANY-SEGMENTS-INTERSECT returns TRUE (on lines 7

and 10) only if it ﬁnds an intersection between two of the input segments. Hence,

if it returns TRUE, there is an intersection.

We also need to show the converse: that if there is an intersection, then ANY-

SEGMENTS-INTERSECT returns TRUE. Let us suppose that there is at least one

intersection. Let p be the leftmost intersection point, breaking ties by choosing the

point with the lowest y-coordinate, and let a and b be the segments that intersect

at p. Since no intersections occur to the left of p, the order given by T is correct at

all points to the left of p. Because no three segments intersect at the same point, a

and b become consecutive in the total preorder at some sweep line ´. 2 Moreover,

´ is to the left of p or goes through p. Some segment endpoint q on sweep line ´

2

If we allow three segments to intersect at the same point, there may be an intervening segment c that

intersects both a and b at point p. That is, we may have a < w c and c < w b for all sweep lines w to

the left of p for which a < w b. Exercise 33.2-8 asks you to show that ANY-SEGMENTS-INTERSECT

is correct even if three segments do intersect at the same point.

33.2 Determining whether any pair of segments intersects 1027

is the event point at which a and b become consecutive in the total preorder. If p

is on sweep line ´, then q D p. If p is not on sweep line ´, then q is to the left

of p. In either case, the order given by T is correct just before encountering q.

(Here is where we use the lexicographic order in which the algorithm processes

event points. Because p is the lowest of the leftmost intersection points, even if p

is on sweep line ´ and some other intersection point p 0

is on ´, event point q D p

is processed before the other intersection p 0

can interfere with the total preorder T .

Moreover, even if p is the left endpoint of one segment, say a, and the right end-

point of the other segment, say b, because left endpoint events occur before right

endpoint events, segment b is in T upon ﬁrst encountering segment a.) Either event

point q is processed by ANY-SEGMENTS-INTERSECT or it is not processed.

If q is processed by ANY-SEGMENTS-INTERSECT, only two possible actions

may occur:

1. Either a or b is inserted into T , and the other segment is above or below it in

the total preorder. Lines 4–7 detect this case.

2. Segments a and b are already in T , and a segment between them in the total

preorder is deleted, making a and b become consecutive. Lines 8–11 detect this

case.

In either case, we ﬁnd the intersection p and ANY-SEGMENTS-INTERSECT returns

TRUE.

If event point q is not processed by ANY-SEGMENTS-INTERSECT, the proce-

dure must have returned before processing all event points. This situation could

have occurred only if ANY-SEGMENTS-INTERSECT had already found an inter-

section and returned TRUE.

Thus, if there is an intersection, ANY-SEGMENTS-INTERSECT returns TRUE.

As we have already seen, if ANY-SEGMENTS-INTERSECT returns TRUE, there is

an intersection. Therefore, ANY-SEGMENTS-INTERSECT always returns a correct

answer.

Running time

If set S contains n segments, then ANY-SEGMENTS-INTERSECT runs in time

O.n lg n/. Line 1 takes O.1/ time. Line 2 takes O.n lg n/ time, using merge

sort or heapsort. The for loop of lines 3–11 iterates at most once per event point,

and so with 2n event points, the loop iterates at most 2n times. Each iteration takes

O.lg n/ time, since each red-black-tree operation takes O.lg n/ time and, using the

method of Section 33.1, each intersection test takes O.1/ time. The total time is

thus O.n lg n/.

1028 Chapter 33 Computational Geometry

Exercises

33.2-1

Show that a set of n line segments may contain ‚.n 2 / intersections.

33.2-2

Given two segments a and b that are comparable at x, show how to determine

in O.1/ time which of a < x b or b < x a holds. Assume that neither segment

is vertical. (Hint: If a and b do not intersect, you can just use cross products.

If a and b intersect—which you can of course determine using only cross prod-

ucts—you can still use only addition, subtraction, and multiplication, avoiding

division. Of course, in the application of the < x relation used here, if a and b

intersect, we can just stop and declare that we have found an intersection.)

33.2-3

Professor Mason suggests that we modify ANY-SEGMENTS-INTERSECT so that

instead of returning upon ﬁnding an intersection, it prints the segments that inter-

sect and continues on to the next iteration of the for loop. The professor calls the

resulting procedure PRINT-INTERSECTING-SEGMENTS and claims that it prints

all intersections, from left to right, as they occur in the set of line segments. Pro-

fessor Dixon disagrees, claiming that Professor Mason’s idea is incorrect. Which

professor is right? Will PRINT-INTERSECTING-SEGMENTS always ﬁnd the left-

most intersection ﬁrst? Will it always ﬁnd all the intersections?

33.2-4

Give an O.n lg n/-time algorithm to determine whether an n-vertex polygon is

simple.

33.2-5

Give an O.n lg n/-time algorithm to determine whether two simple polygons with

a total of n vertices intersect.

33.2-6

A disk consists of a circle plus its interior and is represented by its center point and

radius. Two disks intersect if they have any point in common. Give an O.n lg n/-

time algorithm to determine whether any two disks in a set of n intersect.

33.2-7

Given a set of n line segments containing a total of k intersections, show how to

output all k intersections in O..n C k/ lg n/ time.

33.3 Finding the convex hull 1029

33.2-8

Argue that ANY-SEGMENTS-INTERSECT works correctly even if three or more

segments intersect at the same point.

33.2-9

Show that ANY-SEGMENTS-INTERSECT works correctly in the presence of verti-

cal segments if we treat the bottom endpoint of a vertical segment as if it were a

left endpoint and the top endpoint as if it were a right endpoint. How does your

answer to Exercise 33.2-2 change if we allow vertical segments?

33.3 Finding the convex hull

The convex hull of a set Q of points, denoted by CH.Q/, is the smallest convex

polygon P for which each point in Q is either on the boundary of P or in its

interior. (See Exercise 33.1-5 for a precise deﬁnition of a convex polygon.) We

implicitly assume that all points in the set Q are unique and that Q contains at

least three points which are not colinear. Intuitively, we can think of each point

in Q as being a nail sticking out from a board. The convex hull is then the shape

formed by a tight rubber band that surrounds all the nails. Figure 33.6 shows a set

of points and its convex hull.

In this section, we shall present two algorithms that compute the convex hull

of a set of n points. Both algorithms output the vertices of the convex hull in

counterclockwise order. The ﬁrst, known as Graham’s scan, runs in O.n lg n/ time.

The second, called Jarvis’s march, runs in O.nh/ time, where h is the number of

vertices of the convex hull. As Figure 33.6 illustrates, every vertex of CH.Q/ is a

p

0

p

1

p

2

p

3

p

4

p

5

p

6 p

7

p

8

p

9

p

10

p

11

p

12

Figure 33.6 A set of points Q D fp0; p1; : : : ; p12g with its convex hull CH.Q/ in gray.

1030 Chapter 33 Computational Geometry

point in Q. Both algorithms exploit this property, deciding which vertices in Q to

keep as vertices of the convex hull and which vertices in Q to reject.

We can compute convex hulls in O.n lg n/ time by any one of several methods.

Both Graham’s scan and Jarvis’s march use a technique called “rotational sweep,”

processing vertices in the order of the polar angles they form with a reference

vertex. Other methods include the following:

In the incremental method, we ﬁrst sort the points from left to right, yielding a

sequence hp 1 ; p 2 ; : : : ; p n i. At the ith stage, we update the convex hull of the

i 1 leftmost points, CH.fp 1 ; p 2 ; : : : ; p i1 g/, according to the ith point from

the left, thus forming CH.fp 1 ; p 2 ; : : : ; p i g/. Exercise 33.3-6 asks you how to

implement this method to take a total of O.n lg n/ time.

In the divide-and-conquer method, we divide the set of n points in ‚.n/ time

into two subsets, one containing the leftmost dn=2e points and one containing

the rightmost bn=2c points, recursively compute the convex hulls of the subsets,

and then, by means of a clever method, combine the hulls in O.n/ time. The

running time is described by the familiar recurrence T .n/ D 2T .n=2/ C O.n/,

and so the divide-and-conquer method runs in O.n lg n/ time.

The prune-and-search method is similar to the worst-case linear-time median

algorithm of Section 9.3. With this method, we ﬁnd the upper portion (or “upper

chain”) of the convex hull by repeatedly throwing out a constant fraction of the

remaining points until only the upper chain of the convex hull remains. We then

do the same for the lower chain. This method is asymptotically the fastest: if

the convex hull contains h vertices, it runs in only O.n lg h/ time.

Computing the convex hull of a set of points is an interesting problem in its own

right. Moreover, algorithms for some other computational-geometry problems start

by computing a convex hull. Consider, for example, the two-dimensional farthest-

pair problem: we are given a set of n points in the plane and wish to ﬁnd the

two points whose distance from each other is maximum. As Exercise 33.3-3 asks

you to prove, these two points must be vertices of the convex hull. Although we

won’t prove it here, we can ﬁnd the farthest pair of vertices of an n-vertex convex

polygon in O.n/ time. Thus, by computing the convex hull of the n input points

in O.n lg n/ time and then ﬁnding the farthest pair of the resulting convex-polygon

vertices, we can ﬁnd the farthest pair of points in any set of n points in O.n lg n/

time.

Graham’s scan

Graham’s scan solves the convex-hull problem by maintaining a stack S of can-

didate points. It pushes each point of the input set Q onto the stack one time,

33.3 Finding the convex hull 1031

and it eventually pops from the stack each point that is not a vertex of CH.Q/.

When the algorithm terminates, stack S contains exactly the vertices of CH.Q/, in

counterclockwise order of their appearance on the boundary.

The procedure GRAHAM-SCAN takes as input a set Q of points, where jQj 3.

It calls the functions TOP.S/, which returns the point on top of stack S without

changing S, and NEXT-TO-TOP.S/, which returns the point one entry below the

top of stack S without changing S. As we shall prove in a moment, the stack S

returned by GRAHAM-SCAN contains, from bottom to top, exactly the vertices

of CH.Q/ in counterclockwise order.

GRAHAM-SCAN.Q/

1 let p 0 be the point in Q with the minimum y-coordinate,

or the leftmost such point in case of a tie

2 let hp 1 ; p 2 ; : : : ; p m i be the remaining points in Q,

sorted by polar angle in counterclockwise order around p 0

(if more than one point has the same angle, remove all but

the one that is farthest from p 0 )

3 let S be an empty stack

4 PUSH.p 0 ; S/

5 PUSH.p 1 ; S/

6 PUSH.p 2 ; S/

7 for i D 3 to m

8 while the angle formed by points NEXT-TO-TOP.S/, TOP.S/,

and p i makes a nonleft turn

9 POP.S/

10 PUSH.p i ; S/

11 return S

Figure 33.7 illustrates the progress of GRAHAM-SCAN. Line 1 chooses point p 0

as the point with the lowest y-coordinate, picking the leftmost such point in case

of a tie. Since there is no point in Q that is below p 0 and any other points with

the same y-coordinate are to its right, p 0 must be a vertex of CH.Q/. Line 2

sorts the remaining points of Q by polar angle relative to p 0 , using the same

method—comparing cross products—as in Exercise 33.1-3. If two or more points

have the same polar angle relative to p 0 , all but the farthest such point are convex

combinations of p 0 and the farthest point, and so we remove them entirely from

consideration. We let m denote the number of points other than p 0 that remain.

The polar angle, measured in radians, of each point in Q relative to p 0 is in the

half-open interval Œ0; /. Since the points are sorted according to polar angles,

they are sorted in counterclockwise order relative to p 0 . We designate this sorted

sequence of points by hp 1 ; p 2 ; : : : ; p m i. Note that points p 1 and p m are vertices

1032 Chapter 33 Computational Geometry

p

12

p

11

p

10

p

9

p

8

p

7

p

6

p

5

p

4

p

3

p

2

p

1

p

0 (a)

p

12

p

11

p

10

p

9

p

8

p

7

p

6

p

5

p

4

p

3

p

2

p

1

p

0 (b)

p

12

p

11

p

10

p

9

p

8

p

7

p

6

p

5

p

4

p

3

p

2

p

1

p

0 (c)

p

12

p

11

p

10

p

9

p

8

p

7

p

6

p

5

p

4

p

3

p

2

p

1

p

0 (d)

p

12

p

11

p

10

p

9

p

8

p

7

p

6

p

5

p

4

p

3

p

2

p

1

p

0 (e)

p

12

p

11

p

10

p

9

p

8

p

7

p

6

p

5

p

4

p

3

p

2

p

1

p

0 (f)

Figure 33.7 The execution of GRAHAM-SCAN on the set Q of Figure 33.6. The current convex

hull contained in stack S is shown in gray at each step. (a) The sequence hp1; p2; : : : ; p12i of points

numbered in order of increasing polar angle relative to p0, and the initial stack S containing p0, p1,

and p2. (b)–(k) Stack S after each iteration of the for loop of lines 7–10. Dashed lines show nonleft

turns, which cause points to be popped from the stack. In part (h), for example, the right turn at

angle †p7p8p9 causes p8 to be popped, and then the right turn at angle †p6p7p9 causes p7 to be

popped.

33.3 Finding the convex hull 1033

p

12

p

11

p

10

p

9

p

8

p

7

p

6

p

5

p

4

p

3

p

2

p

1

p

0 (g)

p

12

p

11

p

10

p

9

p

8

p

6

p

5

p

4

p

3

p

2

p

1

p

0 (h)

p

12

p

11

p

10

p

9

p

8

p

7

p

6 p

5

p

3

p

2

p

1

p

0 (i)

p

12

p

11

p

10

p

9

p

8

p

7

p

6

p

3

p

2

p

1

p

0 (j)

p

12

p

11

p

10

p

9

p

8

p

7

p

6

p

5

p

4

p

3

p

2

p

1

p

0 (k)

p

12

p

11

p

10

p

9

p

8

p

7

p

6

p

5

p

4

p

3

p

2

p

1

p

0 (l)

p

4

p

4

p

5

p

7

Figure 33.7, continued (l) The convex hull returned by the procedure, which matches that of

Figure 33.6.

1034 Chapter 33 Computational Geometry

of CH.Q/ (see Exercise 33.3-1). Figure 33.7(a) shows the points of Figure 33.6

sequentially numbered in order of increasing polar angle relative to p 0 .

The remainder of the procedure uses the stack S. Lines 3–6 initialize the stack

to contain, from bottom to top, the ﬁrst three points p 0 , p 1 , and p 2 . Figure 33.7(a)

shows the initial stack S. The for loop of lines 7–10 iterates once for each point

in the subsequence hp 3 ; p 4 ; : : : ; p m i. We shall see that after processing point p i ,

stack S contains, from bottom to top, the vertices of CH.fp 0 ; p 1 ; : : : ; p i g/ in coun-

terclockwise order. The while loop of lines 8–9 removes points from the stack if

we ﬁnd them not to be vertices of the convex hull. When we traverse the convex

hull counterclockwise, we should make a left turn at each vertex. Thus, each time

the while loop ﬁnds a vertex at which we make a nonleft turn, we pop the vertex

from the stack. (By checking for a nonleft turn, rather than just a right turn, this

test precludes the possibility of a straight angle at a vertex of the resulting convex

hull. We want no straight angles, since no vertex of a convex polygon may be a

convex combination of other vertices of the polygon.) After we pop all vertices

that have nonleft turns when heading toward point p i , we push p i onto the stack.

Figures 33.7(b)–(k) show the state of the stack S after each iteration of the for

loop. Finally, GRAHAM-SCAN returns the stack S in line 11. Figure 33.7(l) shows

the corresponding convex hull.

The following theorem formally proves the correctness of GRAHAM-SCAN.

Theorem 33.1 (Correctness of Graham’s scan)

If GRAHAM-SCAN executes on a set Q of points, where jQj 3, then at termina-

tion, the stack S consists of, from bottom to top, exactly the vertices of CH.Q/ in

counterclockwise order.

Proof After line 2, we have the sequence of points hp 1 ; p 2 ; : : : ; p m i. Let us

deﬁne, for i D 2; 3; : : : ; m, the subset of points Q i D fp 0 ; p 1 ; : : : ; p i g. The

points in Q Q m are those that were removed because they had the same polar

angle relative to p 0 as some point in Q m ; these points are not in CH.Q/, and

so CH.Q m / D CH.Q/. Thus, it sufﬁces to show that when GRAHAM-SCAN

terminates, the stack S consists of the vertices of CH.Q m / in counterclockwise

order, when listed from bottom to top. Note that just as p 0 , p 1 , and p m are vertices

of CH.Q/, the points p 0 , p 1 , and p i are all vertices of CH.Q i /.

The proof uses the following loop invariant:

At the start of each iteration of the for loop of lines 7–10, stack S consists of,

from bottom to top, exactly the vertices of CH.Q i1 / in counterclockwise

order.

Initialization: The invariant holds the ﬁrst time we execute line 7, since at that

time, stack S consists of exactly the vertices of Q 2 D Q i1 , and this set of three

33.3 Finding the convex hull 1035

p

0

p

1

p

2

p

k

p

j

p

i

Q

j

(a)

p

0

p

1

p

j

p

i

(b)

p

t

p

r

Figure 33.8 The proof of correctness of GRAHAM-SCAN. (a) Because pi ’s polar angle relative

to p0 is greater than pj ’s polar angle, and because the angle †pkpj pi makes a left turn, adding pi

to CH.Qj / gives exactly the vertices of CH.Qj [ fpi g/. (b) If the angle †pr pt pi makes a nonleft

turn, then pt is either in the interior of the triangle formed by p0, pr , and pi or on a side of the

triangle, which means that it cannot be a vertex of CH.Qi /.

vertices forms its own convex hull. Moreover, they appear in counterclockwise

order from bottom to top.

Maintenance: Entering an iteration of the for loop, the top point on stack S

is p i1 , which was pushed at the end of the previous iteration (or before the

ﬁrst iteration, when i D 3). Let p j be the top point on S after executing the

while loop of lines 8–9 but before line 10 pushes p i , and let p k be the point

just below p j on S. At the moment that p j is the top point on S and we have

not yet pushed p i , stack S contains exactly the same points it contained after

iteration j of the for loop. By the loop invariant, therefore, S contains exactly

the vertices of CH.Q j / at that moment, and they appear in counterclockwise

order from bottom to top.

Let us continue to focus on this moment just before pushing p i . We know

that p i ’s polar angle relative to p 0 is greater than p j ’s polar angle and that

the angle †p k p j p i makes a left turn (otherwise we would have popped p j ).

Therefore, because S contains exactly the vertices of CH.Q j /, we see from

Figure 33.8(a) that once we push p i , stack S will contain exactly the vertices

of CH.Q j [ fp i g/, still in counterclockwise order from bottom to top.

We now show that CH.Q j [fp i g/ is the same set of points as CH.Q i /. Consider

any point p t that was popped during iteration i of the for loop, and let p r be

the point just below p t on stack S at the time p t was popped (p r might be p j ).

The angle †p r p t p i makes a nonleft turn, and the polar angle of p t relative

to p 0 is greater than the polar angle of p r . As Figure 33.8(b) shows, p t must

1036 Chapter 33 Computational Geometry

be either in the interior of the triangle formed by p 0 , p r , and p i or on a side of

this triangle (but it is not a vertex of the triangle). Clearly, since p t is within a

triangle formed by three other points of Q i , it cannot be a vertex of CH.Q i /.

Since p t is not a vertex of CH.Q i /, we have that

CH.Q i fp t g/ D CH.Q i / : (33.1)

Let P i be the set of points that were popped during iteration i of the for loop.

Since the equality (33.1) applies for all points in P i , we can apply it repeatedly

to show that CH.Q i P i / D CH.Q i /. But Q i P i D Q j [ fp i g, and so we

conclude that CH.Q j [ fp i g/ D CH.Q i P i / D CH.Q i /.

We have shown that once we push p i , stack S contains exactly the vertices

of CH.Q i / in counterclockwise order from bottom to top. Incrementing i will

then cause the loop invariant to hold for the next iteration.

Termination: When the loop terminates, we have i D m C 1, and so the loop

invariant implies that stack S consists of exactly the vertices of CH.Q m /, which

is CH.Q/, in counterclockwise order from bottom to top. This completes the

proof.

We now show that the running time of GRAHAM-SCAN is O.n lg n/, where

n D jQj. Line 1 takes ‚.n/ time. Line 2 takes O.n lg n/ time, using merge sort

or heapsort to sort the polar angles and the cross-product method of Section 33.1

to compare angles. (We can remove all but the farthest point with the same polar

angle in total of O.n/ time over all n points.) Lines 3–6 take O.1/ time. Because

m n 1, the for loop of lines 7–10 executes at most n 3 times. Since PUSH

takes O.1/ time, each iteration takes O.1/ time exclusive of the time spent in the

while loop of lines 8–9, and thus overall the for loop takes O.n/ time exclusive of

the nested while loop.

We use aggregate analysis to show that the while loop takes O.n/ time overall.

For i D 0; 1; : : : ; m, we push each point p i onto stack S exactly once. As in the

analysis of the MULTIPOP procedure of Section 17.1, we observe that we can pop at

most the number of items that we push. At least three points—p 0 , p 1 , and p m —are

never popped from the stack, so that in fact at most m 2 POP operations are

performed in total. Each iteration of the while loop performs one POP, and so

there are at most m 2 iterations of the while loop altogether. Since the test in

line 8 takes O.1/ time, each call of POP takes O.1/ time, and m n 1, the total

time taken by the while loop is O.n/. Thus, the running time of GRAHAM-SCAN

is O.n lg n/.

33.3 Finding the convex hull 1037

p

4

p

2

p

0

p

1

right chain left chain

right chain left chain

p

3

Figure 33.9 The operation of Jarvis’s march. We choose the ﬁrst vertex as the lowest point p0.

The next vertex, p1, has the smallest polar angle of any point with respect to p0. Then, p2 has the

smallest polar angle with respect to p1. The right chain goes as high as the highest point p3. Then,

we construct the left chain by ﬁnding smallest polar angles with respect to the negative x-axis.

Jarvis’s march

Jarvis’s march computes the convex hull of a set Q of points by a technique known

as package wrapping (or gift wrapping). The algorithm runs in time O.nh/,

where h is the number of vertices of CH.Q/. When h is o.lg n/, Jarvis’s march is

asymptotically faster than Graham’s scan.

Intuitively, Jarvis’s march simulates wrapping a taut piece of paper around the

set Q. We start by taping the end of the paper to the lowest point in the set, that is,

to the same point p 0 with which we start Graham’s scan. We know that this point

must be a vertex of the convex hull. We pull the paper to the right to make it taut,

and then we pull it higher until it touches a point. This point must also be a vertex

of the convex hull. Keeping the paper taut, we continue in this way around the set

of vertices until we come back to our original point p 0 .

More formally, Jarvis’s march builds a sequence H D hp 0 ; p 1 ; : : : ; p h1 i of the

vertices of CH.Q/. We start with p 0 . As Figure 33.9 shows, the next vertex p 1

in the convex hull has the smallest polar angle with respect to p 0 . (In case of ties,

we choose the point farthest from p 0 .) Similarly, p 2 has the smallest polar angle

1038 Chapter 33 Computational Geometry

with respect to p 1 , and so on. When we reach the highest vertex, say p k (breaking

ties by choosing the farthest such vertex), we have constructed, as Figure 33.9

shows, the right chain of CH.Q/. To construct the left chain, we start at p k and

choose p kC1 as the point with the smallest polar angle with respect to p k , but from

the negative x-axis. We continue on, forming the left chain by taking polar angles

from the negative x-axis, until we come back to our original vertex p 0 .

We could implement Jarvis’s march in one conceptual sweep around the convex

hull, that is, without separately constructing the right and left chains. Such imple-

mentations typically keep track of the angle of the last convex-hull side chosen and

require the sequence of angles of hull sides to be strictly increasing (in the range

of 0 to 2 radians). The advantage of constructing separate chains is that we need

not explicitly compute angles; the techniques of Section 33.1 sufﬁce to compare

angles.

If implemented properly, Jarvis’s march has a running time of O.nh/. For each

of the h vertices of CH.Q/, we ﬁnd the vertex with the minimum polar angle. Each

comparison between polar angles takes O.1/ time, using the techniques of Sec-

tion 33.1. As Section 9.1 shows, we can compute the minimum of n values in O.n/

time if each comparison takes O.1/ time. Thus, Jarvis’s march takes O.nh/ time.

Exercises

33.3-1

Prove that in the procedure GRAHAM-SCAN, points p 1 and p m must be vertices

of CH.Q/.

33.3-2

Consider a model of computation that supports addition, comparison, and multipli-

cation and for which there is a lower bound of .n lg n/ to sort n numbers. Prove

that .n lg n/ is a lower bound for computing, in order, the vertices of the convex

hull of a set of n points in such a model.

33.3-3

Given a set of points Q, prove that the pair of points farthest from each other must

be vertices of CH.Q/.

33.3-4

For a given polygon P and a point q on its boundary, the shadow of q is the set

of points r such that the segment qr is entirely on the boundary or in the interior

of P . As Figure 33.10 illustrates, a polygon P is star-shaped if there exists a

point p in the interior of P that is in the shadow of every point on the boundary

of P . The set of all such points p is called the kernel of P . Given an n-vertex,

33.4 Finding the closest pair of points 1039

p

(a) (b)

q

q ′

Figure 33.10 The deﬁnition of a star-shaped polygon, for use in Exercise 33.3-4. (a) A star-shaped

polygon. The segment from point p to any point q on the boundary intersects the boundary only at q.

(b) A non-star-shaped polygon. The shaded region on the left is the shadow of q, and the shaded

region on the right is the shadow of q

0

. Since these regions are disjoint, the kernel is empty.

star-shaped polygon P speciﬁed by its vertices in counterclockwise order, show

how to compute CH.P / in O.n/ time.

33.3-5

In the on-line convex-hull problem, we are given the set Q of n points one point at

a time. After receiving each point, we compute the convex hull of the points seen

so far. Obviously, we could run Graham’s scan once for each point, with a total

running time of O.n 2

lg n/. Show how to solve the on-line convex-hull problem in

a total of O.n 2 / time.

33.3-6 ?

Show how to implement the incremental method for computing the convex hull

of n points so that it runs in O.n lg n/ time.

33.4 Finding the closest pair of points

We now consider the problem of ﬁnding the closest pair of points in a set Q of

n 2 points. “Closest” refers to the usual euclidean distance: the distance between

points p 1 D .x 1 ; y 1 / and p 2 D .x 2 ; y 2 / is

p

.x 1 x 2 / 2 C .y 1 y 2 / 2 . Two points

in set Q may be coincident, in which case the distance between them is zero. This

problem has applications in, for example, trafﬁc-control systems. A system for

controlling air or sea trafﬁc might need to identify the two closest vehicles in order

to detect potential collisions.

A brute-force closest-pair algorithm simply looks at all the

n

2

D ‚.n 2 / pairs

of points. In this section, we shall describe a divide-and-conquer algorithm for

1040 Chapter 33 Computational Geometry

this problem, whose running time is described by the familiar recurrence T .n/ D

2T .n=2/ C O.n/. Thus, this algorithm uses only O.n lg n/ time.

The divide-and-conquer algorithm

Each recursive invocation of the algorithm takes as input a subset P Q and

arrays X and Y , each of which contains all the points of the input subset P .

The points in array X are sorted so that their x-coordinates are monotonically

increasing. Similarly, array Y is sorted by monotonically increasing y-coordinate.

Note that in order to attain the O.n lg n/ time bound, we cannot afford to sort

in each recursive call; if we did, the recurrence for the running time would be

T .n/ D 2T .n=2/ C O.n lg n/, whose solution is T .n/ D O.n lg

2

n/. (Use the

version of the master method given in Exercise 4.6-2.) We shall see a little later

how to use “presorting” to maintain this sorted property without actually sorting in

each recursive call.

A given recursive invocation with inputs P , X, and Y ﬁrst checks whether

jP j 3. If so, the invocation simply performs the brute-force method described

above: try all

jP j

2

pairs of points and return the closest pair. If jP j > 3, the

recursive invocation carries out the divide-and-conquer paradigm as follows.

Divide: Find a vertical line l that bisects the point set P into two sets P L and P R

such that jP L j D djP j =2e, jP R j D bjP j =2c, all points in P L are on or to the

left of line l, and all points in P R are on or to the right of l. Divide the array X

into arrays X L and X R , which contain the points of P L and P R respectively,

sorted by monotonically increasing x-coordinate. Similarly, divide the array Y

into arrays Y L and Y R , which contain the points of P L and P R respectively,

sorted by monotonically increasing y-coordinate.

Conquer: Having divided P into P L and P R , make two recursive calls, one to ﬁnd

the closest pair of points in P L and the other to ﬁnd the closest pair of points

in P R . The inputs to the ﬁrst call are the subset P L and arrays X L and Y L ; the

second call receives the inputs P R , X R , and Y R . Let the closest-pair distances

returned for P L and P R be ı L and ı R , respectively, and let ı D min.ı L ; ı R /.

Combine: The closest pair is either the pair with distance ı found by one of the

recursive calls, or it is a pair of points with one point in P L and the other in P R .

The algorithm determines whether there is a pair with one point in P L and the

other point in P R and whose distance is less than ı. Observe that if a pair of

points has distance less than ı, both points of the pair must be within ı units

of line l. Thus, as Figure 33.11(a) shows, they both must reside in the 2ı-wide

vertical strip centered at line l. To ﬁnd such a pair, if one exists, we do the

following:

33.4 Finding the closest pair of points 1041

1. Create an array Y 0

, which is the array Y with all points not in the 2ı-wide

vertical strip removed. The array Y 0

is sorted by y-coordinate, just as Y is.

2. For each point p in the array Y 0

, try to ﬁnd points in Y 0

that are within ı

units of p. As we shall see shortly, only the 7 points in Y 0

that follow p need

be considered. Compute the distance from p to each of these 7 points, and

keep track of the closest-pair distance ı 0

found over all pairs of points in Y 0

.

3. If ı 0 < ı, then the vertical strip does indeed contain a closer pair than the

recursive calls found. Return this pair and its distance ı 0

. Otherwise, return

the closest pair and its distance ı found by the recursive calls.

The above description omits some implementation details that are necessary to

achieve the O.n lg n/ running time. After proving the correctness of the algorithm,

we shall show how to implement the algorithm to achieve the desired time bound.

Correctness

The correctness of this closest-pair algorithm is obvious, except for two aspects.

First, by bottoming out the recursion when jP j 3, we ensure that we never try to

solve a subproblem consisting of only one point. The second aspect is that we need

only check the 7 points following each point p in array Y 0

; we shall now prove this

property.

Suppose that at some level of the recursion, the closest pair of points is p L 2 P L

and p R 2 P R . Thus, the distance ı 0

between p L and p R is strictly less than ı.

Point p L must be on or to the left of line l and less than ı units away. Similarly, p R

is on or to the right of l and less than ı units away. Moreover, p L and p R are

within ı units of each other vertically. Thus, as Figure 33.11(a) shows, p L and p R

are within a ı 2ı rectangle centered at line l. (There may be other points within

this rectangle as well.)

We next show that at most 8 points of P can reside within this ı 2ı rectangle.

Consider the ı ı square forming the left half of this rectangle. Since all points

within P L are at least ı units apart, at most 4 points can reside within this square;

Figure 33.11(b) shows how. Similarly, at most 4 points in P R can reside within

the ı ı square forming the right half of the rectangle. Thus, at most 8 points of P

can reside within the ı 2ı rectangle. (Note that since points on line l may be in

either P L or P R , there may be up to 4 points on l. This limit is achieved if there are

two pairs of coincident points such that each pair consists of one point from P L and

one point from P R , one pair is at the intersection of l and the top of the rectangle,

and the other pair is where l intersects the bottom of the rectangle.)

Having shown that at most 8 points of P can reside within the rectangle, we

can easily see why we need to check only the 7 points following each point in the

array Y 0

. Still assuming that the closest pair is p L and p R , let us assume without

1042 Chapter 33 Computational Geometry

l

p

L

p

R

P

L

P

R

δ

2

δ

(a)

P

R

P

L

(b)

l

coincident points,

one in P

L

,

one in P

R

coincident points,

one in P

L

,

one in P

R

δ δ

δ

Figure 33.11 Key concepts in the proof that the closest-pair algorithm needs to check only 7 points

following each point in the array Y

0

. (a) If pL 2 PL and pR 2 PR are less than ı units apart, they

must reside within a ı 2ı rectangle centered at line l. (b) How 4 points that are pairwise at least ı

units apart can all reside within a ı ı square. On the left are 4 points in PL, and on the right are 4

points in PR. The ı 2ı rectangle can contain 8 points if the points shown on line l are actually

pairs of coincident points with one point in PL and one in PR.

loss of generality that p L precedes p R in array Y 0

. Then, even if p L occurs as early

as possible in Y 0

and p R occurs as late as possible, p R is in one of the 7 positions

following p L . Thus, we have shown the correctness of the closest-pair algorithm.

Implementation and running time

As we have noted, our goal is to have the recurrence for the running time be T .n/ D

2T .n=2/ C O.n/, where T .n/ is the running time for a set of n points. The main

difﬁculty comes from ensuring that the arrays X L , X R , Y L , and Y R , which are

passed to recursive calls, are sorted by the proper coordinate and also that the

array Y 0

is sorted by y-coordinate. (Note that if the array X that is received by a

recursive call is already sorted, then we can easily divide set P into P L and P R in

linear time.)

The key observation is that in each call, we wish to form a sorted subset of a

sorted array. For example, a particular invocation receives the subset P and the

array Y , sorted by y-coordinate. Having partitioned P into P L and P R , it needs to

form the arrays Y L and Y R , which are sorted by y-coordinate, in linear time. We

can view the method as the opposite of the MERGE procedure from merge sort in

33.4 Finding the closest pair of points 1043

Section 2.3.1: we are splitting a sorted array into two sorted arrays. The following

pseudocode gives the idea.

1 let Y L Œ1 : : Y:length and Y R Œ1 : : Y:length be new arrays

2 Y L :length D Y R :length D 0

3 for i D 1 to Y:length

4 if Y Œi 2 P L

5 Y L :length D Y L :length C 1

6 Y L ŒY L :length D Y Œi

7 else Y R :length D Y R :length C 1

8 Y R ŒY R :length D Y Œi

We simply examine the points in array Y in order. If a point Y Œi is in P L , we

append it to the end of array Y L ; otherwise, we append it to the end of array Y R .

Similar pseudocode works for forming arrays X L , X R , and Y 0

.

The only remaining question is how to get the points sorted in the ﬁrst place. We

presort them; that is, we sort them once and for all before the ﬁrst recursive call.

We pass these sorted arrays into the ﬁrst recursive call, and from there we whittle

them down through the recursive calls as necessary. Presorting adds an additional

O.n lg n/ term to the running time, but now each step of the recursion takes linear

time exclusive of the recursive calls. Thus, if we let T .n/ be the running time of

each recursive step and T 0 .n/ be the running time of the entire algorithm, we get

T 0 .n/ D T .n/ C O.n lg n/ and

T .n/ D

(

2T .n=2/ C O.n/ if n > 3 ;

O.1/ if n 3 :

Thus, T .n/ D O.n lg n/ and T 0 .n/ D O.n lg n/.

Exercises

33.4-1

Professor Williams comes up with a scheme that allows the closest-pair algorithm

to check only 5 points following each point in array Y 0

. The idea is always to place

points on line l into set P L . Then, there cannot be pairs of coincident points on

line l with one point in P L and one in P R . Thus, at most 6 points can reside in

the ı 2ı rectangle. What is the ﬂaw in the professor’s scheme?

33.4-2

Show that it actually sufﬁces to check only the points in the 5 array positions fol-

lowing each point in the array Y 0

.

1044 Chapter 33 Computational Geometry

33.4-3

We can deﬁne the distance between two points in ways other than euclidean. In

the plane, the L

m-distance between points p 1 and p 2 is given by the expres-

sion .jx 1 x 2 j

m

C jy 1 y 2 j

m

/

1=m

. Euclidean distance, therefore, is L 2 -distance.

Modify the closest-pair algorithm to use the L 1 -distance, which is also known as

the Manhattan distance.

33.4-4

Given two points p 1 and p 2 in the plane, the L 1 -distance between them is

given by max.jx 1 x 2 j ; jy 1 y 2 j/. Modify the closest-pair algorithm to use the

L 1 -distance.

33.4-5

Suppose that .n/ of the points given to the closest-pair algorithm are covertical.

Show how to determine the sets P L and P R and how to determine whether each

point of Y is in P L or P R so that the running time for the closest-pair algorithm

remains O.n lg n/.

33.4-6

Suggest a change to the closest-pair algorithm that avoids presorting the Y array

but leaves the running time as O.n lg n/. (Hint: Merge sorted arrays Y L and Y R to

form the sorted array Y .)

Problems

33-1 Convex layers

Given a set Q of points in the plane, we deﬁne the convex layers of Q inductively.

The ﬁrst convex layer of Q consists of those points in Q that are vertices of CH.Q/.

For i > 1, deﬁne Q i to consist of the points of Q with all points in convex layers

1; 2; : : : ; i 1 removed. Then, the ith convex layer of Q is CH.Q i / if Q i ¤ ; and

is undeﬁned otherwise.

a. Give an O.n 2 /-time algorithm to ﬁnd the convex layers of a set of n points.

b. Prove that .n lg n/ time is required to compute the convex layers of a set of n

points with any model of computation that requires .n lg n/ time to sort n real

numbers.

Problems for Chapter 33 1045

33-2 Maximal layers

Let Q be a set of n points in the plane. We say that point .x; y/ dominates

point .x 0 ; y 0 / if x x 0

and y y 0

. A point in Q that is dominated by no other

points in Q is said to be maximal. Note that Q may contain many maximal points,

which can be organized into maximal layers as follows. The ﬁrst maximal layer L 1

is the set of maximal points of Q. For i > 1, the ith maximal layer L i is the set of

maximal points in Q

S i1

j D1

L j .

Suppose that Q has k nonempty maximal layers, and let y i be the y-coordinate

of the leftmost point in L i for i D 1; 2; : : : ; k. For now, assume that no two points

in Q have the same x- or y-coordinate.

a. Show that y 1 > y 2 > > y k .

Consider a point .x; y/ that is to the left of any point in Q and for which y is

distinct from the y-coordinate of any point in Q. Let Q 0 D Q [ f.x; y/g.

b. Let j be the minimum index such that y j < y, unless y < y k , in which case

we let j D k C 1. Show that the maximal layers of Q 0

are as follows:

If j k, then the maximal layers of Q 0

are the same as the maximal layers

of Q, except that L j also includes .x; y/ as its new leftmost point.

If j D k C1, then the ﬁrst k maximal layers of Q 0

are the same as for Q, but

in addition, Q 0

has a nonempty .k C 1/st maximal layer: L kC1 D f.x; y/g.

c. Describe an O.n lg n/-time algorithm to compute the maximal layers of a set Q

of n points. (Hint: Move a sweep line from right to left.)

d. Do any difﬁculties arise if we now allow input points to have the same x- or

y-coordinate? Suggest a way to resolve such problems.

33-3 Ghostbusters and ghosts

A group of n Ghostbusters is battling n ghosts. Each Ghostbuster carries a proton

pack, which shoots a stream at a ghost, eradicating it. A stream goes in a straight

line and terminates when it hits the ghost. The Ghostbusters decide upon the fol-

lowing strategy. They will pair off with the ghosts, forming n Ghostbuster-ghost

pairs, and then simultaneously each Ghostbuster will shoot a stream at his cho-

sen ghost. As we all know, it is very dangerous to let streams cross, and so the

Ghostbusters must choose pairings for which no streams will cross.

Assume that the position of each Ghostbuster and each ghost is a ﬁxed point in

the plane and that no three positions are colinear.

a. Argue that there exists a line passing through one Ghostbuster and one ghost

such that the number of Ghostbusters on one side of the line equals the number

of ghosts on the same side. Describe how to ﬁnd such a line in O.n lg n/ time.

1046 Chapter 33 Computational Geometry

b. Give an O.n 2

lg n/-time algorithm to pair Ghostbusters with ghosts in such a

way that no streams cross.

33-4 Picking up sticks

Professor Charon has a set of n sticks, which are piled up in some conﬁguration.

Each stick is speciﬁed by its endpoints, and each endpoint is an ordered triple

giving its .x; y; ´/ coordinates. No stick is vertical. He wishes to pick up all the

sticks, one at a time, subject to the condition that he may pick up a stick only if

there is no other stick on top of it.

a. Give a procedure that takes two sticks a and b and reports whether a is above,

below, or unrelated to b.

b. Describe an efﬁcient algorithm that determines whether it is possible to pick up

all the sticks, and if so, provides a legal order in which to pick them up.

33-5 Sparse-hulled distributions

Consider the problem of computing the convex hull of a set of points in the plane

that have been drawn according to some known random distribution. Sometimes,

the number of points, or size, of the convex hull of n points drawn from such a

distribution has expectation O.n 1 / for some constant > 0. We call such a

distribution sparse-hulled. Sparse-hulled distributions include the following:

Points drawn uniformly from a unit-radius disk. The convex hull has expected

size ‚.n 1=3 /.

Points drawn uniformly from the interior of a convex polygon with k sides, for

any constant k. The convex hull has expected size ‚.lg n/.

Points drawn according to a two-dimensional normal distribution. The convex

hull has expected size ‚.

p

lg n/.

a. Given two convex polygons with n 1 and n 2 vertices respectively, show how to

compute the convex hull of all n 1 Cn 2 points in O.n 1 Cn 2 / time. (The polygons

may overlap.)

b. Show how to compute the convex hull of a set of n points drawn independently

according to a sparse-hulled distribution in O.n/ average-case time. (Hint:

Recursively ﬁnd the convex hulls of the ﬁrst n=2 points and the second n=2

points, and then combine the results.)

Notes for Chapter 33 1047

Chapter notes

This chapter barely scratches the surface of computational-geometry algorithms

and techniques. Books on computational geometry include those by Preparata and

Shamos [282], Edelsbrunner [99], and O’Rourke [269].

Although geometry has been studied since antiquity, the development of algo-

rithms for geometric problems is relatively new. Preparata and Shamos note that

the earliest notion of the complexity of a problem was given by E. Lemoine in 1902.

He was studying euclidean constructions—those using a compass and a ruler—and

devised a set of ﬁve primitives: placing one leg of the compass on a given point,

placing one leg of the compass on a given line, drawing a circle, passing the ruler’s

edge through a given point, and drawing a line. Lemoine was interested in the

number of primitives needed to effect a given construction; he called this amount

the “simplicity” of the construction.

The algorithm of Section 33.2, which determines whether any segments inter-

sect, is due to Shamos and Hoey [313].

The original version of Graham’s scan is given by Graham [150]. The package-

wrapping algorithm is due to Jarvis [189]. Using a decision-tree model of com-

putation, Yao [359] proved a worst-case lower bound of .n lg n/ for the running

time of any convex-hull algorithm. When the number of vertices h of the con-

vex hull is taken into account, the prune-and-search algorithm of Kirkpatrick and

Seidel [206], which takes O.n lg h/ time, is asymptotically optimal.

The O.n lg n/-time divide-and-conquer algorithm for ﬁnding the closest pair of

points is by Shamos and appears in Preparata and Shamos [282]. Preparata and

Shamos also show that the algorithm is asymptotically optimal in a decision-tree

model.

34 NP-Completeness

Almost all the algorithms we have studied thus far have been polynomial-time al-

gorithms: on inputs of size n, their worst-case running time is O.n k / for some con-

stant k. You might wonder whether all problems can be solved in polynomial time.

The answer is no. For example, there are problems, such as Turing’s famous “Halt-

ing Problem,” that cannot be solved by any computer, no matter how much time we

allow. There are also problems that can be solved, but not in time O.n k / for any

constant k. Generally, we think of problems that are solvable by polynomial-time

algorithms as being tractable, or easy, and problems that require superpolynomial

time as being intractable, or hard.

The subject of this chapter, however, is an interesting class of problems, called

the “NP-complete” problems, whose status is unknown. No polynomial-time al-

gorithm has yet been discovered for an NP-complete problem, nor has anyone yet

been able to prove that no polynomial-time algorithm can exist for any one of them.

This so-called P ¤ NP question has been one of the deepest, most perplexing open

research problems in theoretical computer science since it was ﬁrst posed in 1971.

Several NP-complete problems are particularly tantalizing because they seem

on the surface to be similar to problems that we know how to solve in polynomial

time. In each of the following pairs of problems, one is solvable in polynomial

time and the other is NP-complete, but the difference between problems appears to

be slight:

Shortest vs. longest simple paths: In Chapter 24, we saw that even with negative

edge weights, we can ﬁnd shortest paths from a single source in a directed

graph G D .V; E/ in O.VE/ time. Finding a longest simple path between two

vertices is difﬁcult, however. Merely determining whether a graph contains a

simple path with at least a given number of edges is NP-complete.

Euler tour vs. hamiltonian cycle: An Euler tour of a connected, directed graph

G D .V; E/ is a cycle that traverses each edge of G exactly once, although

it is allowed to visit each vertex more than once. By Problem 22-3, we can

determine whether a graph has an Euler tour in only O.E/ time and, in fact,

Chapter 34 NP-Completeness 1049

we can ﬁnd the edges of the Euler tour in O.E/ time. A hamiltonian cycle of

a directed graph G D .V; E/ is a simple cycle that contains each vertex in V .

Determining whether a directed graph has a hamiltonian cycle is NP-complete.

(Later in this chapter, we shall prove that determining whether an undirected

graph has a hamiltonian cycle is NP-complete.)

2-CNF satisﬁability vs. 3-CNF satisﬁability: A boolean formula contains vari-

ables whose values are 0 or 1; boolean connectives such as ^ (AND), \_ (OR),

and : (NOT); and parentheses. A boolean formula is satisﬁable if there exists

some assignment of the values 0 and 1 to its variables that causes it to evaluate

to 1. We shall deﬁne terms more formally later in this chapter, but informally, a

boolean formula is in k-conjunctive normal form, or k-CNF, if it is the AND

of clauses of ORs of exactly k variables or their negations. For example, the

boolean formula .x 1 \_ :x 2 / ^ .:x 1 \_ x 3 / ^ .:x 2 \_ :x 3 / is in 2-CNF. (It has

the satisfying assignment x 1 D 1; x 2 D 0; x 3 D 1.) Although we can deter-

mine in polynomial time whether a 2-CNF formula is satisﬁable, we shall see

later in this chapter that determining whether a 3-CNF formula is satisﬁable is

NP-complete.

NP-completeness and the classes P and NP

Throughout this chapter, we shall refer to three classes of problems: P, NP, and

NPC, the latter class being the NP-complete problems. We describe them infor-

mally here, and we shall deﬁne them more formally later on.

The class P consists of those problems that are solvable in polynomial time.

More speciﬁcally, they are problems that can be solved in time O.n k / for some

constant k, where n is the size of the input to the problem. Most of the problems

examined in previous chapters are in P.

The class NP consists of those problems that are “veriﬁable” in polynomial time.

What do we mean by a problem being veriﬁable? If we were somehow given a

“certiﬁcate” of a solution, then we could verify that the certiﬁcate is correct in time

polynomial in the size of the input to the problem. For example, in the hamiltonian-

cycle problem, given a directed graph G D .V; E/, a certiﬁcate would be a se-

quence h 1 ; 2 ; 3 ; : : : ; jV j i of jV j vertices. We could easily check in polynomial

time that . i ; iC1 / 2 E for i D 1; 2; 3; : : : ; jV j1 and that . jV j ; 1 / 2 E as well.

As another example, for 3-CNF satisﬁability, a certiﬁcate would be an assignment

of values to variables. We could check in polynomial time that this assignment

satisﬁes the boolean formula.

Any problem in P is also in NP, since if a problem is in P then we can solve it

in polynomial time without even being supplied a certiﬁcate. We shall formalize

this notion later in this chapter, but for now we can believe that P NP. The open

question is whether or not P is a proper subset of NP.

1050 Chapter 34 NP-Completeness

Informally, a problem is in the class NPC—and we refer to it as being NP-

complete—if it is in NP and is as “hard” as any problem in NP. We shall formally

deﬁne what it means to be as hard as any problem in NP later in this chapter.

In the meantime, we will state without proof that if any NP-complete problem

can be solved in polynomial time, then every problem in NP has a polynomial-

time algorithm. Most theoretical computer scientists believe that the NP-complete

problems are intractable, since given the wide range of NP-complete problems

that have been studied to date—without anyone having discovered a polynomial-

time solution to any of them—it would be truly astounding if all of them could

be solved in polynomial time. Yet, given the effort devoted thus far to proving

that NP-complete problems are intractable—without a conclusive outcome—we

cannot rule out the possibility that the NP-complete problems are in fact solvable

in polynomial time.

To become a good algorithm designer, you must understand the rudiments of the

theory of NP-completeness. If you can establish a problem as NP-complete, you

provide good evidence for its intractability. As an engineer, you would then do

better to spend your time developing an approximation algorithm (see Chapter 35)

or solving a tractable special case, rather than searching for a fast algorithm that

solves the problem exactly. Moreover, many natural and interesting problems that

on the surface seem no harder than sorting, graph searching, or network ﬂow are

in fact NP-complete. Therefore, you should become familiar with this remarkable

class of problems.

Overview of showing problems to be NP-complete

The techniques we use to show that a particular problem is NP-complete differ

fundamentally from the techniques used throughout most of this book to design

and analyze algorithms. When we demonstrate that a problem is NP-complete,

we are making a statement about how hard it is (or at least how hard we think it

is), rather than about how easy it is. We are not trying to prove the existence of

an efﬁcient algorithm, but instead that no efﬁcient algorithm is likely to exist. In

this way, NP-completeness proofs bear some similarity to the proof in Section 8.1

of an .n lg n/-time lower bound for any comparison sort algorithm; the speciﬁc

techniques used for showing NP-completeness differ from the decision-tree method

used in Section 8.1, however.

We rely on three key concepts in showing a problem to be NP-complete:

Decision problems vs. optimization problems

Many problems of interest are optimization problems, in which each feasible (i.e.,

“legal”) solution has an associated value, and we wish to ﬁnd a feasible solution

with the best value. For example, in a problem that we call SHORTEST-PATH,

Chapter 34 NP-Completeness 1051

we are given an undirected graph G and vertices u and , and we wish to ﬁnd a

path from u to that uses the fewest edges. In other words, SHORTEST-PATH

is the single-pair shortest-path problem in an unweighted, undirected graph. NP-

completeness applies directly not to optimization problems, however, but to deci-

sion problems, in which the answer is simply “yes” or “no” (or, more formally, “1”

or “0”).

Although NP-complete problems are conﬁned to the realm of decision problems,

we can take advantage of a convenient relationship between optimization problems

and decision problems. We usually can cast a given optimization problem as a re-

lated decision problem by imposing a bound on the value to be optimized. For

example, a decision problem related to SHORTEST-PATH is PATH: given a di-

rected graph G, vertices u and , and an integer k, does a path exist from u to

consisting of at most k edges?

The relationship between an optimization problem and its related decision prob-

lem works in our favor when we try to show that the optimization problem is

“hard.” That is because the decision problem is in a sense “easier,” or at least “no

harder.” As a speciﬁc example, we can solve PATH by solving SHORTEST-PATH

and then comparing the number of edges in the shortest path found to the value

of the decision-problem parameter k. In other words, if an optimization prob-

lem is easy, its related decision problem is easy as well. Stated in a way that has

more relevance to NP-completeness, if we can provide evidence that a decision

problem is hard, we also provide evidence that its related optimization problem is

hard. Thus, even though it restricts attention to decision problems, the theory of

NP-completeness often has implications for optimization problems as well.

Reductions

The above notion of showing that one problem is no harder or no easier than an-

other applies even when both problems are decision problems. We take advantage

of this idea in almost every NP-completeness proof, as follows. Let us consider a

decision problem A, which we would like to solve in polynomial time. We call the

input to a particular problem an instance of that problem; for example, in PATH,

an instance would be a particular graph G, particular vertices u and of G, and a

particular integer k. Now suppose that we already know how to solve a different

decision problem B in polynomial time. Finally, suppose that we have a procedure

that transforms any instance ˛ of A into some instance ˇ of B with the following

characteristics:

The transformation takes polynomial time.

The answers are the same. That is, the answer for ˛ is “yes” if and only if the

answer for ˇ is also “yes.”

1052 Chapter 34 NP-Completeness

polynomial-time

reduction algorithm

instance

β

polynomial-time

algorithm to decide B

yes

yes

polynomial-time algorithm to decide A

no no of B

instance

α

of A

Figure 34.1 How to use a polynomial-time reduction algorithm to solve a decision problem A in

polynomial time, given a polynomial-time decision algorithm for another problem B. In polynomial

time, we transform an instance ˛ of A into an instance ˇ of B, we solve B in polynomial time, and

we use the answer for ˇ as the answer for ˛.

We call such a procedure a polynomial-time reduction algorithm and, as Fig-

ure 34.1 shows, it provides us a way to solve problem A in polynomial time:

1. Given an instance ˛ of problem A, use a polynomial-time reduction algorithm

to transform it to an instance ˇ of problem B.

2. Run the polynomial-time decision algorithm for B on the instance ˇ.

3. Use the answer for ˇ as the answer for ˛.

As long as each of these steps takes polynomial time, all three together do also, and

so we have a way to decide on ˛ in polynomial time. In other words, by “reducing”

solving problem A to solving problem B, we use the “easiness” of B to prove the

“easiness” of A.

Recalling that NP-completeness is about showing how hard a problem is rather

than how easy it is, we use polynomial-time reductions in the opposite way to show

that a problem is NP-complete. Let us take the idea a step further, and show how we

could use polynomial-time reductions to show that no polynomial-time algorithm

can exist for a particular problem B. Suppose we have a decision problem A for

which we already know that no polynomial-time algorithm can exist. (Let us not

concern ourselves for now with how to ﬁnd such a problem A.) Suppose further

that we have a polynomial-time reduction transforming instances of A to instances

of B. Now we can use a simple proof by contradiction to show that no polynomial-

time algorithm can exist for B. Suppose otherwise; i.e., suppose that B has a

polynomial-time algorithm. Then, using the method shown in Figure 34.1, we

would have a way to solve problem A in polynomial time, which contradicts our

assumption that there is no polynomial-time algorithm for A.

For NP-completeness, we cannot assume that there is absolutely no polynomial-

time algorithm for problem A. The proof methodology is similar, however, in that

we prove that problem B is NP-complete on the assumption that problem A is also

NP-complete.

34.1 Polynomial time 1053

A ﬁrst NP-complete problem

Because the technique of reduction relies on having a problem already known to

be NP-complete in order to prove a different problem NP-complete, we need a

“ﬁrst” NP-complete problem. The problem we shall use is the circuit-satisﬁability

problem, in which we are given a boolean combinational circuit composed of AND,

OR, and NOT gates, and we wish to know whether there exists some set of boolean

inputs to this circuit that causes its output to be 1. We shall prove that this ﬁrst

problem is NP-complete in Section 34.3.

Chapter outline

This chapter studies the aspects of NP-completeness that bear most directly on the

analysis of algorithms. In Section 34.1, we formalize our notion of “problem” and

deﬁne the complexity class P of polynomial-time solvable decision problems. We

also see how these notions ﬁt into the framework of formal-language theory. Sec-

tion 34.2 deﬁnes the class NP of decision problems whose solutions are veriﬁable

in polynomial time. It also formally poses the P ¤ NP question.

Section 34.3 shows we can relate problems via polynomial-time “reductions.”

It deﬁnes NP-completeness and sketches a proof that one problem, called “circuit

satisﬁability,” is NP-complete. Having found one NP-complete problem, we show

in Section 34.4 how to prove other problems to be NP-complete much more simply

by the methodology of reductions. We illustrate this methodology by showing that

two formula-satisﬁability problems are NP-complete. With additional reductions,

we show in Section 34.5 a variety of other problems to be NP-complete.

34.1 Polynomial time

We begin our study of NP-completeness by formalizing our notion of polynomial-

time solvable problems. We generally regard these problems as tractable, but for

philosophical, not mathematical, reasons. We can offer three supporting argu-

ments.

First, although we may reasonably regard a problem that requires time ‚.n 100 /

to be intractable, very few practical problems require time on the order of such a

high-degree polynomial. The polynomial-time computable problems encountered

in practice typically require much less time. Experience has shown that once the

ﬁrst polynomial-time algorithm for a problem has been discovered, more efﬁcient

algorithms often follow. Even if the current best algorithm for a problem has a

running time of ‚.n 100 /, an algorithm with a much better running time will likely

soon be discovered.

1054 Chapter 34 NP-Completeness

Second, for many reasonable models of computation, a problem that can be

solved in polynomial time in one model can be solved in polynomial time in an-

other. For example, the class of problems solvable in polynomial time by the serial

random-access machine used throughout most of this book is the same as the class

of problems solvable in polynomial time on abstract Turing machines. 1 It is also

the same as the class of problems solvable in polynomial time on a parallel com-

puter when the number of processors grows polynomially with the input size.

Third, the class of polynomial-time solvable problems has nice closure proper-

ties, since polynomials are closed under addition, multiplication, and composition.

For example, if the output of one polynomial-time algorithm is fed into the input of

another, the composite algorithm is polynomial. Exercise 34.1-5 asks you to show

that if an algorithm makes a constant number of calls to polynomial-time subrou-

tines and performs an additional amount of work that also takes polynomial time,

then the running time of the composite algorithm is polynomial.

Abstract problems

To understand the class of polynomial-time solvable problems, we must ﬁrst have

a formal notion of what a “problem” is. We deﬁne an abstract problem Q to be a

binary relation on a set I of problem instances and a set S of problem solutions.

For example, an instance for SHORTEST-PATH is a triple consisting of a graph

and two vertices. A solution is a sequence of vertices in the graph, with perhaps

the empty sequence denoting that no path exists. The problem SHORTEST-PATH

itself is the relation that associates each instance of a graph and two vertices with

a shortest path in the graph that connects the two vertices. Since shortest paths are

not necessarily unique, a given problem instance may have more than one solution.

This formulation of an abstract problem is more general than we need for our

purposes. As we saw above, the theory of NP-completeness restricts attention to

decision problems: those having a yes/no solution. In this case, we can view an

abstract decision problem as a function that maps the instance set I to the solution

set f0; 1g. For example, a decision problem related to SHORTEST-PATH is the

problem PATH that we saw earlier. If i D hG;u;;ki is an instance of the decision

problem PATH, then PATH.i/ D 1 (yes) if a shortest path from u to has at

most k edges, and PATH.i/ D 0 (no) otherwise. Many abstract problems are not

decision problems, but rather optimization problems, which require some value to

be minimized or maximized. As we saw above, however, we can usually recast an

optimization problem as a decision problem that is no harder.

1

See Hopcroft and Ullman [180] or Lewis and Papadimitriou [236] for a thorough treatment of the

Turing-machine model.

34.1 Polynomial time 1055

Encodings

In order for a computer program to solve an abstract problem, we must represent

problem instances in a way that the program understands. An encoding of a set S

of abstract objects is a mapping e from S to the set of binary strings. 2 For example,

we are all familiar with encoding the natural numbers N D f0; 1; 2; 3; 4; : : :g as

the strings f0; 1; 10; 11; 100; : : :g. Using this encoding, e.17/ D 10001. If you

have looked at computer representations of keyboard characters, you probably have

seen the ASCII code, where, for example, the encoding of A is 1000001. We can

encode a compound object as a binary string by combining the representations of

its constituent parts. Polygons, graphs, functions, ordered pairs, programs—all can

be encoded as binary strings.

Thus, a computer algorithm that “solves” some abstract decision problem actu-

ally takes an encoding of a problem instance as input. We call a problem whose

instance set is the set of binary strings a concrete problem. We say that an algo-

rithm solves a concrete problem in time O.T .n// if, when it is provided a problem

instance i of length n D jij, the algorithm can produce the solution in O.T .n//

time. 3 A concrete problem is polynomial-time solvable, therefore, if there exists

an algorithm to solve it in time O.n k / for some constant k.

We can now formally deﬁne the complexity class P as the set of concrete deci-

sion problems that are polynomial-time solvable.

We can use encodings to map abstract problems to concrete problems. Given

an abstract decision problem Q mapping an instance set I to f0; 1g, an encoding

e W I ! f0; 1g

can induce a related concrete decision problem, which we denote

by e.Q/. 4 If the solution to an abstract-problem instance i 2 I is Q.i/ 2 f0; 1g,

then the solution to the concrete-problem instance e.i/ 2 f0; 1g

is also Q.i/. As a

technicality, some binary strings might represent no meaningful abstract-problem

instance. For convenience, we shall assume that any such string maps arbitrarily

to 0. Thus, the concrete problem produces the same solutions as the abstract prob-

lem on binary-string instances that represent the encodings of abstract-problem

instances.

We would like to extend the deﬁnition of polynomial-time solvability from con-

crete problems to abstract problems by using encodings as the bridge, but we would

2

The codomain of e need not be binary strings; any set of strings over a ﬁnite alphabet having at

least 2 symbols will do.

3

We assume that the algorithm’s output is separate from its input. Because it takes at least one time

step to produce each bit of the output and the algorithm takes O.T .n// time steps, the size of the

output is O.T .n//.

4

We denote by f0; 1g

the set of all strings composed of symbols from the set f0; 1g.

1056 Chapter 34 NP-Completeness

like the deﬁnition to be independent of any particular encoding. That is, the ef-

ﬁciency of solving a problem should not depend on how the problem is encoded.

Unfortunately, it depends quite heavily on the encoding. For example, suppose that

an integer k is to be provided as the sole input to an algorithm, and suppose that

the running time of the algorithm is ‚.k/. If the integer k is provided in unary—a

string of k 1s—then the running time of the algorithm is O.n/ on length-n inputs,

which is polynomial time. If we use the more natural binary representation of the

integer k, however, then the input length is n D blg kc C 1. In this case, the run-

ning time of the algorithm is ‚.k/ D ‚.2 n /, which is exponential in the size of the

input. Thus, depending on the encoding, the algorithm runs in either polynomial

or superpolynomial time.

How we encode an abstract problem matters quite a bit to how we understand

polynomial time. We cannot really talk about solving an abstract problem without

ﬁrst specifying an encoding. Nevertheless, in practice, if we rule out “expensive”

encodings such as unary ones, the actual encoding of a problem makes little dif-

ference to whether the problem can be solved in polynomial time. For example,

representing integers in base 3 instead of binary has no effect on whether a prob-

lem is solvable in polynomial time, since we can convert an integer represented in

base 3 to an integer represented in base 2 in polynomial time.

We say that a function f W f0; 1g

! f0; 1g

is polynomial-time computable

if there exists a polynomial-time algorithm A that, given any input x 2 f0; 1g

,

produces as output f .x/. For some set I of problem instances, we say that two en-

codings e 1 and e 2 are polynomially related if there exist two polynomial-time com-

putable functions f 12 and f 21 such that for any i 2 I, we have f 12 .e 1 .i// D e 2 .i/

and f 21 .e 2 .i// D e 1 .i/. 5 That is, a polynomial-time algorithm can compute the en-

coding e 2 .i/ from the encoding e 1 .i/, and vice versa. If two encodings e 1 and e 2 of

an abstract problem are polynomially related, whether the problem is polynomial-

time solvable or not is independent of which encoding we use, as the following

lemma shows.

Lemma 34.1

Let Q be an abstract decision problem on an instance set I, and let e 1 and e 2 be

polynomially related encodings on I. Then, e 1 .Q/ 2 P if and only if e 2 .Q/ 2 P.

5

Technically, we also require the functions f12 and f21 to “map noninstances to noninstances.”

A noninstance of an encoding e is a string x 2 f0; 1g

such that there is no instance i for which

e.i/ D x. We require that f12.x/ D y for every noninstance x of encoding e1, where y is some non-

instance of e2, and that f21.x

0

/ D y

0

for every noninstance x

0

of e2, where y

0

is some noninstance

of e1.

34.1 Polynomial time 1057

Proof We need only prove the forward direction, since the backward direction is

symmetric. Suppose, therefore, that e 1 .Q/ can be solved in time O.n k / for some

constant k. Further, suppose that for any problem instance i, the encoding e 1 .i/

can be computed from the encoding e 2 .i/ in time O.n c / for some constant c, where

n D je 2 .i/j. To solve problem e 2 .Q/, on input e 2 .i/, we ﬁrst compute e 1 .i/ and

then run the algorithm for e 1 .Q/ on e 1 .i/. How long does this take? Converting

encodings takes time O.n c /, and therefore je 1 .i/j D O.n c /, since the output of

a serial computer cannot be longer than its running time. Solving the problem

on e 1 .i/ takes time O.je 1 .i/j

k

/ D O.n ck /, which is polynomial since both c and k

are constants.

Thus, whether an abstract problem has its instances encoded in binary or base 3

does not affect its “complexity,” that is, whether it is polynomial-time solvable or

not; but if instances are encoded in unary, its complexity may change. In order to

be able to converse in an encoding-independent fashion, we shall generally assume

that problem instances are encoded in any reasonable, concise fashion, unless we

speciﬁcally say otherwise. To be precise, we shall assume that the encoding of an

integer is polynomially related to its binary representation, and that the encoding of

a ﬁnite set is polynomially related to its encoding as a list of its elements, enclosed

in braces and separated by commas. (ASCII is one such encoding scheme.) With

such a “standard” encoding in hand, we can derive reasonable encodings of other

mathematical objects, such as tuples, graphs, and formulas. To denote the standard

encoding of an object, we shall enclose the object in angle braces. Thus, hGi

denotes the standard encoding of a graph G.

As long as we implicitly use an encoding that is polynomially related to this

standard encoding, we can talk directly about abstract problems without reference

to any particular encoding, knowing that the choice of encoding has no effect on

whether the abstract problem is polynomial-time solvable. Henceforth, we shall

generally assume that all problem instances are binary strings encoded using the

standard encoding, unless we explicitly specify the contrary. We shall also typically

neglect the distinction between abstract and concrete problems. You should watch

out for problems that arise in practice, however, in which a standard encoding is

not obvious and the encoding does make a difference.

A formal-language framework

By focusing on decision problems, we can take advantage of the machinery of

formal-language theory. Let’s review some deﬁnitions from that theory. An

alphabet † is a ﬁnite set of symbols. A language L over † is any set of

strings made up of symbols from †. For example, if † D f0; 1g, the set

L D f10; 11; 101; 111; 1011; 1101; 10001; : : :g is the language of binary represen-

1058 Chapter 34 NP-Completeness

tations of prime numbers. We denote the empty string by ", the empty language

by ;, and the language of all strings over † by †

. For example, if † D f0; 1g,

then † D f"; 0; 1; 00; 01; 10; 11; 000; : : :g is the set of all binary strings. Every

language L over † is a subset of †

.

We can perform a variety of operations on languages. Set-theoretic operations,

such as union and intersection, follow directly from the set-theoretic deﬁnitions.

We deﬁne the complement of L by L D † L. The concatenation L 1 L 2 of two

languages L 1 and L 2 is the language

L D fx 1 x 2 W x 1 2 L 1 and x 2 2 L 2 g :

The closure or Kleene star of a language L is the language

L

D f"g [ L [ L

2

[ L

3

[ ;

where L k

is the language obtained by concatenating L to itself k times.

From the point of view of language theory, the set of instances for any decision

problem Q is simply the set †

, where † D f0; 1g. Since Q is entirely character-

ized by those problem instances that produce a 1 (yes) answer, we can view Q as

a language L over † D f0; 1g, where

L D fx 2 †

W Q.x/ D 1g :

For example, the decision problem PATH has the corresponding language

PATH D fhG; u; ; ki W G D .V; E/ is an undirected graph,

u; 2 V;

k 0 is an integer, and

there exists a path from u to in G

consisting of at most k edgesg :

(Where convenient, we shall sometimes use the same name—PATH in this case—

to refer to both a decision problem and its corresponding language.)

The formal-language framework allows us to express concisely the relation be-

tween decision problems and algorithms that solve them. We say that an al-

gorithm A accepts a string x 2 f0; 1g

if, given input x, the algorithm’s out-

put A.x/ is 1. The language accepted by an algorithm A is the set of strings

L D fx 2 f0; 1g

W A.x/ D 1g, that is, the set of strings that the algorithm accepts.

An algorithm A rejects a string x if A.x/ D 0.

Even if language L is accepted by an algorithm A, the algorithm will not neces-

sarily reject a string x 62 L provided as input to it. For example, the algorithm may

loop forever. A language L is decided by an algorithm A if every binary string

in L is accepted by A and every binary string not in L is rejected by A. A lan-

guage L is accepted in polynomial time by an algorithm A if it is accepted by A

and if in addition there exists a constant k such that for any length-n string x 2 L,

34.1 Polynomial time 1059

algorithm A accepts x in time O.n k /. A language L is decided in polynomial

time by an algorithm A if there exists a constant k such that for any length-n string

x 2 f0; 1g

, the algorithm correctly decides whether x 2 L in time O.n k /. Thus,

to accept a language, an algorithm need only produce an answer when provided a

string in L, but to decide a language, it must correctly accept or reject every string

in f0; 1g

.

As an example, the language PATH can be accepted in polynomial time. One

polynomial-time accepting algorithm veriﬁes that G encodes an undirected graph,

veriﬁes that u and are vertices in G, uses breadth-ﬁrst search to compute a short-

est path from u to in G, and then compares the number of edges on the shortest

path obtained with k. If G encodes an undirected graph and the path found from u

to has at most k edges, the algorithm outputs 1 and halts. Otherwise, the algo-

rithm runs forever. This algorithm does not decide PATH, however, since it does

not explicitly output 0 for instances in which a shortest path has more than k edges.

A decision algorithm for PATH must explicitly reject binary strings that do not be-

long to PATH. For a decision problem such as PATH, such a decision algorithm is

easy to design: instead of running forever when there is not a path from u to with

at most k edges, it outputs 0 and halts. (It must also output 0 and halt if the input

encoding is faulty.) For other problems, such as Turing’s Halting Problem, there

exists an accepting algorithm, but no decision algorithm exists.

We can informally deﬁne a complexity class as a set of languages, membership

in which is determined by a complexity measure, such as running time, of an

algorithm that determines whether a given string x belongs to language L. The

actual deﬁnition of a complexity class is somewhat more technical. 6

Using this language-theoretic framework, we can provide an alternative deﬁni-

tion of the complexity class P:

P D fL f0; 1g

W there exists an algorithm A that decides L

in polynomial timeg :

In fact, P is also the class of languages that can be accepted in polynomial time.

Theorem 34.2

P D fL W L is accepted by a polynomial-time algorithmg :

Proof Because the class of languages decided by polynomial-time algorithms is

a subset of the class of languages accepted by polynomial-time algorithms, we

need only show that if L is accepted by a polynomial-time algorithm, it is de-

cided by a polynomial-time algorithm. Let L be the language accepted by some

6

For more on complexity classes, see the seminal paper by Hartmanis and Stearns [162].

1060 Chapter 34 NP-Completeness

polynomial-time algorithm A. We shall use a classic “simulation” argument to

construct another polynomial-time algorithm A 0

that decides L. Because A ac-

cepts L in time O.n k / for some constant k, there also exists a constant c such

that A accepts L in at most cn k

steps. For any input string x, the algorithm A 0

simulates cn k

steps of A. After simulating cn k

steps, algorithm A 0

inspects the be-

havior of A. If A has accepted x, then A 0

accepts x by outputting a 1. If A has not

accepted x, then A 0

rejects x by outputting a 0. The overhead of A 0

simulating A

does not increase the running time by more than a polynomial factor, and thus A 0

is a polynomial-time algorithm that decides L.

Note that the proof of Theorem 34.2 is nonconstructive. For a given language

L 2 P, we may not actually know a bound on the running time for the algorithm A

that accepts L. Nevertheless, we know that such a bound exists, and therefore, that

an algorithm A 0

exists that can check the bound, even though we may not be able

to ﬁnd the algorithm A 0

easily.

Exercises

34.1-1

Deﬁne the optimization problem LONGEST-PATH-LENGTH as the relation that

associates each instance of an undirected graph and two vertices with the num-

ber of edges in a longest simple path between the two vertices. Deﬁne the de-

cision problem LONGEST-PATH D fhG; u; ; ki W G D .V; E/ is an undi-

rected graph, u; 2 V , k 0 is an integer, and there exists a simple path

from u to in G consisting of at least k edgesg. Show that the optimization prob-

lem LONGEST-PATH-LENGTH can be solved in polynomial time if and only if

LONGEST-PATH 2 P.

34.1-2

Give a formal deﬁnition for the problem of ﬁnding the longest simple cycle in an

undirected graph. Give a related decision problem. Give the language correspond-

ing to the decision problem.

34.1-3

Give a formal encoding of directed graphs as binary strings using an adjacency-

matrix representation. Do the same using an adjacency-list representation. Argue

that the two representations are polynomially related.

34.1-4

Is the dynamic-programming algorithm for the 0-1 knapsack problem that is asked

for in Exercise 16.2-2 a polynomial-time algorithm? Explain your answer.

34.2 Polynomial-time veriﬁcation 1061

34.1-5

Show that if an algorithm makes at most a constant number of calls to polynomial-

time subroutines and performs an additional amount of work that also takes polyno-

mial time, then it runs in polynomial time. Also show that a polynomial number of

calls to polynomial-time subroutines may result in an exponential-time algorithm.

34.1-6

Show that the class P, viewed as a set of languages, is closed under union, inter-

section, concatenation, complement, and Kleene star. That is, if L 1 ; L 2 2 P, then

L 1 [ L 2 2 P, L 1 \ L 2 2 P, L 1 L 2 2 P, L 1 2 P, and L

1

2 P.

34.2 Polynomial-time veriﬁcation

We now look at algorithms that verify membership in languages. For example,

suppose that for a given instance hG; u; ; ki of the decision problem PATH, we

are also given a path p from u to . We can easily check whether p is a path in G

and whether the length of p is at most k, and if so, we can view p as a “certiﬁcate”

that the instance indeed belongs to PATH. For the decision problem PATH, this

certiﬁcate doesn’t seem to buy us much. After all, PATH belongs to P—in fact,

we can solve PATH in linear time—and so verifying membership from a given

certiﬁcate takes as long as solving the problem from scratch. We shall now examine

a problem for which we know of no polynomial-time decision algorithm and yet,

given a certiﬁcate, veriﬁcation is easy.

Hamiltonian cycles

The problem of ﬁnding a hamiltonian cycle in an undirected graph has been stud-

ied for over a hundred years. Formally, a hamiltonian cycle of an undirected graph

G D .V; E/ is a simple cycle that contains each vertex in V . A graph that con-

tains a hamiltonian cycle is said to be hamiltonian; otherwise, it is nonhamilto-

nian. The name honors W. R. Hamilton, who described a mathematical game on

the dodecahedron (Figure 34.2(a)) in which one player sticks ﬁve pins in any ﬁve

consecutive vertices and the other player must complete the path to form a cycle

1062 Chapter 34 NP-Completeness

(a) (b)

Figure 34.2 (a) A graph representing the vertices, edges, and faces of a dodecahedron, with a

hamiltonian cycle shown by shaded edges. (b) A bipartite graph with an odd number of vertices.

Any such graph is nonhamiltonian.

containing all the vertices. 7 The dodecahedron is hamiltonian, and Figure 34.2(a)

shows one hamiltonian cycle. Not all graphs are hamiltonian, however. For ex-

ample, Figure 34.2(b) shows a bipartite graph with an odd number of vertices.

Exercise 34.2-2 asks you to show that all such graphs are nonhamiltonian.

We can deﬁne the hamiltonian-cycle problem, “Does a graph G have a hamil-

tonian cycle?” as a formal language:

HAM-CYCLE D fhGi W G is a hamiltonian graphg :

How might an algorithm decide the language HAM-CYCLE? Given a problem

instance hGi, one possible decision algorithm lists all permutations of the vertices

of G and then checks each permutation to see if it is a hamiltonian path. What is

the running time of this algorithm? If we use the “reasonable” encoding of a graph

as its adjacency matrix, the number m of vertices in the graph is .

p

n/, where

n D jhGij is the length of the encoding of G. There are mŠ possible permutations

7

In a letter dated 17 October 1856 to his friend John T. Graves, Hamilton [157, p. 624] wrote, “I

have found that some young persons have been much amused by trying a new mathematical game

which the Icosion furnishes, one person sticking ﬁve pins in any ﬁve consecutive points . . . and the

other player then aiming to insert, which by the theory in this letter can always be done, ﬁfteen other

pins, in cyclical succession, so as to cover all the other points, and to end in immediate proximity to

the pin wherewith his antagonist had begun.”

34.2 Polynomial-time veriﬁcation 1063

of the vertices, and therefore the running time is .mŠ/ D .

p

n Š/ D .2

p

n /,

which is not O.n k / for any constant k. Thus, this naive algorithm does not run

in polynomial time. In fact, the hamiltonian-cycle problem is NP-complete, as we

shall prove in Section 34.5.

Veriﬁcation algorithms

Consider a slightly easier problem. Suppose that a friend tells you that a given

graph G is hamiltonian, and then offers to prove it by giving you the vertices in

order along the hamiltonian cycle. It would certainly be easy enough to verify the

proof: simply verify that the provided cycle is hamiltonian by checking whether

it is a permutation of the vertices of V and whether each of the consecutive edges

along the cycle actually exists in the graph. You could certainly implement this

veriﬁcation algorithm to run in O.n 2 / time, where n is the length of the encoding

of G. Thus, a proof that a hamiltonian cycle exists in a graph can be veriﬁed in

polynomial time.

We deﬁne a veriﬁcation algorithm as being a two-argument algorithm A, where

one argument is an ordinary input string x and the other is a binary string y called

a certiﬁcate. A two-argument algorithm A veriﬁes an input string x if there exists

a certiﬁcate y such that A.x; y/ D 1. The language veriﬁed by a veriﬁcation

algorithm A is

L D fx 2 f0; 1g

W there exists y 2 f0; 1g

such that A.x; y/ D 1g :

Intuitively, an algorithm A veriﬁes a language L if for any string x 2 L, there

exists a certiﬁcate y that A can use to prove that x 2 L. Moreover, for any string

x 62 L, there must be no certiﬁcate proving that x 2 L. For example, in the

hamiltonian-cycle problem, the certiﬁcate is the list of vertices in some hamilto-

nian cycle. If a graph is hamiltonian, the hamiltonian cycle itself offers enough

information to verify this fact. Conversely, if a graph is not hamiltonian, there

can be no list of vertices that fools the veriﬁcation algorithm into believing that the

graph is hamiltonian, since the veriﬁcation algorithm carefully checks the proposed

“cycle” to be sure.

1064 Chapter 34 NP-Completeness

The complexity class NP

The complexity class NP is the class of languages that can be veriﬁed by a poly-

nomial-time algorithm. 8 More precisely, a language L belongs to NP if and only if

there exist a two-input polynomial-time algorithm A and a constant c such that

L D fx 2 f0; 1g

W there exists a certiﬁcate y with jyj D O.jxj

c

/

such that A.x; y/ D 1g :

We say that algorithm A veriﬁes language L in polynomial time.

From our earlier discussion on the hamiltonian-cycle problem, we now see that

HAM-CYCLE 2 NP. (It is always nice to know that an important set is nonempty.)

Moreover, if L 2 P, then L 2 NP, since if there is a polynomial-time algorithm

to decide L, the algorithm can be easily converted to a two-argument veriﬁcation

algorithm that simply ignores any certiﬁcate and accepts exactly those input strings

it determines to be in L. Thus, P NP.

It is unknown whether P D NP, but most researchers believe that P and NP are

not the same class. Intuitively, the class P consists of problems that can be solved

quickly. The class NP consists of problems for which a solution can be veriﬁed

quickly. You may have learned from experience that it is often more difﬁcult to

solve a problem from scratch than to verify a clearly presented solution, especially

when working under time constraints. Theoretical computer scientists generally

believe that this analogy extends to the classes P and NP, and thus that NP includes

languages that are not in P.

There is more compelling, though not conclusive, evidence that P ¤ NP—the

existence of languages that are “NP-complete.” We shall study this class in Sec-

tion 34.3.

Many other fundamental questions beyond the P ¤ NP question remain unre-

solved. Figure 34.3 shows some possible scenarios. Despite much work by many

researchers, no one even knows whether the class NP is closed under comple-

ment. That is, does L 2 NP imply L 2 NP? We can deﬁne the complexity class

co-NP as the set of languages L such that L 2 NP. We can restate the question

of whether NP is closed under complement as whether NP D co-NP. Since P is

closed under complement (Exercise 34.1-6), it follows from Exercise 34.2-9 that

P NP \ co-NP. Once again, however, no one knows whether P D NP \ co-NP

or whether there is some language in NP \ co-NP P.

8

The name “NP” stands for “nondeterministic polynomial time.” The class NP was originally studied

in the context of nondeterminism, but this book uses the somewhat simpler yet equivalent notion of

veriﬁcation. Hopcroft and Ullman [180] give a good presentation of NP-completeness in terms of

nondeterministic models of computation.

34.2 Polynomial-time veriﬁcation 1065

co-NP NP

(c)

P = NP ∩ co-NP

co-NP NP

(d)

P

P = NP = co-NP

(a)

NP = co-NP

(b)

P

NP ∩ co-NP

Figure 34.3 Four possibilities for relationships among complexity classes. In each diagram, one

region enclosing another indicates a proper-subset relation. (a) P D NP D co-NP. Most researchers

regard this possibility as the most unlikely. (b) If NP is closed under complement, then NP D co-NP,

but it need not be the case that P D NP. (c) P D NP\co-NP, but NP is not closed under complement.

(d) NP ¤ co-NP and P ¤ NP \ co-NP. Most researchers regard this possibility as the most likely.

Thus, our understanding of the precise relationship between P and NP is woe-

fully incomplete. Nevertheless, even though we might not be able to prove that a

particular problem is intractable, if we can prove that it is NP-complete, then we

have gained valuable information about it.

Exercises

34.2-1

Consider the language GRAPH-ISOMORPHISM D fhG 1 ; G 2 i W G 1 and G 2 are

isomorphic graphsg. Prove that GRAPH-ISOMORPHISM 2 NP by describing a

polynomial-time algorithm to verify the language.

34.2-2

Prove that if G is an undirected bipartite graph with an odd number of vertices,

then G is nonhamiltonian.

34.2-3

Show that if HAM-CYCLE 2 P, then the problem of listing the vertices of a

hamiltonian cycle, in order, is polynomial-time solvable.

1066 Chapter 34 NP-Completeness

34.2-4

Prove that the class NP of languages is closed under union, intersection, concate-

nation, and Kleene star. Discuss the closure of NP under complement.

34.2-5

Show that any language in NP can be decided by an algorithm running in

time 2 O.n

k

/

for some constant k.

34.2-6

A hamiltonian path in a graph is a simple path that visits every vertex exactly

once. Show that the language HAM-PATH D fhG; u; i W there is a hamiltonian

path from u to in graph Gg belongs to NP.

34.2-7

Show that the hamiltonian-path problem from Exercise 34.2-6 can be solved in

polynomial time on directed acyclic graphs. Give an efﬁcient algorithm for the

problem.

34.2-8

Let be a boolean formula constructed from the boolean input variables x 1 ; x 2 ;

: : : ; x k , negations (:), ANDs (^), ORs (\_), and parentheses. The formula is a

tautology if it evaluates to 1 for every assignment of 1 and 0 to the input variables.

Deﬁne TAUTOLOGY as the language of boolean formulas that are tautologies.

Show that TAUTOLOGY 2 co-NP.

34.2-9

Prove that P co-NP.

34.2-10

Prove that if NP ¤ co-NP, then P ¤ NP.

34.2-11

Let G be a connected, undirected graph with at least 3 vertices, and let G 3

be the

graph obtained by connecting all pairs of vertices that are connected by a path in G

of length at most 3. Prove that G 3

is hamiltonian. (Hint: Construct a spanning tree

for G, and use an inductive argument.)

34.3 NP-completeness and reducibility 1067

34.3 NP-completeness and reducibility

Perhaps the most compelling reason why theoretical computer scientists believe

that P ¤ NP comes from the existence of the class of “NP-complete” problems.

This class has the intriguing property that if any NP-complete problem can be

solved in polynomial time, then every problem in NP has a polynomial-time solu-

tion, that is, P D NP. Despite years of study, though, no polynomial-time algorithm

has ever been discovered for any NP-complete problem.

The language HAM-CYCLE is one NP-complete problem. If we could decide

HAM-CYCLE in polynomial time, then we could solve every problem in NP in

polynomial time. In fact, if NP P should turn out to be nonempty, we could say

with certainty that HAM-CYCLE 2 NP P.

The NP-complete languages are, in a sense, the “hardest” languages in NP. In

this section, we shall show how to compare the relative “hardness” of languages

using a precise notion called “polynomial-time reducibility.” Then we formally

deﬁne the NP-complete languages, and we ﬁnish by sketching a proof that one

such language, called CIRCUIT-SAT, is NP-complete. In Sections 34.4 and 34.5,

we shall use the notion of reducibility to show that many other problems are NP-

complete.

Reducibility

Intuitively, a problem Q can be reduced to another problem Q 0

if any instance of Q

can be “easily rephrased” as an instance of Q 0

, the solution to which provides a

solution to the instance of Q. For example, the problem of solving linear equations

in an indeterminate x reduces to the problem of solving quadratic equations. Given

an instance ax C b D 0, we transform it to 0x 2 C ax C b D 0, whose solution

provides a solution to ax C b D 0. Thus, if a problem Q reduces to another

problem Q 0

, then Q is, in a sense, “no harder to solve” than Q 0

.

Returning to our formal-language framework for decision problems, we say that

a language L 1 is polynomial-time reducible to a language L 2 , written L 1 P L 2 ,

if there exists a polynomial-time computable function f W f0; 1g

! f0; 1g

such

that for all x 2 f0; 1g

,

x 2 L 1 if and only if f .x/ 2 L 2 : (34.1)

We call the function f the reduction function, and a polynomial-time algorithm F

that computes f is a reduction algorithm.

Figure 34.4 illustrates the idea of a polynomial-time reduction from a lan-

guage L 1 to another language L 2 . Each language is a subset of f0; 1g

. The

reduction function f provides a polynomial-time mapping such that if x 2 L 1 ,

1068 Chapter 34 NP-Completeness

L

2

L

1

{0,1}\* {0,1}\* f

Figure 34.4 An illustration of a polynomial-time reduction from a language L1 to a language L2

via a reduction function f . For any input x 2 f0; 1g

, the question of whether x 2 L1 has the same

answer as the question of whether f .x/ 2 L2.

then f .x/ 2 L 2 . Moreover, if x 62 L 1 , then f .x/ 62 L 2 . Thus, the reduction func-

tion maps any instance x of the decision problem represented by the language L 1

to an instance f .x/ of the problem represented by L 2 . Providing an answer to

whether f .x/ 2 L 2 directly provides the answer to whether x 2 L 1 .

Polynomial-time reductions give us a powerful tool for proving that various lan-

guages belong to P.

Lemma 34.3

If L 1 ; L 2 f0; 1g

are languages such that L 1 P L 2 , then L 2 2 P implies

L 1 2 P.

Proof Let A 2 be a polynomial-time algorithm that decides L 2 , and let F be a

polynomial-time reduction algorithm that computes the reduction function f . We

shall construct a polynomial-time algorithm A 1 that decides L 1 .

Figure 34.5 illustrates how we construct A 1 . For a given input x 2 f0; 1g

,

algorithm A 1 uses F to transform x into f .x/, and then it uses A 2 to test whether

f .x/ 2 L 2 . Algorithm A 1 takes the output from algorithm A 2 and produces that

answer as its own output.

The correctness of A 1 follows from condition (34.1). The algorithm runs in poly-

nomial time, since both F and A 2 run in polynomial time (see Exercise 34.1-5).

NP-completeness

Polynomial-time reductions provide a formal means for showing that one prob-

lem is at least as hard as another, to within a polynomial-time factor. That is, if

L 1 P L 2 , then L 1 is not more than a polynomial factor harder than L 2 , which is

34.3 NP-completeness and reducibility 1069

x

F

f .x/

A 1

A 2

yes, f .x/ 2 L 2

no, f .x/ 62 L 2

yes, x 2 L 1

no, x 62 L 1

Figure 34.5 The proof of Lemma 34.3. The algorithm F is a reduction algorithm that computes the

reduction function f from L1 to L2 in polynomial time, and A2 is a polynomial-time algorithm that

decides L2. Algorithm A1 decides whether x 2 L1 by using F to transform any input x into f .x/

and then using A2 to decide whether f .x/ 2 L2.

why the “less than or equal to” notation for reduction is mnemonic. We can now

deﬁne the set of NP-complete languages, which are the hardest problems in NP.

A language L f0; 1g

is NP-complete if

1. L 2 NP, and

2. L 0 P L for every L 0 2 NP.

If a language L satisﬁes property 2, but not necessarily property 1, we say that L

is NP-hard. We also deﬁne NPC to be the class of NP-complete languages.

As the following theorem shows, NP-completeness is at the crux of deciding

whether P is in fact equal to NP.

Theorem 34.4

If any NP-complete problem is polynomial-time solvable, then P D NP. Equiva-

lently, if any problem in NP is not polynomial-time solvable, then no NP-complete

problem is polynomial-time solvable.

Proof Suppose that L 2 P and also that L 2 NPC. For any L 0 2 NP, we

have L 0 P L by property 2 of the deﬁnition of NP-completeness. Thus, by

Lemma 34.3, we also have that L 0 2 P, which proves the ﬁrst statement of the

theorem.

To prove the second statement, note that it is the contrapositive of the ﬁrst state-

ment.

It is for this reason that research into the P ¤ NP question centers around the

NP-complete problems. Most theoretical computer scientists believe that P ¤ NP,

which leads to the relationships among P, NP, and NPC shown in Figure 34.6.

But, for all we know, someone may yet come up with a polynomial-time algo-

rithm for an NP-complete problem, thus proving that P D NP. Nevertheless, since

no polynomial-time algorithm for any NP-complete problem has yet been discov-

1070 Chapter 34 NP-Completeness

NPC

P

NP

Figure 34.6 How most theoretical computer scientists view the relationships among P, NP,

and NPC. Both P and NPC are wholly contained within NP, and P \ NPC D ;.

ered, a proof that a problem is NP-complete provides excellent evidence that it is

intractable.

Circuit satisﬁability

We have deﬁned the notion of an NP-complete problem, but up to this point, we

have not actually proved that any problem is NP-complete. Once we prove that at

least one problem is NP-complete, we can use polynomial-time reducibility as a

tool to prove other problems to be NP-complete. Thus, we now focus on demon-

strating the existence of an NP-complete problem: the circuit-satisﬁability prob-

lem.

Unfortunately, the formal proof that the circuit-satisﬁability problem is NP-

complete requires technical detail beyond the scope of this text. Instead, we shall

informally describe a proof that relies on a basic understanding of boolean combi-

national circuits.

Boolean combinational circuits are built from boolean combinational elements

that are interconnected by wires. A boolean combinational element is any circuit

element that has a constant number of boolean inputs and outputs and that performs

a well-deﬁned function. Boolean values are drawn from the set f0; 1g, where 0

represents FALSE and 1 represents TRUE.

The boolean combinational elements that we use in the circuit-satisﬁability prob-

lem compute simple boolean functions, and they are known as logic gates. Fig-

ure 34.7 shows the three basic logic gates that we use in the circuit-satisﬁability

problem: the NOT gate (or inverter), the AND gate, and the OR gate. The NOT

gate takes a single binary input x, whose value is either 0 or 1, and produces a

binary output ´ whose value is opposite that of the input value. Each of the other

two gates takes two binary inputs x and y and produces a single binary output ´.

We can describe the operation of each gate, and of any boolean combinational

element, by a truth table, shown under each gate in Figure 34.7. A truth table gives

the outputs of the combinational element for each possible setting of the inputs. For

34.3 NP-completeness and reducibility 1071

x

y

z

x

y

z

0 0 0

0 1 0

1 0 0

1 1 1

0 0 0

0 1 1

1 0 1

1 1 1

(b) (c)

x z

0 1

1 0

(a)

x x x y y :x x ^ y x \_ y

Figure 34.7 Three basic logic gates, with binary inputs and outputs. Under each gate is the truth

table that describes the gate’s operation. (a) The NOT gate. (b) The AND gate. (c) The OR gate.

example, the truth table for the OR gate tells us that when the inputs are x D 0

and y D 1, the output value is ´ D 1. We use the symbols : to denote the NOT

function, ^ to denote the AND function, and \_ to denote the OR function. Thus,

for example, 0 \_ 1 D 1.

We can generalize AND and OR gates to take more than two inputs. An AND

gate’s output is 1 if all of its inputs are 1, and its output is 0 otherwise. An OR gate’s

output is 1 if any of its inputs are 1, and its output is 0 otherwise.

A boolean combinational circuit consists of one or more boolean combinational

elements interconnected by wires. A wire can connect the output of one element

to the input of another, thereby providing the output value of the ﬁrst element as an

input value of the second. Figure 34.8 shows two similar boolean combinational

circuits, differing in only one gate. Part (a) of the ﬁgure also shows the values on

the individual wires, given the input hx 1 D 1; x 2 D 1; x 3 D 0i. Although a single

wire may have no more than one combinational-element output connected to it, it

can feed several element inputs. The number of element inputs fed by a wire is

called the fan-out of the wire. If no element output is connected to a wire, the wire

is a circuit input, accepting input values from an external source. If no element

input is connected to a wire, the wire is a circuit output, providing the results of

the circuit’s computation to the outside world. (An internal wire can also fan out

to a circuit output.) For the purpose of deﬁning the circuit-satisﬁability problem,

we limit the number of circuit outputs to 1, though in actual hardware design, a

boolean combinational circuit may have multiple outputs.

Boolean combinational circuits contain no cycles. In other words, suppose we

create a directed graph G D .V; E/ with one vertex for each combinational element

and with k directed edges for each wire whose fan-out is k; the graph contains

a directed edge .u; / if a wire connects the output of element u to an input of

element . Then G must be acyclic.

1072 Chapter 34 NP-Completeness

x

3

x

2

x

1

(a)

1

1

0

1

1

1

1

1

1

1

1

0

0

1

1

1

1

x

3

x

2

x

1

(b)

Figure 34.8 Two instances of the circuit-satisﬁability problem. (a) The assignment hx1 D 1;

x2 D 1; x3 D 0i to the inputs of this circuit causes the output of the circuit to be 1. The circuit

is therefore satisﬁable. (b) No assignment to the inputs of this circuit can cause the output of the

circuit to be 1. The circuit is therefore unsatisﬁable.

A truth assignment for a boolean combinational circuit is a set of boolean input

values. We say that a one-output boolean combinational circuit is satisﬁable if it

has a satisfying assignment: a truth assignment that causes the output of the circuit

to be 1. For example, the circuit in Figure 34.8(a) has the satisfying assignment

hx 1 D 1; x 2 D 1; x 3 D 0i, and so it is satisﬁable. As Exercise 34.3-1 asks you to

show, no assignment of values to x 1 , x 2 , and x 3 causes the circuit in Figure 34.8(b)

to produce a 1 output; it always produces 0, and so it is unsatisﬁable.

The circuit-satisﬁability problem is, “Given a boolean combinational circuit

composed of AND, OR, and NOT gates, is it satisﬁable?” In order to pose this

question formally, however, we must agree on a standard encoding for circuits.

The size of a boolean combinational circuit is the number of boolean combina-

tional elements plus the number of wires in the circuit. We could devise a graphlike

encoding that maps any given circuit C into a binary string hC i whose length is

polynomial in the size of the circuit itself. As a formal language, we can therefore

deﬁne

CIRCUIT-SAT D fhC i W C is a satisﬁable boolean combinational circuitg :

The circuit-satisﬁability problem arises in the area of computer-aided hardware

optimization. If a subcircuit always produces 0, that subcircuit is unnecessary;

the designer can replace it by a simpler subcircuit that omits all logic gates and

provides the constant 0 value as its output. You can see why we would like to have

a polynomial-time algorithm for this problem.

Given a circuit C , we might attempt to determine whether it is satisﬁable by

simply checking all possible assignments to the inputs. Unfortunately, if the circuit

has k inputs, then we would have to check up to 2 k

possible assignments. When

34.3 NP-completeness and reducibility 1073

the size of C is polynomial in k, checking each one takes .2 k / time, which is

superpolynomial in the size of the circuit. 9 In fact, as we have claimed, there is

strong evidence that no polynomial-time algorithm exists that solves the circuit-

satisﬁability problem because circuit satisﬁability is NP-complete. We break the

proof of this fact into two parts, based on the two parts of the deﬁnition of NP-

completeness.

Lemma 34.5

The circuit-satisﬁability problem belongs to the class NP.

Proof We shall provide a two-input, polynomial-time algorithm A that can verify

CIRCUIT-SAT. One of the inputs to A is (a standard encoding of) a boolean com-

binational circuit C . The other input is a certiﬁcate corresponding to an assignment

of boolean values to the wires in C . (See Exercise 34.3-4 for a smaller certiﬁcate.)

We construct the algorithm A as follows. For each logic gate in the circuit, it

checks that the value provided by the certiﬁcate on the output wire is correctly

computed as a function of the values on the input wires. Then, if the output of the

entire circuit is 1, the algorithm outputs 1, since the values assigned to the inputs

of C provide a satisfying assignment. Otherwise, A outputs 0.

Whenever a satisﬁable circuit C is input to algorithm A, there exists a certiﬁcate

whose length is polynomial in the size of C and that causes A to output a 1. When-

ever an unsatisﬁable circuit is input, no certiﬁcate can fool A into believing that

the circuit is satisﬁable. Algorithm A runs in polynomial time: with a good imple-

mentation, linear time sufﬁces. Thus, we can verify CIRCUIT-SAT in polynomial

time, and CIRCUIT-SAT 2 NP.

The second part of proving that CIRCUIT-SAT is NP-complete is to show that

the language is NP-hard. That is, we must show that every language in NP is

polynomial-time reducible to CIRCUIT-SAT. The actual proof of this fact is full

of technical intricacies, and so we shall settle for a sketch of the proof based on

some understanding of the workings of computer hardware.

A computer program is stored in the computer memory as a sequence of in-

structions. A typical instruction encodes an operation to be performed, addresses

of operands in memory, and an address where the result is to be stored. A spe-

cial memory location, called the program counter, keeps track of which instruc-

9

On the other hand, if the size of the circuit C is ‚.2

k

/, then an algorithm whose running time

is O.2

k

/ has a running time that is polynomial in the circuit size. Even if P ¤ NP, this situa-

tion would not contradict the NP-completeness of the problem; the existence of a polynomial-time

algorithm for a special case does not imply that there is a polynomial-time algorithm for all cases.

1074 Chapter 34 NP-Completeness

tion is to be executed next. The program counter automatically increments upon

fetching each instruction, thereby causing the computer to execute instructions se-

quentially. The execution of an instruction can cause a value to be written to the

program counter, however, which alters the normal sequential execution and allows

the computer to loop and perform conditional branches.

At any point during the execution of a program, the computer’s memory holds

the entire state of the computation. (We take the memory to include the program

itself, the program counter, working storage, and any of the various bits of state

that a computer maintains for bookkeeping.) We call any particular state of com-

puter memory a conﬁguration. We can view the execution of an instruction as

mapping one conﬁguration to another. The computer hardware that accomplishes

this mapping can be implemented as a boolean combinational circuit, which we

denote by M in the proof of the following lemma.

Lemma 34.6

The circuit-satisﬁability problem is NP-hard.

Proof Let L be any language in NP. We shall describe a polynomial-time algo-

rithm F computing a reduction function f that maps every binary string x to a

circuit C D f .x/ such that x 2 L if and only if C 2 CIRCUIT-SAT.

Since L 2 NP, there must exist an algorithm A that veriﬁes L in polynomial

time. The algorithm F that we shall construct uses the two-input algorithm A to

compute the reduction function f .

Let T .n/ denote the worst-case running time of algorithm A on length-n input

strings, and let k 1 be a constant such that T .n/ D O.n k / and the length of the

certiﬁcate is O.n k /. (The running time of A is actually a polynomial in the total

input size, which includes both an input string and a certiﬁcate, but since the length

of the certiﬁcate is polynomial in the length n of the input string, the running time

is polynomial in n.)

The basic idea of the proof is to represent the computation of A as a sequence

of conﬁgurations. As Figure 34.9 illustrates, we can break each conﬁguration into

parts consisting of the program for A, the program counter and auxiliary machine

state, the input x, the certiﬁcate y, and working storage. The combinational cir-

cuit M , which implements the computer hardware, maps each conﬁguration c i to

the next conﬁguration c iC1 , starting from the initial conﬁguration c 0 . Algorithm A

writes its output—0 or 1—to some designated location by the time it ﬁnishes ex-

ecuting, and if we assume that thereafter A halts, the value never changes. Thus,

if the algorithm runs for at most T .n/ steps, the output appears as one of the bits

in c T .n/ .

The reduction algorithm F constructs a single combinational circuit that com-

putes all conﬁgurations produced by a given initial conﬁguration. The idea is to

34.3 NP-completeness and reducibility 1075

M

A PC aux machine state x y working storage

A PC aux machine state x y working storage

M

A PC aux machine state x y working storage

M

A PC aux machine state x y

…

working storage

0/1 output

M

c

0

c

1

c

2

c

T(n)

Figure 34.9 The sequence of conﬁgurations produced by an algorithm A running on an input x and

certiﬁcate y. Each conﬁguration represents the state of the computer for one step of the computation

and, besides A, x, and y, includes the program counter (PC), auxiliary machine state, and working

storage. Except for the certiﬁcate y, the initial conﬁguration c0 is constant. A boolean combinational

circuit M maps each conﬁguration to the next conﬁguration. The output is a distinguished bit in the

working storage.

1076 Chapter 34 NP-Completeness

paste together T .n/ copies of the circuit M . The output of the ith circuit, which

produces conﬁguration c i , feeds directly into the input of the .i C1/st circuit. Thus,

the conﬁgurations, rather than being stored in the computer’s memory, simply re-

side as values on the wires connecting copies of M .

Recall what the polynomial-time reduction algorithm F must do. Given an in-

put x, it must compute a circuit C D f .x/ that is satisﬁable if and only if there

exists a certiﬁcate y such that A.x; y/ D 1. When F obtains an input x, it ﬁrst

computes n D jxj and constructs a combinational circuit C 0

consisting of T .n/

copies of M . The input to C 0

is an initial conﬁguration corresponding to a compu-

tation on A.x; y/, and the output is the conﬁguration c T .n/ .

Algorithm F modiﬁes circuit C 0

slightly to construct the circuit C D f .x/.

First, it wires the inputs to C 0

corresponding to the program for A, the initial pro-

gram counter, the input x, and the initial state of memory directly to these known

values. Thus, the only remaining inputs to the circuit correspond to the certiﬁ-

cate y. Second, it ignores all outputs from C 0

, except for the one bit of c T .n/

corresponding to the output of A. This circuit C , so constructed, computes

C.y/ D A.x; y/ for any input y of length O.n k /. The reduction algorithm F ,

when provided an input string x, computes such a circuit C and outputs it.

We need to prove two properties. First, we must show that F correctly computes

a reduction function f . That is, we must show that C is satisﬁable if and only if

there exists a certiﬁcate y such that A.x; y/ D 1. Second, we must show that F

runs in polynomial time.

To show that F correctly computes a reduction function, let us suppose that there

exists a certiﬁcate y of length O.n k / such that A.x; y/ D 1. Then, if we apply the

bits of y to the inputs of C , the output of C is C.y/ D A.x; y/ D 1. Thus, if a

certiﬁcate exists, then C is satisﬁable. For the other direction, suppose that C is

satisﬁable. Hence, there exists an input y to C such that C.y/ D 1, from which

we conclude that A.x; y/ D 1. Thus, F correctly computes a reduction function.

To complete the proof sketch, we need only show that F runs in time polynomial

in n D jxj. The ﬁrst observation we make is that the number of bits required to

represent a conﬁguration is polynomial in n. The program for A itself has constant

size, independent of the length of its input x. The length of the input x is n, and

the length of the certiﬁcate y is O.n k /. Since the algorithm runs for at most O.n k /

steps, the amount of working storage required by A is polynomial in n as well.

(We assume that this memory is contiguous; Exercise 34.3-5 asks you to extend

the argument to the situation in which the locations accessed by A are scattered

across a much larger region of memory and the particular pattern of scattering can

differ for each input x.)

The combinational circuit M implementing the computer hardware has size

polynomial in the length of a conﬁguration, which is O.n k /; hence, the size of M

is polynomial in n. (Most of this circuitry implements the logic of the memory

34.3 NP-completeness and reducibility 1077

system.) The circuit C consists of at most t D O.n k / copies of M , and hence it

has size polynomial in n. The reduction algorithm F can construct C from x in

polynomial time, since each step of the construction takes polynomial time.

The language CIRCUIT-SAT is therefore at least as hard as any language in NP,

and since it belongs to NP, it is NP-complete.

Theorem 34.7

The circuit-satisﬁability problem is NP-complete.

Proof Immediate from Lemmas 34.5 and 34.6 and from the deﬁnition of NP-

completeness.

Exercises

34.3-1

Verify that the circuit in Figure 34.8(b) is unsatisﬁable.

34.3-2

Show that the P relation is a transitive relation on languages. That is, show that if

L 1 P L 2 and L 2 P L 3 , then L 1 P L 3 .

34.3-3

Prove that L P L if and only if L P L.

34.3-4

Show that we could have used a satisfying assignment as a certiﬁcate in an alter-

native proof of Lemma 34.5. Which certiﬁcate makes for an easier proof?

34.3-5

The proof of Lemma 34.6 assumes that the working storage for algorithm A occu-

pies a contiguous region of polynomial size. Where in the proof do we exploit this

assumption? Argue that this assumption does not involve any loss of generality.

34.3-6

A language L is complete for a language class C with respect to polynomial-time

reductions if L 2 C and L 0 P L for all L 0 2 C . Show that ; and f0; 1g

are the

only languages in P that are not complete for P with respect to polynomial-time

reductions.

1078 Chapter 34 NP-Completeness

34.3-7

Show that, with respect to polynomial-time reductions (see Exercise 34.3-6), L is

complete for NP if and only if L is complete for co-NP.

34.3-8

The reduction algorithm F in the proof of Lemma 34.6 constructs the circuit

C D f .x/ based on knowledge of x, A, and k. Professor Sartre observes that

the string x is input to F , but only the existence of A, k, and the constant factor

implicit in the O.n k / running time is known to F (since the language L belongs

to NP), not their actual values. Thus, the professor concludes that F can’t possi-

bly construct the circuit C and that the language CIRCUIT-SAT is not necessarily

NP-hard. Explain the ﬂaw in the professor’s reasoning.

34.4 NP-completeness proofs

We proved that the circuit-satisﬁability problem is NP-complete by a direct proof

that L P CIRCUIT-SAT for every language L 2 NP. In this section, we shall

show how to prove that languages are NP-complete without directly reducing every

language in NP to the given language. We shall illustrate this methodology by

proving that various formula-satisﬁability problems are NP-complete. Section 34.5

provides many more examples of the methodology.

The following lemma is the basis of our method for showing that a language is

NP-complete.

Lemma 34.8

If L is a language such that L 0 P L for some L 0 2 NPC, then L is NP-hard. If, in

addition, L 2 NP, then L 2 NPC.

Proof Since L 0

is NP-complete, for all L 00 2 NP, we have L 00 P L 0

. By sup-

position, L 0 P L, and thus by transitivity (Exercise 34.3-2), we have L 00 P L,

which shows that L is NP-hard. If L 2 NP, we also have L 2 NPC.

In other words, by reducing a known NP-complete language L 0

to L, we implic-

itly reduce every language in NP to L. Thus, Lemma 34.8 gives us a method for

proving that a language L is NP-complete:

1. Prove L 2 NP.

2. Select a known NP-complete language L 0

.

34.4 NP-completeness proofs 1079

3. Describe an algorithm that computes a function f mapping every instance

x 2 f0; 1g

of L 0

to an instance f .x/ of L.

4. Prove that the function f satisﬁes x 2 L 0

if and only if f .x/ 2 L for all

x 2 f0; 1g

.

5. Prove that the algorithm computing f runs in polynomial time.

(Steps 2–5 show that L is NP-hard.) This methodology of reducing from a sin-

gle known NP-complete language is far simpler than the more complicated pro-

cess of showing directly how to reduce from every language in NP. Proving

CIRCUIT-SAT 2 NPC has given us a “foot in the door.” Because we know that the

circuit-satisﬁability problem is NP-complete, we now can prove much more easily

that other problems are NP-complete. Moreover, as we develop a catalog of known

NP-complete problems, we will have more and more choices for languages from

which to reduce.

Formula satisﬁability

We illustrate the reduction methodology by giving an NP-completeness proof for

the problem of determining whether a boolean formula, not a circuit, is satisﬁable.

This problem has the historical honor of being the ﬁrst problem ever shown to be

NP-complete.

We formulate the (formula) satisﬁability problem in terms of the language SAT

as follows. An instance of SAT is a boolean formula composed of

1. n boolean variables: x 1 ; x 2 ; : : : ; x n ;

2. m boolean connectives: any boolean function with one or two inputs and one

output, such as ^ (AND), \_ (OR), : (NOT), ! (implication), $ (if and only

if); and

3. parentheses. (Without loss of generality, we assume that there are no redundant

parentheses, i.e., a formula contains at most one pair of parentheses per boolean

connective.)

We can easily encode a boolean formula in a length that is polynomial in n C m.

As in boolean combinational circuits, a truth assignment for a boolean formula

is a set of values for the variables of , and a satisfying assignment is a truth

assignment that causes it to evaluate to 1. A formula with a satisfying assignment

is a satisﬁable formula. The satisﬁability problem asks whether a given boolean

formula is satisﬁable; in formal-language terms,

SAT D fhi W is a satisﬁable boolean formulag :

As an example, the formula

1080 Chapter 34 NP-Completeness

D ..x 1 ! x 2 / \_ :..:x 1 $ x 3 / \_ x 4 // ^ :x 2

has the satisfying assignment hx 1 D 0; x 2 D 0; x 3 D 1; x 4 D 1i, since

D ..0 ! 0/ \_ :..:0 $ 1/ \_ 1// ^ :0 (34.2)

D .1 \_ :.1 \_ 1// ^ 1

D .1 \_ 0/ ^ 1

D 1 ;

and thus this formula belongs to SAT.

The naive algorithm to determine whether an arbitrary boolean formula is satis-

ﬁable does not run in polynomial time. A formula with n variables has 2 n

possible

assignments. If the length of hi is polynomial in n, then checking every assign-

ment requires .2 n / time, which is superpolynomial in the length of hi. As the

following theorem shows, a polynomial-time algorithm is unlikely to exist.

Theorem 34.9

Satisﬁability of boolean formulas is NP-complete.

Proof We start by arguing that SAT 2 NP. Then we prove that SAT is NP-hard by

showing that CIRCUIT-SAT P SAT; by Lemma 34.8, this will prove the theorem.

To show that SAT belongs to NP, we show that a certiﬁcate consisting of a

satisfying assignment for an input formula can be veriﬁed in polynomial time.

The verifying algorithm simply replaces each variable in the formula with its cor-

responding value and then evaluates the expression, much as we did in equa-

tion (34.2) above. This task is easy to do in polynomial time. If the expression

evaluates to 1, then the algorithm has veriﬁed that the formula is satisﬁable. Thus,

the ﬁrst condition of Lemma 34.8 for NP-completeness holds.

To prove that SAT is NP-hard, we show that CIRCUIT-SAT P SAT. In other

words, we need to show how to reduce any instance of circuit satisﬁability to an

instance of formula satisﬁability in polynomial time. We can use induction to

express any boolean combinational circuit as a boolean formula. We simply look

at the gate that produces the circuit output and inductively express each of the

gate’s inputs as formulas. We then obtain the formula for the circuit by writing an

expression that applies the gate’s function to its inputs’ formulas.

Unfortunately, this straightforward method does not amount to a polynomial-

time reduction. As Exercise 34.4-1 asks you to show, shared subformulas—which

arise from gates whose output wires have fan-out of 2 or more—can cause the

size of the generated formula to grow exponentially. Thus, the reduction algorithm

must be somewhat more clever.

Figure 34.10 illustrates how we overcome this problem, using as an example

the circuit from Figure 34.8(a). For each wire x i in the circuit C , the formula

34.4 NP-completeness proofs 1081

x

6

x

3

x

4

x

7

x

10

x

9

x

8

x

5

x

2

x

1

Figure 34.10 Reducing circuit satisﬁability to formula satisﬁability. The formula produced by the

reduction algorithm has a variable for each wire in the circuit.

has a variable x i . We can now express how each gate operates as a small formula

involving the variables of its incident wires. For example, the operation of the

output AND gate is x 10 $ .x 7 ^ x 8 ^ x 9 /. We call each of these small formulas a

clause.

The formula produced by the reduction algorithm is the AND of the circuit-

output variable with the conjunction of clauses describing the operation of each

gate. For the circuit in the ﬁgure, the formula is

D x 10 ^ .x 4 $ :x 3 /

^ .x 5 $ .x 1 \_ x 2 //

^ .x 6 $ :x 4 /

^ .x 7 $ .x 1 ^ x 2 ^ x 4 //

^ .x 8 $ .x 5 \_ x 6 //

^ .x 9 $ .x 6 \_ x 7 //

^ .x 10 $ .x 7 ^ x 8 ^ x 9 // :

Given a circuit C , it is straightforward to produce such a formula in polynomial

time.

Why is the circuit C satisﬁable exactly when the formula is satisﬁable? If C

has a satisfying assignment, then each wire of the circuit has a well-deﬁned value,

and the output of the circuit is 1. Therefore, when we assign wire values to

variables in , each clause of evaluates to 1, and thus the conjunction of all

evaluates to 1. Conversely, if some assignment causes to evaluate to 1, the

circuit C is satisﬁable by an analogous argument. Thus, we have shown that

CIRCUIT-SAT P SAT, which completes the proof.

1082 Chapter 34 NP-Completeness

3-CNF satisﬁability

We can prove many problems NP-complete by reducing from formula satisﬁability.

The reduction algorithm must handle any input formula, though, and this require-

ment can lead to a huge number of cases that we must consider. We often prefer

to reduce from a restricted language of boolean formulas, so that we need to con-

sider fewer cases. Of course, we must not restrict the language so much that it

becomes polynomial-time solvable. One convenient language is 3-CNF satisﬁabil-

ity, or 3-CNF-SAT.

We deﬁne 3-CNF satisﬁability using the following terms. A literal in a boolean

formula is an occurrence of a variable or its negation. A boolean formula is in

conjunctive normal form, or CNF, if it is expressed as an AND of clauses, each

of which is the OR of one or more literals. A boolean formula is in 3-conjunctive

normal form, or 3-CNF, if each clause has exactly three distinct literals.

For example, the boolean formula

.x 1 \_ :x 1 \_ :x 2 / ^ .x 3 \_ x 2 \_ x 4 / ^ .:x 1 \_ :x 3 \_ :x 4 /

is in 3-CNF. The ﬁrst of its three clauses is .x 1 \_ :x 1 \_ :x 2 /, which contains the

three literals x 1 , :x 1 , and :x 2 .

In 3-CNF-SAT, we are asked whether a given boolean formula in 3-CNF is

satisﬁable. The following theorem shows that a polynomial-time algorithm that

can determine the satisﬁability of boolean formulas is unlikely to exist, even when

they are expressed in this simple normal form.

Theorem 34.10

Satisﬁability of boolean formulas in 3-conjunctive normal form is NP-complete.

Proof The argument we used in the proof of Theorem 34.9 to show that SAT 2

NP applies equally well here to show that 3-CNF-SAT 2 NP. By Lemma 34.8,

therefore, we need only show that SAT P 3-CNF-SAT.

We break the reduction algorithm into three basic steps. Each step progressively

transforms the input formula closer to the desired 3-conjunctive normal form.

The ﬁrst step is similar to the one used to prove CIRCUIT-SAT P SAT in

Theorem 34.9. First, we construct a binary “parse” tree for the input formula ,

with literals as leaves and connectives as internal nodes. Figure 34.11 shows such

a parse tree for the formula

D ..x 1 ! x 2 / \_ :..:x 1 $ x 3 / \_ x 4 // ^ :x 2 : (34.3)

Should the input formula contain a clause such as the OR of several literals, we use

associativity to parenthesize the expression fully so that every internal node in the

resulting tree has 1 or 2 children. We can now think of the binary parse tree as a

circuit for computing the function.

34.4 NP-completeness proofs 1083

:x 1

x 1

:x 2

x 2

x 3

x 4

y 1

y 2

y 3 y 4

y 5

y 6

^

$

\_

\_

: !

Figure 34.11 The tree corresponding to the formula D ..x1 !x2/\_:..:x1 $x3/\_x4//^:x2:

Mimicking the reduction in the proof of Theorem 34.9, we introduce a vari-

able y i for the output of each internal node. Then, we rewrite the original for-

mula as the AND of the root variable and a conjunction of clauses describing the

operation of each node. For the formula (34.3), the resulting expression is

0

D y 1 ^ .y 1 $ .y 2 ^ :x 2 //

^ .y 2 $ .y 3 \_ y 4 //

^ .y 3 $ .x 1 ! x 2 //

^ .y 4 $ :y 5 /

^ .y 5 $ .y 6 \_ x 4 //

^ .y 6 $ .:x 1 $ x 3 // :

Observe that the formula 0

thus obtained is a conjunction of clauses 0

i

, each of

which has at most 3 literals. The only requirement that we might fail to meet is

that each clause has to be an OR of 3 literals.

The second step of the reduction converts each clause 0

i

into conjunctive normal

form. We construct a truth table for 0

i

by evaluating all possible assignments to

its variables. Each row of the truth table consists of a possible assignment of the

variables of the clause, together with the value of the clause under that assignment.

Using the truth-table entries that evaluate to 0, we build a formula in disjunctive

normal form (or DNF)—an OR of ANDs—that is equivalent to : 0

i

. We then

negate this formula and convert it into a CNF formula 00

i

by using DeMorgan’s

1084 Chapter 34 NP-Completeness

y1 y2 x2 .y1 $ .y2 ^ :x2//

1 1 1 0

1 1 0 1

1 0 1 0

1 0 0 0

0 1 1 1

0 1 0 0

0 0 1 1

0 0 0 1

Figure 34.12 The truth table for the clause .y1 $ .y2 ^ :x2//.

laws for propositional logic,

:.a ^ b/ D :a \_ :b ;

:.a \_ b/ D :a ^ :b ;

to complement all literals, change ORs into ANDs, and change ANDs into ORs.

In our example, we convert the clause 0

1

D .y 1 $ .y 2 ^ :x 2 // into CNF

as follows. The truth table for 0

1

appears in Figure 34.12. The DNF formula

equivalent to : 0

1

is

.y 1 ^ y 2 ^ x 2 / \_ .y 1 ^ :y 2 ^ x 2 / \_ .y 1 ^ :y 2 ^ :x 2 / \_ .:y 1 ^ y 2 ^ :x 2 / :

Negating and applying DeMorgan’s laws, we get the CNF formula

00

1

D .:y 1 \_ :y 2 \_ :x 2 / ^ .:y 1 \_ y 2 \_ :x 2 /

^ .:y 1 \_ y 2 \_ x 2 / ^ .y 1 \_ :y 2 \_ x 2 / ;

which is equivalent to the original clause 0

1

.

At this point, we have converted each clause 0

i

of the formula 0

into a CNF

formula 00

i

, and thus 0

is equivalent to the CNF formula 00

consisting of the

conjunction of the 00

i

. Moreover, each clause of 00

has at most 3 literals.

The third and ﬁnal step of the reduction further transforms the formula so that

each clause has exactly 3 distinct literals. We construct the ﬁnal 3-CNF formula 000

from the clauses of the CNF formula 00

. The formula 000

also uses two auxiliary

variables that we shall call p and q. For each clause C i of 00

, we include the

following clauses in 000

:

If C i has 3 distinct literals, then simply include C i as a clause of 000

.

If C i has 2 distinct literals, that is, if C i D .l 1 \_ l 2 /, where l 1 and l 2 are literals,

then include .l 1 \_ l 2 \_ p/ ^ .l 1 \_ l 2 \_ :p/ as clauses of 000

. The literals

p and :p merely fulﬁll the syntactic requirement that each clause of 000

has

34.4 NP-completeness proofs 1085

exactly 3 distinct literals. Whether p D 0 or p D 1, one of the clauses is

equivalent to l 1 \_l 2 , and the other evaluates to 1, which is the identity for AND.

If C i has just 1 distinct literal l, then include .l \_ p \_ q/ ^ .l \_ p \_ :q/ ^

.l \_ :p \_ q/ ^ .l \_ :p \_ :q/ as clauses of 000

. Regardless of the values of p

and q, one of the four clauses is equivalent to l, and the other 3 evaluate to 1.

We can see that the 3-CNF formula 000

is satisﬁable if and only if is satisﬁable

by inspecting each of the three steps. Like the reduction from CIRCUIT-SAT to

SAT, the construction of 0

from in the ﬁrst step preserves satisﬁability. The

second step produces a CNF formula 00

that is algebraically equivalent to 0

. The

third step produces a 3-CNF formula 000

that is effectively equivalent to 00

, since

any assignment to the variables p and q produces a formula that is algebraically

equivalent to 00

.

We must also show that the reduction can be computed in polynomial time. Con-

structing 0

from introduces at most 1 variable and 1 clause per connective in .

Constructing 00

from 0

can introduce at most 8 clauses into 00

for each clause

from 0

, since each clause of 0

has at most 3 variables, and the truth table for

each clause has at most 2 3 D 8 rows. The construction of 000

from 00

introduces

at most 4 clauses into 000

for each clause of 00

. Thus, the size of the resulting

formula 000

is polynomial in the length of the original formula. Each of the con-

structions can easily be accomplished in polynomial time.

Exercises

34.4-1

Consider the straightforward (nonpolynomial-time) reduction in the proof of The-

orem 34.9. Describe a circuit of size n that, when converted to a formula by this

method, yields a formula whose size is exponential in n.

34.4-2

Show the 3-CNF formula that results when we use the method of Theorem 34.10

on the formula (34.3).

34.4-3

Professor Jagger proposes to show that SAT P 3-CNF-SAT by using only the

truth-table technique in the proof of Theorem 34.10, and not the other steps. That

is, the professor proposes to take the boolean formula , form a truth table for

its variables, derive from the truth table a formula in 3-DNF that is equivalent

to :, and then negate and apply DeMorgan’s laws to produce a 3-CNF formula

equivalent to . Show that this strategy does not yield a polynomial-time reduction.

1086 Chapter 34 NP-Completeness

34.4-4

Show that the problem of determining whether a boolean formula is a tautology is

complete for co-NP. (Hint: See Exercise 34.3-7.)

34.4-5

Show that the problem of determining the satisﬁability of boolean formulas in dis-

junctive normal form is polynomial-time solvable.

34.4-6

Suppose that someone gives you a polynomial-time algorithm to decide formula

satisﬁability. Describe how to use this algorithm to ﬁnd satisfying assignments in

polynomial time.

34.4-7

Let 2-CNF-SAT be the set of satisﬁable boolean formulas in CNF with exactly 2

literals per clause. Show that 2-CNF-SAT 2 P. Make your algorithm as efﬁcient as

possible. (Hint: Observe that x \_ y is equivalent to :x ! y. Reduce 2-CNF-SAT

to an efﬁciently solvable problem on a directed graph.)

34.5 NP-complete problems

NP-complete problems arise in diverse domains: boolean logic, graphs, arithmetic,

network design, sets and partitions, storage and retrieval, sequencing and schedul-

ing, mathematical programming, algebra and number theory, games and puzzles,

automata and language theory, program optimization, biology, chemistry, physics,

and more. In this section, we shall use the reduction methodology to provide NP-

completeness proofs for a variety of problems drawn from graph theory and set

partitioning.

Figure 34.13 outlines the structure of the NP-completeness proofs in this section

and Section 34.4. We prove each language in the ﬁgure to be NP-complete by

reduction from the language that points to it. At the root is CIRCUIT-SAT, which

we proved NP-complete in Theorem 34.7.

34.5.1 The clique problem

A clique in an undirected graph G D .V; E/ is a subset V 0 V of vertices, each

pair of which is connected by an edge in E. In other words, a clique is a complete

subgraph of G. The size of a clique is the number of vertices it contains. The

clique problem is the optimization problem of ﬁnding a clique of maximum size in

34.5 NP-complete problems 1087

CIRCUIT-SAT

SAT

3-CNF-SAT

CLIQUE

VERTEX-COVER

SUBSET-SUM

HAM-CYCLE

TSP

Figure 34.13 The structure of NP-completeness proofs in Sections 34.4 and 34.5. All proofs ulti-

mately follow by reduction from the NP-completeness of CIRCUIT-SAT.

a graph. As a decision problem, we ask simply whether a clique of a given size k

exists in the graph. The formal deﬁnition is

CLIQUE D fhG; ki W G is a graph containing a clique of size kg :

A naive algorithm for determining whether a graph G D .V; E/ with jV j ver-

tices has a clique of size k is to list all k-subsets of V , and check each one to

see whether it forms a clique. The running time of this algorithm is .k 2

jV j

k

/,

which is polynomial if k is a constant. In general, however, k could be near jV j =2,

in which case the algorithm runs in superpolynomial time. Indeed, an efﬁcient

algorithm for the clique problem is unlikely to exist.

Theorem 34.11

The clique problem is NP-complete.

Proof To show that CLIQUE 2 NP, for a given graph G D .V; E/, we use the

set V 0 V of vertices in the clique as a certiﬁcate for G. We can check whether V 0

is a clique in polynomial time by checking whether, for each pair u; 2 V 0

, the

edge .u; / belongs to E.

We next prove that 3-CNF-SAT P CLIQUE, which shows that the clique prob-

lem is NP-hard. You might be surprised that we should be able to prove such a

result, since on the surface logical formulas seem to have little to do with graphs.

The reduction algorithm begins with an instance of 3-CNF-SAT. Let D

C 1 ^ C 2 ^ ^ C k be a boolean formula in 3-CNF with k clauses. For r D

1088 Chapter 34 NP-Completeness

x 1

x 1

x 2 x 2

x 3 x 3

:x 1

:x 2 :x 3

C 1 D x 1 \_ :x 2 \_ :x 3

C 2 D :x 1 \_ x 2 \_ x 3 C 3 D x 1 \_ x 2 \_ x 3

Figure 34.14 The graph G derived from the 3-CNF formula D C1 ^ C2 ^ C3, where C1 D

.x1 \_ :x2 \_ :x3/, C2 D .:x1 \_ x2 \_ x3/, and C3 D .x1 \_ x2 \_ x3/, in reducing 3-CNF-SAT to

CLIQUE. A satisfying assignment of the formula has x2 D 0, x3 D 1, and x1 either 0 or 1. This

assignment satisﬁes C1 with :x2, and it satisﬁes C2 and C3 with x3, corresponding to the clique

with lightly shaded vertices.

1; 2; : : : ; k, each clause C r has exactly three distinct literals l r

1

, l r

2

, and l r

3

. We shall

construct a graph G such that is satisﬁable if and only if G has a clique of size k.

We construct the graph G D .V; E/ as follows. For each clause C r D

.l r

1

\_ l r

2

\_ l r

3

/ in , we place a triple of vertices r

1

, r

2

, and r

3

into V . We put

an edge between two vertices r

i

and s

j

if both of the following hold:

r

i

and s

j

are in different triples, that is, r ¤ s, and

their corresponding literals are consistent, that is, l r

i

is not the negation of l s

j

.

We can easily build this graph from in polynomial time. As an example of this

construction, if we have

D .x 1 \_ :x 2 \_ :x 3 / ^ .:x 1 \_ x 2 \_ x 3 / ^ .x 1 \_ x 2 \_ x 3 / ;

then G is the graph shown in Figure 34.14.

We must show that this transformation of into G is a reduction. First, suppose

that has a satisfying assignment. Then each clause C r contains at least one

literal l r

i

that is assigned 1, and each such literal corresponds to a vertex r

i

. Picking

one such “true” literal from each clause yields a set V 0

of k vertices. We claim that

V 0

is a clique. For any two vertices r

i

; s

j

2 V 0

, where r ¤ s, both corresponding

literals l r

i

and l s

j

map to 1 by the given satisfying assignment, and thus the literals

34.5 NP-complete problems 1089

cannot be complements. Thus, by the construction of G, the edge . r

i

; s

j

/ belongs

to E.

Conversely, suppose that G has a clique V 0

of size k. No edges in G connect

vertices in the same triple, and so V 0

contains exactly one vertex per triple. We can

assign 1 to each literal l r

i

such that r

i

2 V 0

without fear of assigning 1 to both a

literal and its complement, since G contains no edges between inconsistent literals.

Each clause is satisﬁed, and so is satisﬁed. (Any variables that do not correspond

to a vertex in the clique may be set arbitrarily.)

In the example of Figure 34.14, a satisfying assignment of has x 2 D 0 and

x 3 D 1. A corresponding clique of size k D 3 consists of the vertices correspond-

ing to :x 2 from the ﬁrst clause, x 3 from the second clause, and x 3 from the third

clause. Because the clique contains no vertices corresponding to either x 1 or :x 1 ,

we can set x 1 to either 0 or 1 in this satisfying assignment.

Observe that in the proof of Theorem 34.11, we reduced an arbitrary instance

of 3-CNF-SAT to an instance of CLIQUE with a particular structure. You might

think that we have shown only that CLIQUE is NP-hard in graphs in which the

vertices are restricted to occur in triples and in which there are no edges between

vertices in the same triple. Indeed, we have shown that CLIQUE is NP-hard only

in this restricted case, but this proof sufﬁces to show that CLIQUE is NP-hard in

general graphs. Why? If we had a polynomial-time algorithm that solved CLIQUE

on general graphs, it would also solve CLIQUE on restricted graphs.

The opposite approach—reducing instances of 3-CNF-SAT with a special struc-

ture to general instances of CLIQUE—would not have sufﬁced, however. Why

not? Perhaps the instances of 3-CNF-SAT that we chose to reduce from were

“easy,” and so we would not have reduced an NP-hard problem to CLIQUE.

Observe also that the reduction used the instance of 3-CNF-SAT, but not the

solution. We would have erred if the polynomial-time reduction had relied on

knowing whether the formula is satisﬁable, since we do not know how to decide

whether is satisﬁable in polynomial time.

34.5.2 The vertex-cover problem

A vertex cover of an undirected graph G D .V; E/ is a subset V 0 V such that

if .u; / 2 E, then u 2 V 0

or 2 V 0

(or both). That is, each vertex “covers” its

incident edges, and a vertex cover for G is a set of vertices that covers all the edges

in E. The size of a vertex cover is the number of vertices in it. For example, the

graph in Figure 34.15(b) has a vertex cover fw; ´g of size 2.

The vertex-cover problem is to ﬁnd a vertex cover of minimum size in a given

graph. Restating this optimization problem as a decision problem, we wish to

1090 Chapter 34 NP-Completeness

u v

y x

z w

(a)

u v

y x

z w

(b)

Figure 34.15 Reducing CLIQUE to VERTEX-COVER. (a) An undirected graph G D .V; E/ with

clique V

0

D fu; ; x; yg. (b) The graph G produced by the reduction algorithm that has vertex cover

V V

0

D fw; ´g.

determine whether a graph has a vertex cover of a given size k. As a language, we

deﬁne

VERTEX-COVER D fhG; ki W graph G has a vertex cover of size kg :

The following theorem shows that this problem is NP-complete.

Theorem 34.12

The vertex-cover problem is NP-complete.

Proof We ﬁrst show that VERTEX-COVER 2 NP. Suppose we are given a graph

G D .V; E/ and an integer k. The certiﬁcate we choose is the vertex cover V 0 V

itself. The veriﬁcation algorithm afﬁrms that jV 0 j D k, and then it checks, for each

edge .u; / 2 E, that u 2 V 0

or 2 V 0

. We can easily verify the certiﬁcate in

polynomial time.

We prove that the vertex-cover problem is NP-hard by showing that CLIQUE P

VERTEX-COVER. This reduction relies on the notion of the “complement” of a

graph. Given an undirected graph G D .V; E/, we deﬁne the complement of G

as G D .V; E/, where E D f.u; / W u; 2 V; u ¤ ; and .u; / 62 Eg. In other

words, G is the graph containing exactly those edges that are not in G. Figure 34.15

shows a graph and its complement and illustrates the reduction from CLIQUE to

VERTEX-COVER.

The reduction algorithm takes as input an instance hG; ki of the clique problem.

It computes the complement G, which we can easily do in polynomial time. The

output of the reduction algorithm is the instance hG; jV j ki of the vertex-cover

problem. To complete the proof, we show that this transformation is indeed a

34.5 NP-complete problems 1091

reduction: the graph G has a clique of size k if and only if the graph G has a vertex

cover of size jV j k.

Suppose that G has a clique V 0 V with jV 0 j D k. We claim that V V 0

is a

vertex cover in G. Let .u; / be any edge in E. Then, .u; / 62 E, which implies

that at least one of u or does not belong to V 0

, since every pair of vertices in V 0

is

connected by an edge of E. Equivalently, at least one of u or is in V V 0

, which

means that edge .u; / is covered by V V 0

. Since .u; / was chosen arbitrarily

from E, every edge of E is covered by a vertex in V V 0

. Hence, the set V V 0

,

which has size jV j k, forms a vertex cover for G.

Conversely, suppose that G has a vertex cover V 0 V , where jV 0 j D jV j k.

Then, for all u; 2 V , if .u; / 2 E, then u 2 V 0

or 2 V 0

or both. The

contrapositive of this implication is that for all u; 2 V , if u 62 V 0

and 62 V 0

,

then .u; / 2 E. In other words, V V 0

is a clique, and it has size jV jjV 0 j D k.

Since VERTEX-COVER is NP-complete, we don’t expect to ﬁnd a polynomial-

time algorithm for ﬁnding a minimum-size vertex cover. Section 35.1 presents a

polynomial-time “approximation algorithm,” however, which produces “approxi-

mate” solutions for the vertex-cover problem. The size of a vertex cover produced

by the algorithm is at most twice the minimum size of a vertex cover.

Thus, we shouldn’t give up hope just because a problem is NP-complete. We

may be able to design a polynomial-time approximation algorithm that obtains

near-optimal solutions, even though ﬁnding an optimal solution is NP-complete.

Chapter 35 gives several approximation algorithms for NP-complete problems.

34.5.3 The hamiltonian-cycle problem

We now return to the hamiltonian-cycle problem deﬁned in Section 34.2.

Theorem 34.13

The hamiltonian cycle problem is NP-complete.

Proof We ﬁrst show that HAM-CYCLE belongs to NP. Given a graph G D

.V; E/, our certiﬁcate is the sequence of jV j vertices that makes up the hamiltonian

cycle. The veriﬁcation algorithm checks that this sequence contains each vertex

in V exactly once and that with the ﬁrst vertex repeated at the end, it forms a cycle

in G. That is, it checks that there is an edge between each pair of consecutive

vertices and between the ﬁrst and last vertices. We can verify the certiﬁcate in

polynomial time.

We now prove that VERTEX-COVER P HAM-CYCLE, which shows that

HAM-CYCLE is NP-complete. Given an undirected graph G D .V; E/ and an

1092 Chapter 34 NP-Completeness

[u,v,1]

[u,v,2]

[u,v,3]

[u,v,4]

[u,v,5]

[u,v,6]

[v,u,1]

[v,u,2]

[v,u,3]

[v,u,4]

[v,u,5]

[v,u,6]

W

uv

(a)

W

uv

(b)

[u,v,1]

[u,v,6]

[v,u,1]

[v,u,6]

W

uv

(c)

[u,v,1]

[u,v,6]

[v,u,1]

[v,u,6]

W

uv

(d)

[u,v,1]

[u,v,6]

[v,u,1]

[v,u,6]

Figure 34.16 The widget used in reducing the vertex-cover problem to the hamiltonian-cycle prob-

lem. An edge .u; / of graph G corresponds to widget Wu in the graph G

0

created in the reduction.

(a) The widget, with individual vertices labeled. (b)–(d) The shaded paths are the only possible ones

through the widget that include all vertices, assuming that the only connections from the widget to

the remainder of G

0

are through vertices Œu; ; 1, Œu; ; 6, Œ; u; 1, and Œ; u; 6.

integer k, we construct an undirected graph G 0 D .V 0 ; E 0 / that has a hamiltonian

cycle if and only if G has a vertex cover of size k.

Our construction uses a widget, which is a piece of a graph that enforces certain

properties. Figure 34.16(a) shows the widget we use. For each edge .u; / 2 E, the

graph G 0

that we construct will contain one copy of this widget, which we denote

by W u . We denote each vertex in W u by Œu; ; i or Œ; u; i, where 1 i 6, so

that each widget W u contains 12 vertices. Widget W u also contains the 14 edges

shown in Figure 34.16(a).

Along with the internal structure of the widget, we enforce the properties we

want by limiting the connections between the widget and the remainder of the

graph G 0

that we construct. In particular, only vertices Œu; ; 1, Œu; ; 6, Œ; u; 1,

and Œ; u; 6 will have edges incident from outside W u . Any hamiltonian cycle

of G 0

must traverse the edges of W u in one of the three ways shown in Fig-

ures 34.16(b)–(d). If the cycle enters through vertex Œu; ; 1, it must exit through

vertex Œu; ; 6, and it either visits all 12 of the widget’s vertices (Figure 34.16(b))

or the six vertices Œu; ; 1 through Œu; ; 6 (Figure 34.16(c)). In the latter case,

the cycle will have to reenter the widget to visit vertices Œ; u; 1 through Œ; u; 6.

Similarly, if the cycle enters through vertex Œ; u; 1, it must exit through ver-

tex Œ; u; 6, and it either visits all 12 of the widget’s vertices (Figure 34.16(d)) or

the six vertices Œ; u; 1 through Œ; u; 6 (Figure 34.16(c)). No other paths through

the widget that visit all 12 vertices are possible. In particular, it is impossible to

construct two vertex-disjoint paths, one of which connects Œu; ; 1 to Œ; u; 6 and

the other of which connects Œ; u; 1 to Œu; ; 6, such that the union of the two paths

contains all of the widget’s vertices.

34.5 NP-complete problems 1093

[w,x,1]

[w,x,6]

[x,w,1]

[x,w,6]

W

wx

(b)

[x,y,1]

[x,y,6]

[y,x,1]

[y,x,6]

W

xy

[w,y,1]

[w,y,6]

[y,w,1]

[y,w,6]

W

wy

[w,z,1]

[w,z,6]

[z,w,1]

[z,w,6]

W

wz

s

1

s

2

w x

z y

(a)

Figure 34.17 Reducing an instance of the vertex-cover problem to an instance of the hamiltonian-

cycle problem. (a) An undirected graph G with a vertex cover of size 2, consisting of the lightly

shaded vertices w and y. (b) The undirected graph G

0

produced by the reduction, with the hamilto-

nian path corresponding to the vertex cover shaded. The vertex cover fw; yg corresponds to edges

.s1; Œw; x; 1/ and .s2; Œy; x; 1/ appearing in the hamiltonian cycle.

The only other vertices in V 0

other than those of widgets are selector vertices

s 1 ; s 2 ; : : : ; s k . We use edges incident on selector vertices in G 0

to select the k

vertices of the cover in G.

In addition to the edges in widgets, E 0

contains two other types of edges, which

Figure 34.17 shows. First, for each vertex u 2 V , we add edges to join pairs

of widgets in order to form a path containing all widgets corresponding to edges

incident on u in G. We arbitrarily order the vertices adjacent to each vertex

u 2 V as u .1/ ; u .2/ ; : : : ; u .degree.u//

, where degree.u/ is the number of vertices

adjacent to u. We create a path in G 0

through all the widgets corresponding

to edges incident on u by adding to E 0

the edges f.Œu; u .i/ ; 6; Œu; u .iC1/ ; 1/ W

1 i degree.u/ 1g. In Figure 34.17, for example, we order the vertices ad-

jacent to w as x; y; ´, and so graph G 0

in part (b) of the ﬁgure includes the edges

1094 Chapter 34 NP-Completeness

.Œw; x; 6; Œw; y; 1/ and .Œw; y; 6; Œw; ´; 1/. For each vertex u 2 V , these edges

in G 0

ﬁll in a path containing all widgets corresponding to edges incident on u

in G.

The intuition behind these edges is that if we choose a vertex u 2 V in the vertex

cover of G, we can construct a path from Œu; u .1/ ; 1 to Œu; u .degree.u// ; 6 in G 0

that

“covers” all widgets corresponding to edges incident on u. That is, for each of these

widgets, say W u;u .i/ , the path either includes all 12 vertices (if u is in the vertex

cover but u .i/

is not) or just the six vertices Œu; u .i/ ; 1; Œu; u .i/ ; 2; : : : ; Œu; u .i/ ; 6 (if

both u and u .i/

are in the vertex cover).

The ﬁnal type of edge in E 0

joins the ﬁrst vertex Œu; u .1/ ; 1 and the last vertex

Œu; u .degree.u// ; 6 of each of these paths to each of the selector vertices. That is, we

include the edges

f.s j ; Œu; u

.1/

; 1/ W u 2 V and 1 j kg

[ f.s j ; Œu; u

.degree.u//

; 6/ W u 2 V and 1 j kg :

Next, we show that the size of G 0

is polynomial in the size of G, and hence we

can construct G 0

in time polynomial in the size of G. The vertices of G 0

are those

in the widgets, plus the selector vertices. With 12 vertices per widget, plus k jV j

selector vertices, we have a total of

jV

0

j D 12 jEj C k

12 jEj C jV j

vertices. The edges of G 0

are those in the widgets, those that go between widgets,

and those connecting selector vertices to widgets. Each widget contains 14 edges,

totaling 14 jEj in all widgets. For each vertex u 2 V , graph G 0

has degree.u/ 1

edges going between widgets, so that summed over all vertices in V ,

X

u2V

.degree.u/ 1/ D 2 jEj jV j

edges go between widgets. Finally, G 0

has two edges for each pair consisting of a

selector vertex and a vertex of V , totaling 2k jV j such edges. The total number of

edges of G 0

is therefore

jE

0

j D .14 jEj/ C .2 jEj jV j/ C .2k jV j/

D 16 jEj C .2k 1/ jV j

16 jEj C .2 jV j 1/ jV j :

Now we show that the transformation from graph G to G 0

is a reduction. That is,

we must show that G has a vertex cover of size k if and only if G 0

has a hamiltonian

cycle.

34.5 NP-complete problems 1095

Suppose that G D .V; E/ has a vertex cover V V of size k. Let

V D fu 1 ; u 2 ; : : : ; u k g. As Figure 34.17 shows, we form a hamiltonian cy-

cle in G 0

by including the following edges 10 for each vertex u j 2 V

. Include

edges

˚

.Œu j ; u

.i/

j

; 6; Œu j ; u

.iC1/

j

; 1/ W 1 i degree.u j / 1

, which connect all

widgets corresponding to edges incident on u j . We also include the edges within

these widgets as Figures 34.16(b)–(d) show, depending on whether the edge is cov-

ered by one or two vertices in V

. The hamiltonian cycle also includes the edges

f.s j ; Œu j ; u

.1/

j

; 1/ W 1 j kg

[ f.s j C1 ; Œu j ; u

.degree.u j //

j

; 6/ W 1 j k 1g

[ f.s 1 ; Œu k ; u

.degree.u k //

k

; 6/g :

By inspecting Figure 34.17, you can verify that these edges form a cycle. The cycle

starts at s 1 , visits all widgets corresponding to edges incident on u 1 , then visits s 2 ,

visits all widgets corresponding to edges incident on u 2 , and so on, until it returns

to s 1 . The cycle visits each widget either once or twice, depending on whether one

or two vertices of V

cover its corresponding edge. Because V

is a vertex cover

for G, each edge in E is incident on some vertex in V

, and so the cycle visits each

vertex in each widget of G 0

. Because the cycle also visits every selector vertex, it

is hamiltonian.

Conversely, suppose that G 0 D .V 0 ; E 0 / has a hamiltonian cycle C E 0

. We

claim that the set

V

D fu 2 V W .s j ; Œu; u

.1/

; 1/ 2 C for some 1 j kg (34.4)

is a vertex cover for G. To see why, partition C into maximal paths that start at

some selector vertex s i , traverse an edge .s i ; Œu; u .1/ ; 1/ for some u 2 V , and end

at a selector vertex s j without passing through any other selector vertex. Let us call

each such path a “cover path.” From how G 0

is constructed, each cover path must

start at some s i , take the edge .s i ; Œu; u .1/ ; 1/ for some vertex u 2 V , pass through

all the widgets corresponding to edges in E incident on u, and then end at some

selector vertex s j . We refer to this cover path as p u , and by equation (34.4), we

put u into V

. Each widget visited by p u must be W u or W u for some 2 V .

For each widget visited by p u , its vertices are visited by either one or two cover

paths. If they are visited by one cover path, then edge .u; / 2 E is covered in G

by vertex u. If two cover paths visit the widget, then the other cover path must

be p , which implies that 2 V

, and edge .u; / 2 E is covered by both u and .

10

Technically, we deﬁne a cycle in terms of vertices rather than edges (see Section B.4). In the

interest of clarity, we abuse notation here and deﬁne the hamiltonian cycle in terms of edges.

1096 Chapter 34 NP-Completeness

u v

x w

4

2 3

5

1

1

Figure 34.18 An instance of the traveling-salesman problem. Shaded edges represent a minimum-

cost tour, with cost 7.

Because each vertex in each widget is visited by some cover path, we see that each

edge in E is covered by some vertex in V

.

34.5.4 The traveling-salesman problem

In the traveling-salesman problem, which is closely related to the hamiltonian-

cycle problem, a salesman must visit n cities. Modeling the problem as a complete

graph with n vertices, we can say that the salesman wishes to make a tour, or

hamiltonian cycle, visiting each city exactly once and ﬁnishing at the city he starts

from. The salesman incurs a nonnegative integer cost c.i; j / to travel from city i

to city j , and the salesman wishes to make the tour whose total cost is minimum,

where the total cost is the sum of the individual costs along the edges of the tour.

For example, in Figure 34.18, a minimum-cost tour is hu; w; ; x; ui, with cost 7.

The formal language for the corresponding decision problem is

TSP D fhG; c; ki W G D .V; E/ is a complete graph;

c is a function from V V ! Z ;

k 2 Z , and

G has a traveling-salesman tour with cost at most kg :

The following theorem shows that a fast algorithm for the traveling-salesman

problem is unlikely to exist.

Theorem 34.14

The traveling-salesman problem is NP-complete.

Proof We ﬁrst show that TSP belongs to NP. Given an instance of the problem,

we use as a certiﬁcate the sequence of n vertices in the tour. The veriﬁcation

algorithm checks that this sequence contains each vertex exactly once, sums up the

edge costs, and checks whether the sum is at most k. This process can certainly be

done in polynomial time.

34.5 NP-complete problems 1097

To prove that TSP is NP-hard, we show that HAM-CYCLE P TSP. Let

G D .V; E/ be an instance of HAM-CYCLE. We construct an instance of TSP as

follows. We form the complete graph G 0 D .V; E 0 /, where E 0 D f.i; j / W i; j 2 V

and i ¤ j g, and we deﬁne the cost function c by

c.i; j / D

(

0 if .i; j / 2 E ;

1 if .i; j / 62 E :

(Note that because G is undirected, it has no self-loops, and so c.; / D 1 for all

vertices 2 V .) The instance of TSP is then hG 0 ; c; 0i, which we can easily create

in polynomial time.

We now show that graph G has a hamiltonian cycle if and only if graph G 0

has a

tour of cost at most 0. Suppose that graph G has a hamiltonian cycle h. Each edge

in h belongs to E and thus has cost 0 in G 0

. Thus, h is a tour in G 0

with cost 0.

Conversely, suppose that graph G 0

has a tour h 0

of cost at most 0. Since the costs

of the edges in E 0

are 0 and 1, the cost of tour h 0

is exactly 0 and each edge on the

tour must have cost 0. Therefore, h 0

contains only edges in E. We conclude that h 0

is a hamiltonian cycle in graph G.

34.5.5 The subset-sum problem

We next consider an arithmetic NP-complete problem. In the subset-sum problem,

we are given a ﬁnite set S of positive integers and an integer target t > 0. We ask

whether there exists a subset S 0 S whose elements sum to t. For example,

if S D f1; 2; 7; 14; 49; 98; 343; 686; 2409; 2793; 16808; 17206; 117705; 117993g

and t D 138457, then the subset S 0 D f1; 2; 7; 98; 343; 686; 2409; 17206; 117705g

is a solution.

As usual, we deﬁne the problem as a language:

SUBSET-SUM D fhS; ti W there exists a subset S 0 S such that t D

P

s2S 0 sg :

As with any arithmetic problem, it is important to recall that our standard encoding

assumes that the input integers are coded in binary. With this assumption in mind,

we can show that the subset-sum problem is unlikely to have a fast algorithm.

Theorem 34.15

The subset-sum problem is NP-complete.

Proof To show that SUBSET-SUM is in NP, for an instance hS;ti of the problem,

we let the subset S 0

be the certiﬁcate. A veriﬁcation algorithm can check whether

t D

P

s2S 0 s in polynomial time.

We now show that 3-CNF-SAT P SUBSET-SUM. Given a 3-CNF formula

over variables x 1 ; x 2 ; : : : ; x n with clauses C 1 ; C 2 ; : : : ; C k , each containing exactly

1098 Chapter 34 NP-Completeness

three distinct literals, the reduction algorithm constructs an instance hS; ti of the

subset-sum problem such that is satisﬁable if and only if there exists a subset

of S whose sum is exactly t. Without loss of generality, we make two simplifying

assumptions about the formula . First, no clause contains both a variable and its

negation, for such a clause is automatically satisﬁed by any assignment of values

to the variables. Second, each variable appears in at least one clause, because it

does not matter what value is assigned to a variable that appears in no clauses.

The reduction creates two numbers in set S for each variable x i and two numbers

in S for each clause C j . We shall create numbers in base 10, where each number

contains nCk digits and each digit corresponds to either one variable or one clause.

Base 10 (and other bases, as we shall see) has the property we need of preventing

carries from lower digits to higher digits.

As Figure 34.19 shows, we construct set S and target t as follows. We label

each digit position by either a variable or a clause. The least signiﬁcant k digits are

labeled by the clauses, and the most signiﬁcant n digits are labeled by variables.

The target t has a 1 in each digit labeled by a variable and a 4 in each digit

labeled by a clause.

For each variable x i , set S contains two integers i and 0

i

. Each of i and 0

i

has a 1 in the digit labeled by x i and 0s in the other variable digits. If literal x i

appears in clause C j , then the digit labeled by C j in i contains a 1. If lit-

eral :x i appears in clause C j , then the digit labeled by C j in 0

i

contains a 1.

All other digits labeled by clauses in i and 0

i

are 0.

All i and 0

i

values in set S are unique. Why? For l ¤ i, no l or 0

l

values can

equal i and 0

i

in the most signiﬁcant n digits. Furthermore, by our simplifying

assumptions above, no i and 0

i

can be equal in all k least signiﬁcant digits.

If i and 0

i

were equal, then x i and :x i would have to appear in exactly the

same set of clauses. But we assume that no clause contains both x i and :x i

and that either x i or :x i appears in some clause, and so there must be some

clause C j for which i and 0

i

differ.

For each clause C j , set S contains two integers s j and s 0

j

. Each of s j and s 0

j

has

0s in all digits other than the one labeled by C j . For s j , there is a 1 in the C j

digit, and s 0

j

has a 2 in this digit. These integers are “slack variables,” which we

use to get each clause-labeled digit position to add to the target value of 4.

Simple inspection of Figure 34.19 demonstrates that all s j and s 0

j

values in S

are unique in set S.

Note that the greatest sum of digits in any one digit position is 6, which occurs in

the digits labeled by clauses (three 1s from the i and 0

i

values, plus 1 and 2 from

34.5 NP-complete problems 1099

= 1 0 0 1 0 0 1

= 1 0 0 0 1 1 0

= 0 1 0 0 0 0 1

= 0 1 0 1 1 1 0

= 0 0 1 0 0 1 1

= 0 0 1 1 1 0 0

= 0 0 0 1 0 0 0

= 0 0 0 2 0 0 0

= 0 0 0 0 1 0 0

= 0 0 0 0 2 0 0

= 0 0 0 0 0 1 0

= 0 0 0 0 0 2 0

= 0 0 0 0 0 0 1

= 0 0 0 0 0 0 2

= 1 1 1 4 4 4 4

x 1 x 2 x 3 C 1 C 2 C 3 C 4

1

0

1

2

0

2

3

0

3

s 1

s 0

1

s 2

s 0

2

s 3

s 0

3

s 4

s 0

4

t

Figure 34.19 The reduction of 3-CNF-SAT to SUBSET-SUM. The formula in 3-CNF is D

C1^C2^C3^C4, where C1 D .x1\_:x2\_:x3/, C2 D .:x1\_:x2\_:x3/, C3 D .:x1\_:x2\_x3/,

and C4 D .x1 \_ x2 \_ x3/. A satisfying assignment of is hx1 D 0; x2 D 0; x3 D 1i. The set S

produced by the reduction consists of the base-10 numbers shown; reading from top to bottom, S D

f1001001; 1000110; 100001; 101110; 10011; 11100; 1000; 2000; 100; 200; 10; 20; 1; 2g. The target t

is 1114444. The subset S

0

S is lightly shaded, and it contains

0

1

,

0

2

, and 3, corresponding to the

satisfying assignment. It also contains slack variables s1, s

0

1

, s

0

2

, s3, s4, and s

0

4

to achieve the target

value of 4 in the digits labeled by C1 through C4.

the s j and s 0

j

values). Interpreting these numbers in base 10, therefore, no carries

can occur from lower digits to higher digits. 11

We can perform the reduction in polynomial time. The set S contains 2n C 2k

values, each of which has n C k digits, and the time to produce each digit is poly-

nomial in n C k. The target t has n C k digits, and the reduction produces each in

constant time.

We now show that the 3-CNF formula is satisﬁable if and only if there exists

a subset S 0 S whose sum is t. First, suppose that has a satisfying assignment.

For i D 1; 2; : : : ; n, if x i D 1 in this assignment, then include i in S 0

. Otherwise,

include 0

i

. In other words, we include in S 0

exactly the i and 0

i

values that cor-

11

In fact, any base b, where b 7, would work. The instance at the beginning of this subsection is

the set S and target t in Figure 34.19 interpreted in base 7, with S listed in sorted order.

1100 Chapter 34 NP-Completeness

respond to literals with the value 1 in the satisfying assignment. Having included

either i or 0

i

, but not both, for all i, and having put 0 in the digits labeled by

variables in all s j and s 0

j

, we see that for each variable-labeled digit, the sum of the

values of S 0

must be 1, which matches those digits of the target t. Because each

clause is satisﬁed, the clause contains some literal with the value 1. Therefore,

each digit labeled by a clause has at least one 1 contributed to its sum by a i or 0

i

value in S 0

. In fact, 1, 2, or 3 literals may be 1 in each clause, and so each clause-

labeled digit has a sum of 1, 2, or 3 from the i and 0

i

values in S 0

. In Figure 34.19

for example, literals :x 1 , :x 2 , and x 3 have the value 1 in a satisfying assignment.

Each of clauses C 1 and C 4 contains exactly one of these literals, and so together 0

1

,

0

2

, and 3 contribute 1 to the sum in the digits for C 1 and C 4 . Clause C 2 contains

two of these literals, and 0

1

, 0

2

, and 3 contribute 2 to the sum in the digit for C 2 .

Clause C 3 contains all three of these literals, and 0

1

, 0

2

, and 3 contribute 3 to the

sum in the digit for C 3 . We achieve the target of 4 in each digit labeled by clause C j

by including in S 0

the appropriate nonempty subset of slack variables fs j ; s 0

j

g. In

Figure 34.19, S 0

includes s 1 , s 0

1

, s 0

2

, s 3 , s 4 , and s 0

4

. Since we have matched the target

in all digits of the sum, and no carries can occur, the values of S 0

sum to t.

Now, suppose that there is a subset S 0 S that sums to t. The subset S 0

must

include exactly one of i and 0

i

for each i D 1; 2; : : : ; n, for otherwise the digits

labeled by variables would not sum to 1. If i 2 S 0

, we set x i D 1. Otherwise,

0

i

2 S 0

, and we set x i D 0. We claim that every clause C j , for j D 1; 2; : : : ; k, is

satisﬁed by this assignment. To prove this claim, note that to achieve a sum of 4 in

the digit labeled by C j , the subset S 0

must include at least one i or 0

i

value that

has a 1 in the digit labeled by C j , since the contributions of the slack variables s j

and s 0

j

together sum to at most 3. If S 0

includes a i that has a 1 in C j ’s position,

then the literal x i appears in clause C j . Since we have set x i D 1 when i 2 S 0

,

clause C j is satisﬁed. If S 0

includes a 0

i

that has a 1 in that position, then the

literal :x i appears in C j . Since we have set x i D 0 when 0

i

2 S 0

, clause C j is

again satisﬁed. Thus, all clauses of are satisﬁed, which completes the proof.

Exercises

34.5-1

The subgraph-isomorphism problem takes two undirected graphs G 1 and G 2 , and

it asks whether G 1 is isomorphic to a subgraph of G 2 . Show that the subgraph-

isomorphism problem is NP-complete.

34.5-2

Given an integer m n matrix A and an integer m-vector b, the 0-1 integer-

programming problem asks whether there exists an integer n-vector x with ele-

Problems for Chapter 34 1101

ments in the set f0; 1g such that Ax b. Prove that 0-1 integer programming is

NP-complete. (Hint: Reduce from 3-CNF-SAT.)

34.5-3

The integer linear-programming problem is like the 0-1 integer-programming

problem given in Exercise 34.5-2, except that the values of the vector x may be

any integers rather than just 0 or 1. Assuming that the 0-1 integer-programming

problem is NP-hard, show that the integer linear-programming problem is NP-

complete.

34.5-4

Show how to solve the subset-sum problem in polynomial time if the target value t

is expressed in unary.

34.5-5

The set-partition problem takes as input a set S of numbers. The question is

whether the numbers can be partitioned into two sets A and A D S A such

that

P

x2A

x D

P

x2A

x. Show that the set-partition problem is NP-complete.

34.5-6

Show that the hamiltonian-path problem is NP-complete.

34.5-7

The longest-simple-cycle problem is the problem of determining a simple cycle

(no repeated vertices) of maximum length in a graph. Formulate a related decision

problem, and show that the decision problem is NP-complete.

34.5-8

In the half 3-CNF satisﬁability problem, we are given a 3-CNF formula with n

variables and m clauses, where m is even. We wish to determine whether there

exists a truth assignment to the variables of such that exactly half the clauses

evaluate to 0 and exactly half the clauses evaluate to 1. Prove that the half 3-CNF

satisﬁability problem is NP-complete.

Problems

34-1 Independent set

An independent set of a graph G D .V; E/ is a subset V 0 V of vertices such

that each edge in E is incident on at most one vertex in V 0

. The independent-set

problem is to ﬁnd a maximum-size independent set in G.

1102 Chapter 34 NP-Completeness

a. Formulate a related decision problem for the independent-set problem, and

prove that it is NP-complete. (Hint: Reduce from the clique problem.)

b. Suppose that you are given a “black-box” subroutine to solve the decision prob-

lem you deﬁned in part (a). Give an algorithm to ﬁnd an independent set of max-

imum size. The running time of your algorithm should be polynomial in jV j

and jEj, counting queries to the black box as a single step.

Although the independent-set decision problem is NP-complete, certain special

cases are polynomial-time solvable.

c. Give an efﬁcient algorithm to solve the independent-set problem when each ver-

tex in G has degree 2. Analyze the running time, and prove that your algorithm

works correctly.

d. Give an efﬁcient algorithm to solve the independent-set problem when G is

bipartite. Analyze the running time, and prove that your algorithm works cor-

rectly. (Hint: Use the results of Section 26.3.)

34-2 Bonnie and Clyde

Bonnie and Clyde have just robbed a bank. They have a bag of money and want

to divide it up. For each of the following scenarios, either give a polynomial-time

algorithm, or prove that the problem is NP-complete. The input in each case is a

list of the n items in the bag, along with the value of each.

a. The bag contains n coins, but only 2 different denominations: some coins are

worth x dollars, and some are worth y dollars. Bonnie and Clyde wish to divide

the money exactly evenly.

b. The bag contains n coins, with an arbitrary number of different denominations,

but each denomination is a nonnegative integer power of 2, i.e., the possible

denominations are 1 dollar, 2 dollars, 4 dollars, etc. Bonnie and Clyde wish to

divide the money exactly evenly.

c. The bag contains n checks, which are, in an amazing coincidence, made out to

“Bonnie or Clyde.” They wish to divide the checks so that they each get the

exact same amount of money.

d. The bag contains n checks as in part (c), but this time Bonnie and Clyde are

willing to accept a split in which the difference is no larger than 100 dollars.

Problems for Chapter 34 1103

34-3 Graph coloring

Mapmakers try to use as few colors as possible when coloring countries on a map,

as long as no two countries that share a border have the same color. We can model

this problem with an undirected graph G D .V; E/ in which each vertex repre-

sents a country and vertices whose respective countries share a border are adjacent.

Then, a k-coloring is a function c W V ! f1; 2; : : : ; kg such that c.u/ ¤ c./ for

every edge .u; / 2 E. In other words, the numbers 1; 2; : : : ; k represent the k col-

ors, and adjacent vertices must have different colors. The graph-coloring problem

is to determine the minimum number of colors needed to color a given graph.

a. Give an efﬁcient algorithm to determine a 2-coloring of a graph, if one exists.

b. Cast the graph-coloring problem as a decision problem. Show that your deci-

sion problem is solvable in polynomial time if and only if the graph-coloring

problem is solvable in polynomial time.

c. Let the language 3-COLOR be the set of graphs that can be 3-colored. Show

that if 3-COLOR is NP-complete, then your decision problem from part (b) is

NP-complete.

To prove that 3-COLOR is NP-complete, we use a reduction from 3-CNF-SAT.

Given a formula of m clauses on n variables x 1 , x 2 , . . . , x n , we construct a graph

G D .V; E/ as follows. The set V consists of a vertex for each variable, a vertex

for the negation of each variable, 5 vertices for each clause, and 3 special vertices:

TRUE, FALSE, and RED. The edges of the graph are of two types: “literal” edges

that are independent of the clauses and “clause” edges that depend on the clauses.

The literal edges form a triangle on the special vertices and also form a triangle on

x i , :x i , and RED for i D 1; 2; : : : ; n.

d. Argue that in any 3-coloring c of a graph containing the literal edges, exactly

one of a variable and its negation is colored c.TRUE/ and the other is colored

c.FALSE/. Argue that for any truth assignment for , there exists a 3-coloring

of the graph containing just the literal edges.

The widget shown in Figure 34.20 helps to enforce the condition corresponding to

a clause .x \_ y \_ ´/. Each clause requires a unique copy of the 5 vertices that are

heavily shaded in the ﬁgure; they connect as shown to the literals of the clause and

the special vertex TRUE.

e. Argue that if each of x, y, and ´ is colored c.TRUE/ or c.FALSE/, then the

widget is 3-colorable if and only if at least one of x, y, or ´ is colored c.TRUE/.

f. Complete the proof that 3-COLOR is NP-complete.

1104 Chapter 34 NP-Completeness

x

y

z

TRUE

Figure 34.20 The widget corresponding to a clause .x \_ y \_ ´/, used in Problem 34-3.

34-4 Scheduling with proﬁts and deadlines

Suppose that we have one machine and a set of n tasks a 1 ; a 2 ; : : : ; a n , each of

which requires time on the machine. Each task a j requires t j time units on the

machine (its processing time), yields a proﬁt of p j , and has a deadline d j . The

machine can process only one task at a time, and task a j must run without inter-

ruption for t j consecutive time units. If we complete task a j by its deadline d j , we

receive a proﬁt p j , but if we complete it after its deadline, we receive no proﬁt. As

an optimization problem, we are given the processing times, proﬁts, and deadlines

for a set of n tasks, and we wish to ﬁnd a schedule that completes all the tasks and

returns the greatest amount of proﬁt. The processing times, proﬁts, and deadlines

are all nonnegative numbers.

a. State this problem as a decision problem.

b. Show that the decision problem is NP-complete.

c. Give a polynomial-time algorithm for the decision problem, assuming that all

processing times are integers from 1 to n. (Hint: Use dynamic programming.)

d. Give a polynomial-time algorithm for the optimization problem, assuming that

all processing times are integers from 1 to n.

Chapter notes

The book by Garey and Johnson [129] provides a wonderful guide to NP-complete-

ness, discussing the theory at length and providing a catalogue of many problems

that were known to be NP-complete in 1979. The proof of Theorem 34.13 is

adapted from their book, and the list of NP-complete problem domains at the begin-

ning of Section 34.5 is drawn from their table of contents. Johnson wrote a series

Notes for Chapter 34 1105

of 23 columns in the Journal of Algorithms between 1981 and 1992 reporting new

developments in NP-completeness. Hopcroft, Motwani, and Ullman [177], Lewis

and Papadimitriou [236], Papadimitriou [270], and Sipser [317] have good treat-

ments of NP-completeness in the context of complexity theory. NP-completeness

and several reductions also appear in books by Aho, Hopcroft, and Ullman [5];

Dasgupta, Papadimitriou, and Vazirani [82]; Johnsonbaugh and Schaefer [193];

and Kleinberg and Tardos [208].

The class P was introduced in 1964 by Cobham [72] and, independently, in 1965

by Edmonds [100], who also introduced the class NP and conjectured that P ¤ NP.

The notion of NP-completeness was proposed in 1971 by Cook [75], who gave

the ﬁrst NP-completeness proofs for formula satisﬁability and 3-CNF satisﬁabil-

ity. Levin [234] independently discovered the notion, giving an NP-completeness

proof for a tiling problem. Karp [199] introduced the methodology of reductions

in 1972 and demonstrated the rich variety of NP-complete problems. Karp’s pa-

per included the original NP-completeness proofs of the clique, vertex-cover, and

hamiltonian-cycle problems. Since then, thousands of problems have been proven

to be NP-complete by many researchers. In a talk at a meeting celebrating Karp’s

60th birthday in 1995, Papadimitriou remarked, “about 6000 papers each year have

the term ‘NP-complete’ on their title, abstract, or list of keywords. This is more

than each of the terms ‘compiler,’ ‘database,’ ‘expert,’ ‘neural network,’ or ‘oper-

ating system.’ ”

Recent work in complexity theory has shed light on the complexity of computing

approximate solutions. This work gives a new deﬁnition of NP using “probabilis-

tically checkable proofs.” This new deﬁnition implies that for problems such as

clique, vertex cover, the traveling-salesman problem with the triangle inequality,

and many others, computing good approximate solutions is NP-hard and hence no

easier than computing optimal solutions. An introduction to this area can be found

in Arora’s thesis [20]; a chapter by Arora and Lund in Hochbaum [172]; a survey

article by Arora [21]; a book edited by Mayr, Pr¨omel, and Steger [246]; and a

survey article by Johnson [191].

35 Approximation Algorithms

Many problems of practical signiﬁcance are NP-complete, yet they are too impor-

tant to abandon merely because we don’t know how to ﬁnd an optimal solution in

polynomial time. Even if a problem is NP-complete, there may be hope. We have at

least three ways to get around NP-completeness. First, if the actual inputs are small,

an algorithm with exponential running time may be perfectly satisfactory. Second,

we may be able to isolate important special cases that we can solve in polynomial

time. Third, we might come up with approaches to ﬁnd near-optimal solutions in

polynomial time (either in the worst case or the expected case). In practice, near-

optimality is often good enough. We call an algorithm that returns near-optimal

solutions an approximation algorithm. This chapter presents polynomial-time ap-

proximation algorithms for several NP-complete problems.

Performance ratios for approximation algorithms

Suppose that we are working on an optimization problem in which each potential

solution has a positive cost, and we wish to ﬁnd a near-optimal solution. Depending

on the problem, we may deﬁne an optimal solution as one with maximum possi-

ble cost or one with minimum possible cost; that is, the problem may be either a

maximization or a minimization problem.

We say that an algorithm for a problem has an approximation ratio of .n/ if,

for any input of size n, the cost C of the solution produced by the algorithm is

within a factor of .n/ of the cost C

of an optimal solution:

max

C

C

;

C

C

.n/ : (35.1)

If an algorithm achieves an approximation ratio of .n/, we call it a .n/-approx-

imation algorithm. The deﬁnitions of the approximation ratio and of a .n/-

approximation algorithm apply to both minimization and maximization problems.

For a maximization problem, 0 < C C

, and the ratio C =C gives the factor

by which the cost of an optimal solution is larger than the cost of the approximate

Chapter 35 Approximation Algorithms 1107

solution. Similarly, for a minimization problem, 0 < C C , and the ratio C=C

gives the factor by which the cost of the approximate solution is larger than the

cost of an optimal solution. Because we assume that all solutions have positive

cost, these ratios are always well deﬁned. The approximation ratio of an approx-

imation algorithm is never less than 1, since C=C 1 implies C =C 1.

Therefore, a 1-approximation algorithm 1 produces an optimal solution, and an ap-

proximation algorithm with a large approximation ratio may return a solution that

is much worse than optimal.

For many problems, we have polynomial-time approximation algorithms with

small constant approximation ratios, although for other problems, the best known

polynomial-time approximation algorithms have approximation ratios that grow

as functions of the input size n. An example of such a problem is the set-cover

problem presented in Section 35.3.

Some NP-complete problems allow polynomial-time approximation algorithms

that can achieve increasingly better approximation ratios by using more and more

computation time. That is, we can trade computation time for the quality of the

approximation. An example is the subset-sum problem studied in Section 35.5.

This situation is important enough to deserve a name of its own.

An approximation scheme for an optimization problem is an approximation al-

gorithm that takes as input not only an instance of the problem, but also a value

> 0 such that for any ﬁxed , the scheme is a .1 C /-approximation algorithm.

We say that an approximation scheme is a polynomial-time approximation scheme

if for any ﬁxed > 0, the scheme runs in time polynomial in the size n of its input

instance.

The running time of a polynomial-time approximation scheme can increase very

rapidly as decreases. For example, the running time of a polynomial-time ap-

proximation scheme might be O.n 2= /. Ideally, if decreases by a constant factor,

the running time to achieve the desired approximation should not increase by more

than a constant factor (though not necessarily the same constant factor by which

decreased).

We say that an approximation scheme is a fully polynomial-time approximation

scheme if it is an approximation scheme and its running time is polynomial in

both 1= and the size n of the input instance. For example, the scheme might have

a running time of O..1=/ 2 n 3 /. With such a scheme, any constant-factor decrease

in comes with a corresponding constant-factor increase in the running time.

1

When the approximation ratio is independent of n, we use the terms “approximation ratio of ” and

“-approximation algorithm,” indicating no dependence on n.

1108 Chapter 35 Approximation Algorithms

Chapter outline

The ﬁrst four sections of this chapter present some examples of polynomial-time

approximation algorithms for NP-complete problems, and the ﬁfth section presents

a fully polynomial-time approximation scheme. Section 35.1 begins with a study

of the vertex-cover problem, an NP-complete minimization problem that has an

approximation algorithm with an approximation ratio of 2. Section 35.2 presents

an approximation algorithm with an approximation ratio of 2 for the case of the

traveling-salesman problem in which the cost function satisﬁes the triangle in-

equality. It also shows that without the triangle inequality, for any constant 1,

a -approximation algorithm cannot exist unless P D NP. In Section 35.3, we

show how to use a greedy method as an effective approximation algorithm for the

set-covering problem, obtaining a covering whose cost is at worst a logarithmic

factor larger than the optimal cost. Section 35.4 presents two more approximation

algorithms. First we study the optimization version of 3-CNF satisﬁability and

give a simple randomized algorithm that produces a solution with an expected ap-

proximation ratio of 8=7. Then we examine a weighted variant of the vertex-cover

problem and show how to use linear programming to develop a 2-approximation

algorithm. Finally, Section 35.5 presents a fully polynomial-time approximation

scheme for the subset-sum problem.

35.1 The vertex-cover problem

Section 34.5.2 deﬁned the vertex-cover problem and proved it NP-complete. Recall

that a vertex cover of an undirected graph G D .V; E/ is a subset V 0 V such

that if .u; / is an edge of G, then either u 2 V 0

or 2 V 0

(or both). The size of a

vertex cover is the number of vertices in it.

The vertex-cover problem is to ﬁnd a vertex cover of minimum size in a given

undirected graph. We call such a vertex cover an optimal vertex cover. This prob-

lem is the optimization version of an NP-complete decision problem.

Even though we don’t know how to ﬁnd an optimal vertex cover in a graph G

in polynomial time, we can efﬁciently ﬁnd a vertex cover that is near-optimal.

The following approximation algorithm takes as input an undirected graph G and

returns a vertex cover whose size is guaranteed to be no more than twice the size

of an optimal vertex cover.

35.1 The vertex-cover problem 1109

b c d

a e f g

(a)

b c d

a e f g

(b)

b c d

a e f g

(c)

b c d

a e f g

(d)

b c d

a e f g

(e)

b c d

a e f g

(f)

Figure 35.1 The operation of APPROX-VERTEX-COVER. (a) The input graph G, which has 7

vertices and 8 edges. (b) The edge .b; c/, shown heavy, is the ﬁrst edge chosen by APPROX-VERTEX-

COVER. Vertices b and c, shown lightly shaded, are added to the set C containing the vertex cover

being created. Edges .a; b/, .c; e/, and .c; d/, shown dashed, are removed since they are now covered

by some vertex in C . (c) Edge .e; f / is chosen; vertices e and f are added to C . (d) Edge .d; g/

is chosen; vertices d and g are added to C . (e) The set C , which is the vertex cover produced by

APPROX-VERTEX-COVER, contains the six vertices b; c; d; e; f; g. (f) The optimal vertex cover for

this problem contains only three vertices: b, d, and e.

APPROX-VERTEX-COVER.G/

1 C D ;

2 E 0 D G:E

3 while E 0 ¤ ;

4 let .u; / be an arbitrary edge of E 0

5 C D C [ fu; g

6 remove from E 0

every edge incident on either u or

7 return C

Figure 35.1 illustrates how APPROX-VERTEX-COVER operates on an example

graph. The variable C contains the vertex cover being constructed. Line 1 ini-

tializes C to the empty set. Line 2 sets E 0

to be a copy of the edge set G:E of

the graph. The loop of lines 3–6 repeatedly picks an edge .u; / from E 0

, adds its

1110 Chapter 35 Approximation Algorithms

endpoints u and to C , and deletes all edges in E 0

that are covered by either u

or . Finally, line 7 returns the vertex cover C . The running time of this algorithm

is O.V C E/, using adjacency lists to represent E 0

.

Theorem 35.1

APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.

Proof We have already shown that APPROX-VERTEX-COVER runs in polyno-

mial time.

The set C of vertices that is returned by APPROX-VERTEX-COVER is a vertex

cover, since the algorithm loops until every edge in G:E has been covered by some

vertex in C .

To see that APPROX-VERTEX-COVER returns a vertex cover that is at most twice

the size of an optimal cover, let A denote the set of edges that line 4 of APPROX-

VERTEX-COVER picked. In order to cover the edges in A, any vertex cover—in

particular, an optimal cover C

—must include at least one endpoint of each edge

in A. No two edges in A share an endpoint, since once an edge is picked in line 4,

all other edges that are incident on its endpoints are deleted from E 0

in line 6. Thus,

no two edges in A are covered by the same vertex from C

, and we have the lower

bound

jC

j jAj (35.2)

on the size of an optimal vertex cover. Each execution of line 4 picks an edge for

which neither of its endpoints is already in C , yielding an upper bound (an exact

upper bound, in fact) on the size of the vertex cover returned:

jC j D 2 jAj : (35.3)

Combining equations (35.2) and (35.3), we obtain

jC j D 2 jAj

2 jC

j ;

thereby proving the theorem.

Let us reﬂect on this proof. At ﬁrst, you might wonder how we can possibly

prove that the size of the vertex cover returned by APPROX-VERTEX-COVER is at

most twice the size of an optimal vertex cover, when we do not even know the size

of an optimal vertex cover. Instead of requiring that we know the exact size of an

optimal vertex cover, we rely on a lower bound on the size. As Exercise 35.1-2 asks

you to show, the set A of edges that line 4 of APPROX-VERTEX-COVER selects is

actually a maximal matching in the graph G. (A maximal matching is a matching

that is not a proper subset of any other matching.) The size of a maximal matching

35.2 The traveling-salesman problem 1111

is, as we argued in the proof of Theorem 35.1, a lower bound on the size of an

optimal vertex cover. The algorithm returns a vertex cover whose size is at most

twice the size of the maximal matching A. By relating the size of the solution

returned to the lower bound, we obtain our approximation ratio. We will use this

methodology in later sections as well.

Exercises

35.1-1

Give an example of a graph for which APPROX-VERTEX-COVER always yields a

suboptimal solution.

35.1-2

Prove that the set of edges picked in line 4 of APPROX-VERTEX-COVER forms a

maximal matching in the graph G.

35.1-3 ?

Professor B¨undchen proposes the following heuristic to solve the vertex-cover

problem. Repeatedly select a vertex of highest degree, and remove all of its in-

cident edges. Give an example to show that the professor’s heuristic does not have

an approximation ratio of 2. (Hint: Try a bipartite graph with vertices of uniform

degree on the left and vertices of varying degree on the right.)

35.1-4

Give an efﬁcient greedy algorithm that ﬁnds an optimal vertex cover for a tree in

linear time.

35.1-5

From the proof of Theorem 34.12, we know that the vertex-cover problem and the

NP-complete clique problem are complementary in the sense that an optimal vertex

cover is the complement of a maximum-size clique in the complement graph. Does

this relationship imply that there is a polynomial-time approximation algorithm

with a constant approximation ratio for the clique problem? Justify your answer.

35.2 The traveling-salesman problem

In the traveling-salesman problem introduced in Section 34.5.4, we are given a

complete undirected graph G D .V; E/ that has a nonnegative integer cost c.u; /

associated with each edge .u; / 2 E, and we must ﬁnd a hamiltonian cycle (a

tour) of G with minimum cost. As an extension of our notation, let c.A/ denote

the total cost of the edges in the subset A E:

1112 Chapter 35 Approximation Algorithms

c.A/ D

X

.u;/2A

c.u; / :

In many practical situations, the least costly way to go from a place u to a place w

is to go directly, with no intermediate steps. Put another way, cutting out an inter-

mediate stop never increases the cost. We formalize this notion by saying that the

cost function c satisﬁes the triangle inequality if, for all vertices u; ; w 2 V ,

c.u; w/ c.u; / C c.; w/ :

The triangle inequality seems as though it should naturally hold, and it is au-

tomatically satisﬁed in several applications. For example, if the vertices of the

graph are points in the plane and the cost of traveling between two vertices is the

ordinary euclidean distance between them, then the triangle inequality is satisﬁed.

Furthermore, many cost functions other than euclidean distance satisfy the triangle

inequality.

As Exercise 35.2-2 shows, the traveling-salesman problem is NP-complete even

if we require that the cost function satisfy the triangle inequality. Thus, we should

not expect to ﬁnd a polynomial-time algorithm for solving this problem exactly.

Instead, we look for good approximation algorithms.

In Section 35.2.1, we examine a 2-approximation algorithm for the traveling-

salesman problem with the triangle inequality. In Section 35.2.2, we show that

without the triangle inequality, a polynomial-time approximation algorithm with a

constant approximation ratio does not exist unless P D NP.

35.2.1 The traveling-salesman problem with the triangle inequality

Applying the methodology of the previous section, we shall ﬁrst compute a struc-

ture—a minimum spanning tree—whose weight gives a lower bound on the length

of an optimal traveling-salesman tour. We shall then use the minimum spanning

tree to create a tour whose cost is no more than twice that of the minimum spanning

tree’s weight, as long as the cost function satisﬁes the triangle inequality. The fol-

lowing algorithm implements this approach, calling the minimum-spanning-tree

algorithm MST-PRIM from Section 23.2 as a subroutine. The parameter G is a

complete undirected graph, and the cost function c satisﬁes the triangle inequality.

APPROX-TSP-TOUR.G; c/

1 select a vertex r 2 G:V to be a “root” vertex

2 compute a minimum spanning tree T for G from root r

using MST-PRIM.G; c; r/

3 let H be a list of vertices, ordered according to when they are ﬁrst visited

in a preorder tree walk of T

4 return the hamiltonian cycle H

35.2 The traveling-salesman problem 1113

(a)

a d

b f

e

g

c

h

(b)

a d

b f

e

g

c

h

(c)

a d

e

c

h

(d)

a d

b f

e

g

c

h

(e)

b f g

e

h

c

a

b f g

d

Figure 35.2 The operation of APPROX-TSP-TOUR. (a) A complete undirected graph. Vertices lie

on intersections of integer grid lines. For example, f is one unit to the right and two units up from h.

The cost function between two points is the ordinary euclidean distance. (b) A minimum spanning

tree T of the complete graph, as computed by MST-PRIM. Vertex a is the root vertex. Only edges

in the minimum spanning tree are shown. The vertices happen to be labeled in such a way that they

are added to the main tree by MST-PRIM in alphabetical order. (c) A walk of T , starting at a. A

full walk of the tree visits the vertices in the order a; b; c; b; h; b; a; d; e; f; e; g; e; d; a. A preorder

walk of T lists a vertex just when it is ﬁrst encountered, as indicated by the dot next to each vertex,

yielding the ordering a; b; c; h; d; e; f; g. (d) A tour obtained by visiting the vertices in the order

given by the preorder walk, which is the tour H returned by APPROX-TSP-TOUR. Its total cost

is approximately 19:074. (e) An optimal tour H

for the original complete graph. Its total cost is

approximately 14:715.

Recall from Section 12.1 that a preorder tree walk recursively visits every vertex

in the tree, listing a vertex when it is ﬁrst encountered, before visiting any of its

children.

Figure 35.2 illustrates the operation of APPROX-TSP-TOUR. Part (a) of the ﬁg-

ure shows a complete undirected graph, and part (b) shows the minimum spanning

tree T grown from root vertex a by MST-PRIM. Part (c) shows how a preorder

walk of T visits the vertices, and part (d) displays the corresponding tour, which is

the tour returned by APPROX-TSP-TOUR. Part (e) displays an optimal tour, which

is about 23% shorter.

1114 Chapter 35 Approximation Algorithms

By Exercise 23.2-2, even with a simple implementation of MST-PRIM, the run-

ning time of APPROX-TSP-TOUR is ‚.V 2 /. We now show that if the cost function

for an instance of the traveling-salesman problem satisﬁes the triangle inequality,

then APPROX-TSP-TOUR returns a tour whose cost is not more than twice the cost

of an optimal tour.

Theorem 35.2

APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the

traveling-salesman problem with the triangle inequality.

Proof We have already seen that APPROX-TSP-TOUR runs in polynomial time.

Let H

denote an optimal tour for the given set of vertices. We obtain a spanning

tree by deleting any edge from a tour, and each edge cost is nonnegative. Therefore,

the weight of the minimum spanning tree T computed in line 2 of APPROX-TSP-

TOUR provides a lower bound on the cost of an optimal tour:

c.T / c.H

/ : (35.4)

A full walk of T lists the vertices when they are ﬁrst visited and also whenever

they are returned to after a visit to a subtree. Let us call this full walk W . The full

walk of our example gives the order

a; b; c; b; h; b; a; d; e; f; e; g; e; d; a :

Since the full walk traverses every edge of T exactly twice, we have (extending

our deﬁnition of the cost c in the natural manner to handle multisets of edges)

c.W / D 2c.T / : (35.5)

Inequality (35.4) and equation (35.5) imply that

c.W / 2c.H

/ ; (35.6)

and so the cost of W is within a factor of 2 of the cost of an optimal tour.

Unfortunately, the full walk W is generally not a tour, since it visits some ver-

tices more than once. By the triangle inequality, however, we can delete a visit to

any vertex from W and the cost does not increase. (If we delete a vertex from W

between visits to u and w, the resulting ordering speciﬁes going directly from u

to w.) By repeatedly applying this operation, we can remove from W all but the

ﬁrst visit to each vertex. In our example, this leaves the ordering

a; b; c; h; d; e; f; g :

This ordering is the same as that obtained by a preorder walk of the tree T . Let H

be the cycle corresponding to this preorder walk. It is a hamiltonian cycle, since ev-

35.2 The traveling-salesman problem 1115

ery vertex is visited exactly once, and in fact it is the cycle computed by APPROX-

TSP-TOUR. Since H is obtained by deleting vertices from the full walk W , we

have

c.H/ c.W / : (35.7)

Combining inequalities (35.6) and (35.7) gives c.H/ 2c.H /, which completes

the proof.

In spite of the nice approximation ratio provided by Theorem 35.2, APPROX-

TSP-TOUR is usually not the best practical choice for this problem. There are other

approximation algorithms that typically perform much better in practice. (See the

references at the end of this chapter.)

35.2.2 The general traveling-salesman problem

If we drop the assumption that the cost function c satisﬁes the triangle inequality,

then we cannot ﬁnd good approximate tours in polynomial time unless P D NP.

Theorem 35.3

If P ¤ NP, then for any constant 1, there is no polynomial-time approximation

algorithm with approximation ratio for the general traveling-salesman problem.

Proof The proof is by contradiction. Suppose to the contrary that for some num-

ber 1, there is a polynomial-time approximation algorithm A with approx-

imation ratio . Without loss of generality, we assume that is an integer, by

rounding it up if necessary. We shall then show how to use A to solve instances

of the hamiltonian-cycle problem (deﬁned in Section 34.2) in polynomial time.

Since Theorem 34.13 tells us that the hamiltonian-cycle problem is NP-complete,

Theorem 34.4 implies that if we can solve it in polynomial time, then P D NP.

Let G D .V; E/ be an instance of the hamiltonian-cycle problem. We wish to

determine efﬁciently whether G contains a hamiltonian cycle by making use of

the hypothesized approximation algorithm A. We turn G into an instance of the

traveling-salesman problem as follows. Let G 0 D .V; E 0 / be the complete graph

on V ; that is,

E

0

D f.u; / W u; 2 V and u ¤ g :

Assign an integer cost to each edge in E 0

as follows:

c.u; / D

(

1 if .u; / 2 E ;

jV j C 1 otherwise :

We can create representations of G 0

and c from a representation of G in time poly-

nomial in jV j and jEj.

1116 Chapter 35 Approximation Algorithms

Now, consider the traveling-salesman problem .G 0 ; c/. If the original graph G

has a hamiltonian cycle H, then the cost function c assigns to each edge of H a

cost of 1, and so .G 0 ; c/ contains a tour of cost jV j. On the other hand, if G does

not contain a hamiltonian cycle, then any tour of G 0

must use some edge not in E.

But any tour that uses an edge not in E has a cost of at least

. jV j C 1/ C .jV j 1/ D jV j C jV j

> jV j :

Because edges not in G are so costly, there is a gap of at least jV j between the cost

of a tour that is a hamiltonian cycle in G (cost jV j) and the cost of any other tour

(cost at least jV j C jV j). Therefore, the cost of a tour that is not a hamiltonian

cycle in G is at least a factor of C 1 greater than the cost of a tour that is a

hamiltonian cycle in G.

Now, suppose that we apply the approximation algorithm A to the traveling-

salesman problem .G 0 ; c/. Because A is guaranteed to return a tour of cost no

more than times the cost of an optimal tour, if G contains a hamiltonian cycle,

then A must return it. If G has no hamiltonian cycle, then A returns a tour of cost

more than jV j. Therefore, we can use A to solve the hamiltonian-cycle problem

in polynomial time.

The proof of Theorem 35.3 serves as an example of a general technique for

proving that we cannot approximate a problem very well. Suppose that given an

NP-hard problem X, we can produce in polynomial time a minimization prob-

lem Y such that “yes” instances of X correspond to instances of Y with value at

most k (for some k), but that “no” instances of X correspond to instances of Y

with value greater than k. Then, we have shown that, unless P D NP, there is no

polynomial-time -approximation algorithm for problem Y .

Exercises

35.2-1

Suppose that a complete undirected graph G D .V; E/ with at least 3 vertices has

a cost function c that satisﬁes the triangle inequality. Prove that c.u; / 0 for all

u; 2 V .

35.2-2

Show how in polynomial time we can transform one instance of the traveling-

salesman problem into another instance whose cost function satisﬁes the triangle

inequality. The two instances must have the same set of optimal tours. Explain

why such a polynomial-time transformation does not contradict Theorem 35.3, as-

suming that P ¤ NP.

35.3 The set-covering problem 1117

35.2-3

Consider the following closest-point heuristic for building an approximate trav-

eling-salesman tour whose cost function satisﬁes the triangle inequality. Begin

with a trivial cycle consisting of a single arbitrarily chosen vertex. At each step,

identify the vertex u that is not on the cycle but whose distance to any vertex on the

cycle is minimum. Suppose that the vertex on the cycle that is nearest u is vertex .

Extend the cycle to include u by inserting u just after . Repeat until all vertices

are on the cycle. Prove that this heuristic returns a tour whose total cost is not more

than twice the cost of an optimal tour.

35.2-4

In the bottleneck traveling-salesman problem, we wish to ﬁnd the hamiltonian cy-

cle that minimizes the cost of the most costly edge in the cycle. Assuming that the

cost function satisﬁes the triangle inequality, show that there exists a polynomial-

time approximation algorithm with approximation ratio 3 for this problem. (Hint:

Show recursively that we can visit all the nodes in a bottleneck spanning tree, as

discussed in Problem 23-3, exactly once by taking a full walk of the tree and skip-

ping nodes, but without skipping more than two consecutive intermediate nodes.

Show that the costliest edge in a bottleneck spanning tree has a cost that is at most

the cost of the costliest edge in a bottleneck hamiltonian cycle.)

35.2-5

Suppose that the vertices for an instance of the traveling-salesman problem are

points in the plane and that the cost c.u; / is the euclidean distance between

points u and . Show that an optimal tour never crosses itself.

35.3 The set-covering problem

The set-covering problem is an optimization problem that models many problems

that require resources to be allocated. Its corresponding decision problem general-

izes the NP-complete vertex-cover problem and is therefore also NP-hard. The ap-

proximation algorithm developed to handle the vertex-cover problem doesn’t apply

here, however, and so we need to try other approaches. We shall examine a simple

greedy heuristic with a logarithmic approximation ratio. That is, as the size of the

instance gets larger, the size of the approximate solution may grow, relative to the

size of an optimal solution. Because the logarithm function grows rather slowly,

however, this approximation algorithm may nonetheless give useful results.

1118 Chapter 35 Approximation Algorithms

S

3

S

6

S

4

S

5

S

2

S

1

Figure 35.3 An instance .X; F / of the set-covering problem, where X consists of the 12 black

points and F D fS1; S2; S3; S4; S5; S6g. A minimum-size set cover is C D fS3; S4; S5g, with

size 3. The greedy algorithm produces a cover of size 4 by selecting either the sets S1, S4, S5,

and S3 or the sets S1, S4, S5, and S6, in order.

An instance .X; F / of the set-covering problem consists of a ﬁnite set X and

a family F of subsets of X, such that every element of X belongs to at least one

subset in F :

X D

[

S2F

S :

We say that a subset S 2 F covers its elements. The problem is to ﬁnd a minimum-

size subset C F whose members cover all of X:

X D

[

S2C

S : (35.8)

We say that any C satisfying equation (35.8) covers X. Figure 35.3 illustrates the

set-covering problem. The size of C is the number of sets it contains, rather than

the number of individual elements in these sets, since every subset C that covers X

must contain all jXj individual elements. In Figure 35.3, the minimum set cover

has size 3.

The set-covering problem abstracts many commonly arising combinatorial prob-

lems. As a simple example, suppose that X represents a set of skills that are needed

to solve a problem and that we have a given set of people available to work on the

problem. We wish to form a committee, containing as few people as possible,

such that for every requisite skill in X, at least one member of the committee has

that skill. In the decision version of the set-covering problem, we ask whether a

covering exists with size at most k, where k is an additional parameter speciﬁed

in the problem instance. The decision version of the problem is NP-complete, as

Exercise 35.3-2 asks you to show.

35.3 The set-covering problem 1119

A greedy approximation algorithm

The greedy method works by picking, at each stage, the set S that covers the great-

est number of remaining elements that are uncovered.

GREEDY-SET-COVER.X; F /

1 U D X

2 C D ;

3 while U ¤ ;

4 select an S 2 F that maximizes jS \ U j

5 U D U S

6 C D C [ fSg

7 return C

In the example of Figure 35.3, GREEDY-SET-COVER adds to C , in order, the sets

S 1 , S 4 , and S 5 , followed by either S 3 or S 6 .

The algorithm works as follows. The set U contains, at each stage, the set of

remaining uncovered elements. The set C contains the cover being constructed.

Line 4 is the greedy decision-making step, choosing a subset S that covers as many

uncovered elements as possible (breaking ties arbitrarily). After S is selected,

line 5 removes its elements from U , and line 6 places S into C . When the algorithm

terminates, the set C contains a subfamily of F that covers X.

We can easily implement GREEDY-SET-COVER to run in time polynomial in jXj

and j F j. Since the number of iterations of the loop on lines 3–6 is bounded from

above by min.jXj ; j F j/, and we can implement the loop body to run in time

O.jXj j F j/, a simple implementation runs in time O.jXj j F j min.jXj ; j F j//. Ex-

ercise 35.3-3 asks for a linear-time algorithm.

Analysis

We now show that the greedy algorithm returns a set cover that is not too much

larger than an optimal set cover. For convenience, in this chapter we denote the dth

harmonic number H d D

P d

iD1

1=i (see Section A.1) by H.d/. As a boundary

condition, we deﬁne H.0/ D 0.

Theorem 35.4

GREEDY-SET-COVER is a polynomial-time .n/-approximation algorithm, where

.n/ D H.max fjSj W S 2 F g/ :

Proof We have already shown that GREEDY-SET-COVER runs in polynomial

time.

1120 Chapter 35 Approximation Algorithms

To show that GREEDY-SET-COVER is a .n/-approximation algorithm, we as-

sign a cost of 1 to each set selected by the algorithm, distribute this cost over

the elements covered for the ﬁrst time, and then use these costs to derive the de-

sired relationship between the size of an optimal set cover C

and the size of the

set cover C returned by the algorithm. Let S i denote the ith subset selected by

GREEDY-SET-COVER; the algorithm incurs a cost of 1 when it adds S i to C . We

spread this cost of selecting S i evenly among the elements covered for the ﬁrst time

by S i . Let c x denote the cost allocated to element x, for each x 2 X. Each element

is assigned a cost only once, when it is covered for the ﬁrst time. If x is covered

for the ﬁrst time by S i , then

c x D

1

jS i .S 1 [ S 2 [ [ S i1 /j

:

Each step of the algorithm assigns 1 unit of cost, and so

j C j D

X

x2X

c x : (35.9)

Each element x 2 X is in at least one set in the optimal cover C

, and so we have

X

S2C

X

x2S

c x

X

x2X

c x : (35.10)

Combining equation (35.9) and inequality (35.10), we have that

j C j

X

S2C

X

x2S

c x : (35.11)

The remainder of the proof rests on the following key inequality, which we shall

prove shortly. For any set S belonging to the family F ,

X

x2S

c x H.jSj/ : (35.12)

From inequalities (35.11) and (35.12), it follows that

j C j

X

S2C

H.jSj/

j C

j H.max fjSj W S 2 F g/ ;

thus proving the theorem.

All that remains is to prove inequality (35.12). Consider any set S 2 F and any

i D 1; 2; : : : ; j C j, and let

u i D jS .S 1 [ S 2 [ [ S i /j

be the number of elements in S that remain uncovered after the algorithm has

selected sets S 1 ; S 2 ; : : : ; S i . We deﬁne u 0 D jSj to be the number of elements

35.3 The set-covering problem 1121

of S, which are all initially uncovered. Let k be the least index such that u k D 0,

so that every element in S is covered by at least one of the sets S 1 ; S 2 ; : : : ; S k and

some element in S is uncovered by S 1 [ S 2 [ [ S k1 . Then, u i1 u i , and

u i1 u i elements of S are covered for the ﬁrst time by S i , for i D 1; 2; : : : ; k.

Thus,

X

x2S

c x D

k X

iD1

.u i1 u i /

1

jS i .S 1 [ S 2 [ [ S i1 /j

:

Observe that

jS i .S 1 [ S 2 [ [ S i1 /j jS .S 1 [ S 2 [ [ S i1 /j

D u i1 ;

because the greedy choice of S i guarantees that S cannot cover more new ele-

ments than S i does (otherwise, the algorithm would have chosen S instead of S i ).

Consequently, we obtain

X

x2S

c x

k X

iD1

.u i1 u i /

1

u i1

:

We now bound this quantity as follows:

X

x2S

c x

k X

iD1

.u i1 u i /

1

u i1

D

k X

iD1

u i1 X

j Du i C1

1

u i1

k X

iD1

u i1 X

j Du i C1

1

j

(because j u i1 )

D

k X

iD1

u i1 X

j D1

1

j

u i X

j D1

1

j

!

D

k X

iD1

.H.u i1 / H.u i //

D H.u 0 / H.u k / (because the sum telescopes)

D H.u 0 / H.0/

D H.u 0 / (because H.0/ D 0)

D H.jSj/ ;

which completes the proof of inequality (35.12).

1122 Chapter 35 Approximation Algorithms

Corollary 35.5

GREEDY-SET-COVER is a polynomial-time .ln jXjC1/-approximation algorithm.

Proof Use inequality (A.14) and Theorem 35.4.

In some applications, max fjSj W S 2 F g is a small constant, and so the solution

returned by GREEDY-SET-COVER is at most a small constant times larger than

optimal. One such application occurs when this heuristic ﬁnds an approximate

vertex cover for a graph whose vertices have degree at most 3. In this case, the

solution found by GREEDY-SET-COVER is not more than H.3/ D 11=6 times as

large as an optimal solution, a performance guarantee that is slightly better than

that of APPROX-VERTEX-COVER.

Exercises

35.3-1

Consider each of the following words as a set of letters: farid; dash; drain;

heard; lost; nose; shun; slate; snare; threadg. Show which set cover

GREEDY-SET-COVER produces when we break ties in favor of the word that ap-

pears ﬁrst in the dictionary.

35.3-2

Show that the decision version of the set-covering problem is NP-complete by

reducing it from the vertex-cover problem.

35.3-3

Show how to implement GREEDY-SET-COVER in such a way that it runs in time

O

P

S2F

jSj

.

35.3-4

Show that the following weaker form of Theorem 35.4 is trivially true:

j C j j C

j max fjSj W S 2 F g :

35.3-5

GREEDY-SET-COVER can return a number of different solutions, depending on

how we break ties in line 4. Give a procedure BAD-SET-COVER-INSTANCE.n/

that returns an n-element instance of the set-covering problem for which, depend-

ing on how we break ties in line 4, GREEDY-SET-COVER can return a number of

different solutions that is exponential in n.

35.4 Randomization and linear programming 1123

35.4 Randomization and linear programming

In this section, we study two useful techniques for designing approximation algo-

rithms: randomization and linear programming. We shall give a simple randomized

algorithm for an optimization version of 3-CNF satisﬁability, and then we shall use

linear programming to help design an approximation algorithm for a weighted ver-

sion of the vertex-cover problem. This section only scratches the surface of these

two powerful techniques. The chapter notes give references for further study of

these areas.

A randomized approximation algorithm for MAX-3-CNF satisﬁability

Just as some randomized algorithms compute exact solutions, some randomized

algorithms compute approximate solutions. We say that a randomized algorithm

for a problem has an approximation ratio of .n/ if, for any input of size n, the

expected cost C of the solution produced by the randomized algorithm is within a

factor of .n/ of the cost C

of an optimal solution:

max

C

C

;

C

C

.n/ : (35.13)

We call a randomized algorithm that achieves an approximation ratio of .n/ a

randomized .n/-approximation algorithm. In other words, a randomized ap-

proximation algorithm is like a deterministic approximation algorithm, except that

the approximation ratio is for an expected cost.

A particular instance of 3-CNF satisﬁability, as deﬁned in Section 34.4, may or

may not be satisﬁable. In order to be satisﬁable, there must exist an assignment of

the variables so that every clause evaluates to 1. If an instance is not satisﬁable, we

may want to compute how “close” to satisﬁable it is, that is, we may wish to ﬁnd an

assignment of the variables that satisﬁes as many clauses as possible. We call the

resulting maximization problem MAX-3-CNF satisﬁability. The input to MAX-3-

CNF satisﬁability is the same as for 3-CNF satisﬁability, and the goal is to return

an assignment of the variables that maximizes the number of clauses evaluating

to 1. We now show that randomly setting each variable to 1 with probability 1=2

and to 0 with probability 1=2 yields a randomized 8=7-approximation algorithm.

According to the deﬁnition of 3-CNF satisﬁability from Section 34.4, we require

each clause to consist of exactly three distinct literals. We further assume that

no clause contains both a variable and its negation. (Exercise 35.4-1 asks you to

remove this last assumption.)

1124 Chapter 35 Approximation Algorithms

Theorem 35.6

Given an instance of MAX-3-CNF satisﬁability with n variables x 1 ; x 2 ; : : : ; x n

and m clauses, the randomized algorithm that independently sets each vari-

able to 1 with probability 1=2 and to 0 with probability 1=2 is a randomized

8=7-approximation algorithm.

Proof Suppose that we have independently set each variable to 1 with probabil-

ity 1=2 and to 0 with probability 1=2. For i D 1; 2; : : : ; m, we deﬁne the indicator

random variable

Y i D I fclause i is satisﬁedg ;

so that Y i D 1 as long as we have set at least one of the literals in the ith clause

to 1. Since no literal appears more than once in the same clause, and since we have

assumed that no variable and its negation appear in the same clause, the settings of

the three literals in each clause are independent. A clause is not satisﬁed only if all

three of its literals are set to 0, and so Pr fclause i is not satisﬁedg D .1=2/ 3 D 1=8.

Thus, we have Pr fclause i is satisﬁedg D 1 1=8 D 7=8, and by Lemma 5.1,

we have E ŒY i D 7=8. Let Y be the number of satisﬁed clauses overall, so that

Y D Y 1 C Y 2 C C Y m . Then, we have

E ŒY D E

"

m X

iD1

Y i

#

D

m X

iD1

E ŒY i (by linearity of expectation)

D

m X

iD1

7=8

D 7m=8 :

Clearly, m is an upper bound on the number of satisﬁed clauses, and hence the

approximation ratio is at most m=.7m=8/ D 8=7.

Approximating weighted vertex cover using linear programming

In the minimum-weight vertex-cover problem, we are given an undirected graph

G D .V; E/ in which each vertex 2 V has an associated positive weight w./.

For any vertex cover V 0 V , we deﬁne the weight of the vertex cover w.V 0 / D P

2V 0 w./. The goal is to ﬁnd a vertex cover of minimum weight.

We cannot apply the algorithm used for unweighted vertex cover, nor can we use

a random solution; both methods may return solutions that are far from optimal.

We shall, however, compute a lower bound on the weight of the minimum-weight

35.4 Randomization and linear programming 1125

vertex cover, by using a linear program. We shall then “round” this solution and

use it to obtain a vertex cover.

Suppose that we associate a variable x./ with each vertex 2 V , and let us

require that x./ equals either 0 or 1 for each 2 V . We put into the vertex cover

if and only if x./ D 1. Then, we can write the constraint that for any edge .u; /,

at least one of u and must be in the vertex cover as x.u/ C x./ 1. This view

gives rise to the following 0-1 integer program for ﬁnding a minimum-weight

vertex cover:

minimize

X

2V

w./ x./ (35.14)

subject to

x.u/ C x./ 1 for each .u; / 2 E (35.15)

x./ 2 f0; 1g for each 2 V : (35.16)

In the special case in which all the weights w./ are equal to 1, this formu-

lation is the optimization version of the NP-hard vertex-cover problem. Sup-

pose, however, that we remove the constraint that x./ 2 f0; 1g and replace it

by 0 x./ 1. We then obtain the following linear program, which is known as

the linear-programming relaxation:

minimize

X

2V

w./ x./ (35.17)

subject to

x.u/ C x./ 1 for each .u; / 2 E (35.18)

x./ 1 for each 2 V (35.19)

x./ 0 for each 2 V : (35.20)

Any feasible solution to the 0-1 integer program in lines (35.14)–(35.16) is also

a feasible solution to the linear program in lines (35.17)–(35.20). Therefore, the

value of an optimal solution to the linear program gives a lower bound on the value

of an optimal solution to the 0-1 integer program, and hence a lower bound on the

optimal weight in the minimum-weight vertex-cover problem.

The following procedure uses the solution to the linear-programming relaxation

to construct an approximate solution to the minimum-weight vertex-cover problem:

1126 Chapter 35 Approximation Algorithms

APPROX-MIN-WEIGHT-VC.G; w/

1 C D ;

2 compute Nx, an optimal solution to the linear program in lines (35.17)–(35.20)

3 for each 2 V

4 if Nx./ 1=2

5 C D C [ fg

6 return C

The APPROX-MIN-WEIGHT-VC procedure works as follows. Line 1 initial-

izes the vertex cover to be empty. Line 2 formulates the linear program in

lines (35.17)–(35.20) and then solves this linear program. An optimal solution

gives each vertex an associated value Nx./, where 0 Nx./ 1. We use this

value to guide the choice of which vertices to add to the vertex cover C in lines 3–5.

If Nx./ 1=2, we add to C ; otherwise we do not. In effect, we are “rounding”

each fractional variable in the solution to the linear program to 0 or 1 in order to

obtain a solution to the 0-1 integer program in lines (35.14)–(35.16). Finally, line 6

returns the vertex cover C .

Theorem 35.7

Algorithm APPROX-MIN-WEIGHT-VC is a polynomial-time 2-approximation al-

gorithm for the minimum-weight vertex-cover problem.

Proof Because there is a polynomial-time algorithm to solve the linear program

in line 2, and because the for loop of lines 3–5 runs in polynomial time, APPROX-

MIN-WEIGHT-VC is a polynomial-time algorithm.

Now we show that APPROX-MIN-WEIGHT-VC is a 2-approximation algo-

rithm. Let C

be an optimal solution to the minimum-weight vertex-cover prob-

lem, and let ´

be the value of an optimal solution to the linear program in

lines (35.17)–(35.20). Since an optimal vertex cover is a feasible solution to the

linear program, ´

must be a lower bound on w.C /, that is,

´

w.C

/ : (35.21)

Next, we claim that by rounding the fractional values of the variables Nx./, we

produce a set C that is a vertex cover and satisﬁes w.C / 2´

. To see that C is

a vertex cover, consider any edge .u; / 2 E. By constraint (35.18), we know that

x.u/ C x./ 1, which implies that at least one of Nx.u/ and Nx./ is at least 1=2.

Therefore, at least one of u and is included in the vertex cover, and so every edge

is covered.

Now, we consider the weight of the cover. We have

35.4 Randomization and linear programming 1127

´

D

X

2V

w./ Nx./

X

2V W Nx./1=2

w./ Nx./

X

2V W Nx./1=2

w./

1

2

D

X

2C

w./

1

2

D

1

2

X

2C

w./

D

1

2

w.C / : (35.22)

Combining inequalities (35.21) and (35.22) gives

w.C / 2´

2w.C

/ ;

and hence APPROX-MIN-WEIGHT-VC is a 2-approximation algorithm.

Exercises

35.4-1

Show that even if we allow a clause to contain both a variable and its negation, ran-

domly setting each variable to 1 with probability 1=2 and to 0 with probability 1=2

still yields a randomized 8=7-approximation algorithm.

35.4-2

The MAX-CNF satisﬁability problem is like the MAX-3-CNF satisﬁability prob-

lem, except that it does not restrict each clause to have exactly 3 literals. Give a

randomized 2-approximation algorithm for the MAX-CNF satisﬁability problem.

35.4-3

In the MAX-CUT problem, we are given an unweighted undirected graph G D

.V; E/. We deﬁne a cut .S; V S/ as in Chapter 23 and the weight of a cut as the

number of edges crossing the cut. The goal is to ﬁnd a cut of maximum weight.

Suppose that for each vertex , we randomly and independently place in S with

probability 1=2 and in V S with probability 1=2. Show that this algorithm is a

randomized 2-approximation algorithm.

1128 Chapter 35 Approximation Algorithms

35.4-4

Show that the constraints in line (35.19) are redundant in the sense that if we re-

move them from the linear program in lines (35.17)–(35.20), any optimal solution

to the resulting linear program must satisfy x./ 1 for each 2 V .

35.5 The subset-sum problem

Recall from Section 34.5.5 that an instance of the subset-sum problem is a

pair .S; t/, where S is a set fx 1 ; x 2 ; : : : ; x n g of positive integers and t is a posi-

tive integer. This decision problem asks whether there exists a subset of S that

adds up exactly to the target value t. As we saw in Section 34.5.5, this problem is

NP-complete.

The optimization problem associated with this decision problem arises in prac-

tical applications. In the optimization problem, we wish to ﬁnd a subset of

fx 1 ; x 2 ; : : : ; x n g whose sum is as large as possible but not larger than t. For ex-

ample, we may have a truck that can carry no more than t pounds, and n different

boxes to ship, the ith of which weighs x i pounds. We wish to ﬁll the truck with as

heavy a load as possible without exceeding the given weight limit.

In this section, we present an exponential-time algorithm that computes the op-

timal value for this optimization problem, and then we show how to modify the

algorithm so that it becomes a fully polynomial-time approximation scheme. (Re-

call that a fully polynomial-time approximation scheme has a running time that is

polynomial in 1= as well as in the size of the input.)

An exponential-time exact algorithm

Suppose that we computed, for each subset S 0

of S, the sum of the elements

in S 0

, and then we selected, among the subsets whose sum does not exceed t,

the one whose sum was closest to t. Clearly this algorithm would return the op-

timal solution, but it could take exponential time. To implement this algorithm,

we could use an iterative procedure that, in iteration i, computes the sums of

all subsets of fx 1 ; x 2 ; : : : ; x i g, using as a starting point the sums of all subsets

of fx 1 ; x 2 ; : : : ; x i1 g. In doing so, we would realize that once a particular subset S 0

had a sum exceeding t, there would be no reason to maintain it, since no super-

set of S 0

could be the optimal solution. We now give an implementation of this

strategy.

The procedure EXACT-SUBSET-SUM takes an input set S D fx 1 ; x 2 ; : : : ; x n g

and a target value t; we’ll see its pseudocode in a moment. This procedure it-

35.5 The subset-sum problem 1129

eratively computes L i , the list of sums of all subsets of fx 1 ; : : : ; x i g that do not

exceed t, and then it returns the maximum value in L n .

If L is a list of positive integers and x is another positive integer, then we let

L C x denote the list of integers derived from L by increasing each element of L

by x. For example, if L D h1; 2; 3; 5; 9i, then L C 2 D h3; 4; 5; 7; 11i. We also use

this notation for sets, so that

S C x D fs C x W s 2 Sg :

We also use an auxiliary procedure MERGE-LISTS.L; L 0 /, which returns the

sorted list that is the merge of its two sorted input lists L and L 0

with duplicate

values removed. Like the MERGE procedure we used in merge sort (Section 2.3.1),

MERGE-LISTS runs in time O.jLj C jL 0 j/. We omit the pseudocode for MERGE-

LISTS.

EXACT-SUBSET-SUM.S; t/

1 n D jSj

2 L 0 D h0i

3 for i D 1 to n

4 L i D MERGE-LISTS.L i1 ; L i1 C x i /

5 remove from L i every element that is greater than t

6 return the largest element in L n

To see how EXACT-SUBSET-SUM works, let P i denote the set of all values

obtained by selecting a (possibly empty) subset of fx 1 ; x 2 ; : : : ; x i g and summing

its members. For example, if S D f1; 4; 5g, then

P 1 D f0; 1g ;

P 2 D f0; 1; 4; 5g ;

P 3 D f0; 1; 4; 5; 6; 9; 10g :

Given the identity

P i D P i1 [ .P i1 C x i / ; (35.23)

we can prove by induction on i (see Exercise 35.5-1) that the list L i is a sorted list

containing every element of P i whose value is not more than t. Since the length

of L i can be as much as 2 i

, EXACT-SUBSET-SUM is an exponential-time algorithm

in general, although it is a polynomial-time algorithm in the special cases in which t

is polynomial in jSj or all the numbers in S are bounded by a polynomial in jSj.

A fully polynomial-time approximation scheme

We can derive a fully polynomial-time approximation scheme for the subset-sum

problem by “trimming” each list L i after it is created. The idea behind trimming is

1130 Chapter 35 Approximation Algorithms

that if two values in L are close to each other, then since we want just an approxi-

mate solution, we do not need to maintain both of them explicitly. More precisely,

we use a trimming parameter ı such that 0 < ı < 1. When we trim a list L by ı,

we remove as many elements from L as possible, in such a way that if L 0

is the

result of trimming L, then for every element y that was removed from L, there is

an element ´ still in L 0

that approximates y, that is,

y

1 C ı

´ y : (35.24)

We can think of such a ´ as “representing” y in the new list L 0

. Each removed

element y is represented by a remaining element ´ satisfying inequality (35.24).

For example, if ı D 0:1 and

L D h10; 11; 12; 15; 20; 21; 22; 23; 24; 29i ;

then we can trim L to obtain

L

0

D h10; 12; 15; 20; 23; 29i ;

where the deleted value 11 is represented by 10, the deleted values 21 and 22

are represented by 20, and the deleted value 24 is represented by 23. Because

every element of the trimmed version of the list is also an element of the original

version of the list, trimming can dramatically decrease the number of elements kept

while keeping a close (and slightly smaller) representative value in the list for each

deleted element.

The following procedure trims list L D hy 1 ; y 2 ; : : : ; y m i in time ‚.m/, given L

and ı, and assuming that L is sorted into monotonically increasing order. The

output of the procedure is a trimmed, sorted list.

TRIM.L; ı/

1 let m be the length of L

2 L 0 D hy 1 i

3 last D y 1

4 for i D 2 to m

5 if y i > last .1 C ı/ // y i last because L is sorted

6 append y i onto the end of L 0

7 last D y i

8 return L 0

The procedure scans the elements of L in monotonically increasing order. A num-

ber is appended onto the returned list L 0

only if it is the ﬁrst element of L or if it

cannot be represented by the most recent number placed into L 0

.

Given the procedure TRIM, we can construct our approximation scheme as fol-

lows. This procedure takes as input a set S D fx 1 ; x 2 ; : : : ; x n g of n integers (in

arbitrary order), a target integer t, and an “approximation parameter” , where

35.5 The subset-sum problem 1131

0 < < 1 : (35.25)

It returns a value ´ whose value is within a 1 C factor of the optimal solution.

APPROX-SUBSET-SUM.S; t; /

1 n D jSj

2 L 0 D h0i

3 for i D 1 to n

4 L i D MERGE-LISTS.L i1 ; L i1 C x i /

5 L i D TRIM.L i ; =2n/

6 remove from L i every element that is greater than t

7 let ´

be the largest value in L n

8 return ´

Line 2 initializes the list L 0 to be the list containing just the element 0. The for

loop in lines 3–6 computes L i as a sorted list containing a suitably trimmed ver-

sion of the set P i , with all elements larger than t removed. Since we create L i

from L i1 , we must ensure that the repeated trimming doesn’t introduce too much

compounded inaccuracy. In a moment, we shall see that APPROX-SUBSET-SUM

returns a correct approximation if one exists.

As an example, suppose we have the instance

S D h104; 102; 201; 101i

with t D 308 and D 0:40. The trimming parameter ı is =8 D 0:05. APPROX-

SUBSET-SUM computes the following values on the indicated lines:

line 2: L 0 D h0i ;

line 4: L 1 D h0; 104i ;

line 5: L 1 D h0; 104i ;

line 6: L 1 D h0; 104i ;

line 4: L 2 D h0; 102; 104; 206i ;

line 5: L 2 D h0; 102; 206i ;

line 6: L 2 D h0; 102; 206i ;

line 4: L 3 D h0; 102; 201; 206; 303; 407i ;

line 5: L 3 D h0; 102; 201; 303; 407i ;

line 6: L 3 D h0; 102; 201; 303i ;

line 4: L 4 D h0; 101; 102; 201; 203; 302; 303; 404i ;

line 5: L 4 D h0; 101; 201; 302; 404i ;

line 6: L 4 D h0; 101; 201; 302i :

1132 Chapter 35 Approximation Algorithms

The algorithm returns ´ D 302 as its answer, which is well within D 40% of

the optimal answer 307 D 104 C 102 C 101; in fact, it is within 2%.

Theorem 35.8

APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme for the

subset-sum problem.

Proof The operations of trimming L i in line 5 and removing from L i every ele-

ment that is greater than t maintain the property that every element of L i is also a

member of P i . Therefore, the value ´

returned in line 8 is indeed the sum of some

subset of S. Let y 2 P n denote an optimal solution to the subset-sum problem.

Then, from line 6, we know that ´ y

. By inequality (35.1), we need to show

that y =´ 1 C . We must also show that the running time of this algorithm is

polynomial in both 1= and the size of the input.

As Exercise 35.5-2 asks you to show, for every element y in P i that is at most t,

there exists an element ´ 2 L i such that

y

.1 C =2n/ i

´ y : (35.26)

Inequality (35.26) must hold for y 2 P n , and therefore there exists an element

´ 2 L n such that

y

.1 C =2n/ n

´ y

;

and thus

y

´

1 C

2n

n

: (35.27)

Since there exists an element ´ 2 L n fulﬁlling inequality (35.27), the inequality

must hold for ´

, which is the largest value in L n ; that is,

y

´

1 C

2n

n

: (35.28)

Now, we show that y =´ 1 C . We do so by showing that .1 C =2n/

n

1 C . By equation (3.14), we have lim n!1 .1 C =2n/ n D e =2

. Exercise 35.5-3

asks you to show that

d

dn

1 C

2n

n

> 0 : (35.29)

Therefore, the function .1 C =2n/

n

increases with n as it approaches its limit

of e =2

, and we have

35.5 The subset-sum problem 1133

1 C

2n

n

e

=2

1 C =2 C .=2/

2

(by inequality (3.13))

1 C (by inequality (35.25)) . (35.30)

Combining inequalities (35.28) and (35.30) completes the analysis of the approxi-

mation ratio.

To show that APPROX-SUBSET-SUM is a fully polynomial-time approximation

scheme, we derive a bound on the length of L i . After trimming, successive ele-

ments ´ and ´ 0

of L i must have the relationship ´ 0 =´ > 1C=2n. That is, they must

differ by a factor of at least 1 C =2n. Each list, therefore, contains the value 0,

possibly the value 1, and up to

log 1C=2n t

˘

additional values. The number of

elements in each list L i is at most

log 1C=2n t C 2 D

ln t

ln.1 C =2n/

C 2

2n.1 C =2n/ ln t

C 2 (by inequality (3.17))

<

3n ln t

C 2 (by inequality (35.25)) .

This bound is polynomial in the size of the input—which is the number of bits lg t

needed to represent t plus the number of bits needed to represent the set S, which is

in turn polynomial in n—and in 1=. Since the running time of APPROX-SUBSET-

SUM is polynomial in the lengths of the L i , we conclude that APPROX-SUBSET-

SUM is a fully polynomial-time approximation scheme.

Exercises

35.5-1

Prove equation (35.23). Then show that after executing line 5 of EXACT-SUBSET-

SUM, L i is a sorted list containing every element of P i whose value is not more

than t.

35.5-2

Using induction on i, prove inequality (35.26).

35.5-3

Prove inequality (35.29).

1134 Chapter 35 Approximation Algorithms

35.5-4

How would you modify the approximation scheme presented in this section to ﬁnd

a good approximation to the smallest value not less than t that is a sum of some

subset of the given input list?

35.5-5

Modify the APPROX-SUBSET-SUM procedure to also return the subset of S that

sums to the value ´

.

Problems

35-1 Bin packing

Suppose that we are given a set of n objects, where the size s i of the ith object

satisﬁes 0 < s i < 1. We wish to pack all the objects into the minimum number of

unit-size bins. Each bin can hold any subset of the objects whose total size does

not exceed 1.

a. Prove that the problem of determining the minimum number of bins required is

NP-hard. (Hint: Reduce from the subset-sum problem.)

The ﬁrst-ﬁt heuristic takes each object in turn and places it into the ﬁrst bin that

can accommodate it. Let S D

P n

iD1

s i .

b. Argue that the optimal number of bins required is at least dSe.

c. Argue that the ﬁrst-ﬁt heuristic leaves at most one bin less than half full.

d. Prove that the number of bins used by the ﬁrst-ﬁt heuristic is never more

than d2Se.

e. Prove an approximation ratio of 2 for the ﬁrst-ﬁt heuristic.

f. Give an efﬁcient implementation of the ﬁrst-ﬁt heuristic, and analyze its running

time.

35-2 Approximating the size of a maximum clique

Let G D .V; E/ be an undirected graph. For any k 1, deﬁne G .k/

to be the undi-

rected graph .V .k/ ; E .k/ /, where V .k/

is the set of all ordered k-tuples of vertices

from V and E .k/

is deﬁned so that . 1 ; 2 ; : : : ; k / is adjacent to .w 1 ; w 2 ; : : : ; w k /

if and only if for i D 1; 2; : : : ; k, either vertex i is adjacent to w i in G, or else

i D w i .

Problems for Chapter 35 1135

a. Prove that the size of the maximum clique in G .k/

is equal to the kth power of

the size of the maximum clique in G.

b. Argue that if there is an approximation algorithm that has a constant approxi-

mation ratio for ﬁnding a maximum-size clique, then there is a polynomial-time

approximation scheme for the problem.

35-3 Weighted set-covering problem

Suppose that we generalize the set-covering problem so that each set S i in the

family F has an associated weight w i and the weight of a cover C is

P

S i 2C

w i .

We wish to determine a minimum-weight cover. (Section 35.3 handles the case in

which w i D 1 for all i.)

Show how to generalize the greedy set-covering heuristic in a natural manner

to provide an approximate solution for any instance of the weighted set-covering

problem. Show that your heuristic has an approximation ratio of H.d/, where d is

the maximum size of any set S i .

35-4 Maximum matching

Recall that for an undirected graph G, a matching is a set of edges such that no

two edges in the set are incident on the same vertex. In Section 26.3, we saw how

to ﬁnd a maximum matching in a bipartite graph. In this problem, we will look at

matchings in undirected graphs in general (i.e., the graphs are not required to be

bipartite).

a. A maximal matching is a matching that is not a proper subset of any other

matching. Show that a maximal matching need not be a maximum matching by

exhibiting an undirected graph G and a maximal matching M in G that is not a

maximum matching. (Hint: You can ﬁnd such a graph with only four vertices.)

b. Consider an undirected graph G D .V; E/. Give an O.E/-time greedy algo-

rithm to ﬁnd a maximal matching in G.

In this problem, we shall concentrate on a polynomial-time approximation algo-

rithm for maximum matching. Whereas the fastest known algorithm for maximum

matching takes superlinear (but polynomial) time, the approximation algorithm

here will run in linear time. You will show that the linear-time greedy algorithm

for maximal matching in part (b) is a 2-approximation algorithm for maximum

matching.

c. Show that the size of a maximum matching in G is a lower bound on the size

of any vertex cover for G.

1136 Chapter 35 Approximation Algorithms

d. Consider a maximal matching M in G D .V; E/. Let

T D f 2 V W some edge in M is incident on g :

What can you say about the subgraph of G induced by the vertices of G that

are not in T ?

e. Conclude from part (d) that 2 jM j is the size of a vertex cover for G.

f. Using parts (c) and (e), prove that the greedy algorithm in part (b) is a 2-approx-

imation algorithm for maximum matching.

35-5 Parallel machine scheduling

In the parallel-machine-scheduling problem, we are given n jobs, J 1 ; J 2 ; : : : ; J n ,

where each job J k has an associated nonnegative processing time of p k . We are

also given m identical machines, M 1 ; M 2 ; : : : ; M m . Any job can run on any ma-

chine. A schedule speciﬁes, for each job J k , the machine on which it runs and

the time period during which it runs. Each job J k must run on some machine M i

for p k consecutive time units, and during that time period no other job may run

on M i . Let C k denote the completion time of job J k , that is, the time at which

job J k completes processing. Given a schedule, we deﬁne C max D max 1j n C j to

be the makespan of the schedule. The goal is to ﬁnd a schedule whose makespan

is minimum.

For example, suppose that we have two machines M 1 and M 2 and that we have

four jobs J 1 ; J 2 ; J 3 ; J 4 , with p 1 D 2, p 2 D 12, p 3 D 4, and p 4 D 5. Then one

possible schedule runs, on machine M 1 , job J 1 followed by job J 2 , and on ma-

chine M 2 , it runs job J 4 followed by job J 3 . For this schedule, C 1 D 2, C 2 D 14,

C 3 D 9, C 4 D 5, and C max D 14. An optimal schedule runs J 2 on machine M 1 , and

it runs jobs J 1 , J 3 , and J 4 on machine M 2 . For this schedule, C 1 D 2, C 2 D 12,

C 3 D 6, C 4 D 11, and C max D 12.

Given a parallel-machine-scheduling problem, we let C

max

denote the makespan

of an optimal schedule.

a. Show that the optimal makespan is at least as large as the greatest processing

time, that is,

C

max

max

1kn

p k :

b. Show that the optimal makespan is at least as large as the average machine load,

that is,

C

max

1

m

X

1kn

p k :

Problems for Chapter 35 1137

Suppose that we use the following greedy algorithm for parallel machine schedul-

ing: whenever a machine is idle, schedule any job that has not yet been scheduled.

c. Write pseudocode to implement this greedy algorithm. What is the running

time of your algorithm?

d. For the schedule returned by the greedy algorithm, show that

C max

1

m

X

1kn

p k C max

1kn

p k :

Conclude that this algorithm is a polynomial-time 2-approximation algorithm.

35-6 Approximating a maximum spanning tree

Let G D .V; E/ be an undirected graph with distinct edge weights w.u; / on each

edge .u; / 2 E. For each vertex 2 V , let max./ D max .u;/2E fw.u; /g be

the maximum-weight edge incident on that vertex. Let S G D fmax./ W 2 V g

be the set of maximum-weight edges incident on each vertex, and let T G be the

maximum-weight spanning tree of G, that is, the spanning tree of maximum total

weight. For any subset of edges E 0 E, deﬁne w.E 0 / D

P

.u;/2E 0 w.u; /.

a. Give an example of a graph with at least 4 vertices for which S G D T G .

b. Give an example of a graph with at least 4 vertices for which S G ¤ T G .

c. Prove that S G T G for any graph G.

d. Prove that w.T G / w.S G /=2 for any graph G.

e. Give an O.V C E/-time algorithm to compute a 2-approximation to the maxi-

mum spanning tree.

35-7 An approximation algorithm for the 0-1 knapsack problem

Recall the knapsack problem from Section 16.2. There are n items, where the ith

item is worth i dollars and weighs w i pounds. We are also given a knapsack

that can hold at most W pounds. Here, we add the further assumptions that each

weight w i is at most W and that the items are indexed in monotonically decreasing

order of their values: 1 2 n .

In the 0-1 knapsack problem, we wish to ﬁnd a subset of the items whose total

weight is at most W and whose total value is maximum. The fractional knapsack

problem is like the 0-1 knapsack problem, except that we are allowed to take a

fraction of each item, rather than being restricted to taking either all or none of

1138 Chapter 35 Approximation Algorithms

each item. If we take a fraction x i of item i, where 0 x i 1, we contribute

x i w i to the weight of the knapsack and receive value x i i . Our goal is to develop

a polynomial-time 2-approximation algorithm for the 0-1 knapsack problem.

In order to design a polynomial-time algorithm, we consider restricted instances

of the 0-1 knapsack problem. Given an instance I of the knapsack problem, we

form restricted instances I j , for j D 1; 2; : : : ; n, by removing items 1; 2; : : : ; j 1

and requiring the solution to include item j (all of item j in both the fractional

and 0-1 knapsack problems). No items are removed in instance I 1 . For instance I j ,

let P j denote an optimal solution to the 0-1 problem and Q j denote an optimal

solution to the fractional problem.

a. Argue that an optimal solution to instance I of the 0-1 knapsack problem is one

of fP 1 ; P 2 ; : : : ; P n g.

b. Prove that we can ﬁnd an optimal solution Q j to the fractional problem for in-

stance I j by including item j and then using the greedy algorithm in which

at each step, we take as much as possible of the unchosen item in the set

fj C 1; j C 2; : : : ; ng with maximum value per pound i =w i .

c. Prove that we can always construct an optimal solution Q j to the fractional

problem for instance I j that includes at most one item fractionally. That is, for

all items except possibly one, we either include all of the item or none of the

item in the knapsack.

d. Given an optimal solution Q j to the fractional problem for instance I j , form

solution R j from Q j by deleting any fractional items from Q j . Let .S/ denote

the total value of items taken in a solution S. Prove that .R j / .Q j /=2

.P j /=2.

e. Give a polynomial-time algorithm that returns a maximum-value solution from

the set fR 1 ; R 2 ; : : : ; R n g, and prove that your algorithm is a polynomial-time

2-approximation algorithm for the 0-1 knapsack problem.

Chapter notes

Although methods that do not necessarily compute exact solutions have been

known for thousands of years (for example, methods to approximate the value

of ), the notion of an approximation algorithm is much more recent. Hochbaum

[172] credits Garey, Graham, and Ullman [128] and Johnson [190] with formal-

izing the concept of a polynomial-time approximation algorithm. The ﬁrst such

algorithm is often credited to Graham [149].

Notes for Chapter 35 1139

Since this early work, thousands of approximation algorithms have been de-

signed for a wide range of problems, and there is a wealth of literature on this

ﬁeld. Recent texts by Ausiello et al. [26], Hochbaum [172], and Vazirani [345]

deal exclusively with approximation algorithms, as do surveys by Shmoys [315]

and Klein and Young [207]. Several other texts, such as Garey and Johnson [129]

and Papadimitriou and Steiglitz [271], have signiﬁcant coverage of approximation

algorithms as well. Lawler, Lenstra, Rinnooy Kan, and Shmoys [225] provide an

extensive treatment of approximation algorithms for the traveling-salesman prob-

lem.

Papadimitriou and Steiglitz attribute the algorithm APPROX-VERTEX-COVER

to F. Gavril and M. Yannakakis. The vertex-cover problem has been studied exten-

sively (Hochbaum [172] lists 16 different approximation algorithms for this prob-

lem), but all the approximation ratios are at least 2 o.1/.

The algorithm APPROX-TSP-TOUR appears in a paper by Rosenkrantz, Stearns,

and Lewis [298]. Christoﬁdes improved on this algorithm and gave a 3=2-approx-

imation algorithm for the traveling-salesman problem with the triangle inequality.

Arora [22] and Mitchell [257] have shown that if the points are in the euclidean

plane, there is a polynomial-time approximation scheme. Theorem 35.3 is due to

Sahni and Gonzalez [301].

The analysis of the greedy heuristic for the set-covering problem is modeled

after the proof published by Chv´atal [68] of a more general result; the basic result

as presented here is due to Johnson [190] and Lov´asz [238].

The algorithm APPROX-SUBSET-SUM and its analysis are loosely modeled after

related approximation algorithms for the knapsack and subset-sum problems by

Ibarra and Kim [187].

Problem 35-7 is a combinatorial version of a more general result on approximat-

ing knapsack-type integer programs by Bienstock and McClosky [45].

The randomized algorithm for MAX-3-CNF satisﬁability is implicit in the work

of Johnson [190]. The weighted vertex-cover algorithm is by Hochbaum [171].

Section 35.4 only touches on the power of randomization and linear program-

ming in the design of approximation algorithms. A combination of these two ideas

yields a technique called “randomized rounding,” which formulates a problem as

an integer linear program, solves the linear-programming relaxation, and interprets

the variables in the solution as probabilities. These probabilities then help guide

the solution of the original problem. This technique was ﬁrst used by Raghavan

and Thompson [290], and it has had many subsequent uses. (See Motwani, Naor,

and Raghavan [261] for a survey.) Several other notable recent ideas in the ﬁeld

of approximation algorithms include the primal-dual method (see Goemans and

Williamson [135] for a survey), ﬁnding sparse cuts for use in divide-and-conquer

algorithms [229], and the use of semideﬁnite programming [134].

1140 Chapter 35 Approximation Algorithms

As mentioned in the chapter notes for Chapter 34, recent results in probabilisti-

cally checkable proofs have led to lower bounds on the approximability of many

problems, including several in this chapter. In addition to the references there,

the chapter by Arora and Lund [23] contains a good description of the relation-

ship between probabilistically checkable proofs and the hardness of approximating

various problems.

VIII Appendix: Mathematical Background

Introduction

When we analyze algorithms, we often need to draw upon a body of mathematical

tools. Some of these tools are as simple as high-school algebra, but others may be

new to you. In Part I, we saw how to manipulate asymptotic notations and solve

recurrences. This appendix comprises a compendium of several other concepts and

methods we use to analyze algorithms. As noted in the introduction to Part I, you

may have seen much of the material in this appendix before having read this book

(although the speciﬁc notational conventions we use might occasionally differ from

those you have seen elsewhere). Hence, you should treat this appendix as reference

material. As in the rest of this book, however, we have included exercises and

problems, in order for you to improve your skills in these areas.

Appendix A offers methods for evaluating and bounding summations, which

occur frequently in the analysis of algorithms. Many of the formulas here appear

in any calculus text, but you will ﬁnd it convenient to have these methods compiled

in one place.

Appendix B contains basic deﬁnitions and notations for sets, relations, functions,

graphs, and trees. It also gives some basic properties of these mathematical objects.

Appendix C begins with elementary principles of counting: permutations, com-

binations, and the like. The remainder contains deﬁnitions and properties of basic

probability. Most of the algorithms in this book require no probability for their

analysis, and thus you can easily omit the latter sections of the chapter on a ﬁrst

reading, even without skimming them. Later, when you encounter a probabilistic

analysis that you want to understand better, you will ﬁnd Appendix C well orga-

nized for reference purposes.

1144 Part VIII Appendix: Mathematical Background

Appendix D deﬁnes matrices, their operations, and some of their basic prop-

erties. You have probably seen most of this material already if you have taken a

course in linear algebra, but you might ﬁnd it helpful to have one place to look for

our notation and deﬁnitions.

A Summations

When an algorithm contains an iterative control construct such as a while or for

loop, we can express its running time as the sum of the times spent on each exe-

cution of the body of the loop. For example, we found in Section 2.2 that the j th

iteration of insertion sort took time proportional to j in the worst case. By adding

up the time spent on each iteration, we obtained the summation (or series)

n X

j D2

j :

When we evaluated this summation, we attained a bound of ‚.n 2 / on the worst-

case running time of the algorithm. This example illustrates why you should know

how to manipulate and bound summations.

Section A.1 lists several basic formulas involving summations. Section A.2 of-

fers useful techniques for bounding summations. We present the formulas in Sec-

tion A.1 without proof, though proofs for some of them appear in Section A.2 to

illustrate the methods of that section. You can ﬁnd most of the other proofs in any

calculus text.

A.1 Summation formulas and properties

Given a sequence a 1 ; a 2 ; : : : ; a n of numbers, where n is a nonnegative integer, we

can write the ﬁnite sum a 1 C a 2 C C a n as

n X

kD1

a k :

If n D 0, the value of the summation is deﬁned to be 0. The value of a ﬁnite series

is always well deﬁned, and we can add its terms in any order.

Given an inﬁnite sequence a 1 ; a 2 ; : : : of numbers, we can write the inﬁnite sum

a 1 C a 2 C as

1146 Appendix A Summations

1 X

kD1

a k ;

which we interpret to mean

lim

n!1

n X

kD1

a k :

If the limit does not exist, the series diverges; otherwise, it converges. The terms

of a convergent series cannot always be added in any order. We can, however,

rearrange the terms of an absolutely convergent series, that is, a series

P 1

kD1

a k

for which the series

P 1

kD1

ja k j also converges.

Linearity

For any real number c and any ﬁnite sequences a 1 ; a 2 ; : : : ; a n and b 1 ; b 2 ; : : : ; b n ,

n X

kD1

.ca k C b k / D c

n X

kD1

a k C

n X

kD1

b k :

The linearity property also applies to inﬁnite convergent series.

We can exploit the linearity property to manipulate summations incorporating

asymptotic notation. For example,

n X

kD1

‚.f .k// D ‚

n X

kD1

f .k/

!

:

In this equation, the ‚-notation on the left-hand side applies to the variable k, but

on the right-hand side, it applies to n. We can also apply such manipulations to

inﬁnite convergent series.

Arithmetic series

The summation

n X

kD1

k D 1 C 2 C C n ;

is an arithmetic series and has the value

n X

kD1

k D

1

2

n.n C 1/ (A.1)

D ‚.n

2

/ : (A.2)

A.1 Summation formulas and properties 1147

Sums of squares and cubes

We have the following summations of squares and cubes:

n X

kD0

k

2

D

n.n C 1/.2n C 1/

6

; (A.3)

n X

kD0

k

3

D

n 2 .n C 1/ 2

4

: (A.4)

Geometric series

For real x ¤ 1, the summation

n X

kD0

x

k

D 1 C x C x

2

C C x

n

is a geometric or exponential series and has the value

n X

kD0

x

k

D

x nC1 1

x 1

: (A.5)

When the summation is inﬁnite and jxj < 1, we have the inﬁnite decreasing geo-

metric series

1 X

kD0

x

k

D

1

1 x

: (A.6)

Harmonic series

For positive integers n, the nth harmonic number is

H n D 1 C

1

2

C

1

3

C

1

4

C C

1

n

D

n X

kD1

1

k

D ln n C O.1/ : (A.7)

(We shall prove a related bound in Section A.2.)

Integrating and differentiating series

By integrating or differentiating the formulas above, additional formulas arise. For

example, by differentiating both sides of the inﬁnite geometric series (A.6) and

multiplying by x, we get

1148 Appendix A Summations

1 X

kD0

kx

k

D

x

.1 x/ 2

(A.8)

for jxj < 1.

Telescoping series

For any sequence a 0 ; a 1 ; : : : ; a n ,

n X

kD1

.a k a k1 / D a n a 0 ; (A.9)

since each of the terms a 1 ; a 2 ; : : : ; a n1 is added in exactly once and subtracted out

exactly once. We say that the sum telescopes. Similarly,

n1 X

kD0

.a k a kC1 / D a 0 a n :

As an example of a telescoping sum, consider the series

n1 X

kD1

1

k.k C 1/

:

Since we can rewrite each term as

1

k.k C 1/

D

1

k

1

k C 1

;

we get

n1 X

kD1

1

k.k C 1/

D

n1 X

kD1

1

k

1

k C 1

D 1

1

n

:

Products

We can write the ﬁnite product a 1 a 2 a n as

n Y

kD1

a k :

If n D 0, the value of the product is deﬁned to be 1. We can convert a formula with

a product to a formula with a summation by using the identity

lg

n Y

kD1

a k

!

D

n X

kD1

lg a k :

A.2 Bounding summations 1149

Exercises

A.1-1

Find a simple formula for

P n

kD1

.2k 1/.

A.1-2 ?

Show that

P n

kD1

1=.2k 1/ D ln.

p

n/ C O.1/ by manipulating the harmonic

series.

A.1-3

Show that

P 1

kD0

k 2 x k D x.1 C x/=.1 x/ 3

for 0 < jxj < 1.

A.1-4 ?

Show that

P 1

kD0

.k 1/=2 k D 0.

A.1-5 ?

Evaluate the sum

P 1

kD1

.2k C 1/x 2k

.

A.1-6

Prove that

P n

kD1

O.f k .i// D O

P n

kD1

f k .i/

by using the linearity property of

summations.

A.1-7

Evaluate the product

Q n

kD1

2 4 k

.

A.1-8 ?

Evaluate the product

Q n

kD2

.1 1=k 2 /.

A.2 Bounding summations

We have many techniques at our disposal for bounding the summations that de-

scribe the running times of algorithms. Here are some of the most frequently used

methods.

Mathematical induction

The most basic way to evaluate a series is to use mathematical induction. As an

example, let us prove that the arithmetic series

P n

kD1

k evaluates to

1

2

n.nC1/. We

can easily verify this assertion for n D 1. We make the inductive assumption that

1150 Appendix A Summations

it holds for n, and we prove that it holds for n C 1. We have

nC1 X

kD1

k D

n X

kD1

k C .n C 1/

D

1

2

n.n C 1/ C .n C 1/

D

1

2

.n C 1/.n C 2/ :

You don’t always need to guess the exact value of a summation in order to use

mathematical induction. Instead, you can use induction to prove a bound on a sum-

mation. As an example, let us prove that the geometric series

P n

kD0

3 k

is O.3 n /.

More speciﬁcally, let us prove that

P n

kD0

3 k c3 n

for some constant c. For the

initial condition n D 0, we have

P 0

kD0

3 k D 1 c 1 as long as c 1. Assuming

that the bound holds for n, let us prove that it holds for n C 1. We have

nC1 X

kD0

3

k

D

n X

kD0

3

k

C 3

nC1

c3

n

C 3

nC1

(by the inductive hypothesis)

D

1

3

C

1

c

c3

nC1

c3

nC1

as long as .1=3 C 1=c/ 1 or, equivalently, c 3=2. Thus,

P n

kD0

3 k D O.3 n /,

as we wished to show.

We have to be careful when we use asymptotic notation to prove bounds by in-

duction. Consider the following fallacious proof that

P n

kD1

k D O.n/. Certainly,

P 1

kD1

k D O.1/. Assuming that the bound holds for n, we now prove it for n C 1:

nC1 X

kD1

k D

n X

kD1

k C .n C 1/

D O.n/ C .n C 1/

wrong!!

D O.n C 1/ :

The bug in the argument is that the “constant” hidden by the “big-oh” grows with n

and thus is not constant. We have not shown that the same constant works for all n.

Bounding the terms

We can sometimes obtain a good upper bound on a series by bounding each term

of the series, and it often sufﬁces to use the largest term to bound the others. For

A.2 Bounding summations 1151

example, a quick upper bound on the arithmetic series (A.1) is

n X

kD1

k

n X

kD1

n

D n

2

:

In general, for a series

P n

kD1

a k , if we let a max D max 1kn a k , then

n X

kD1

a k n a max :

The technique of bounding each term in a series by the largest term is a weak

method when the series can in fact be bounded by a geometric series. Given the

series

P n

kD0

a k , suppose that a kC1 =a k r for all k 0, where 0 < r < 1 is a

constant. We can bound the sum by an inﬁnite decreasing geometric series, since

a k a 0 r k

, and thus

n X

kD0

a k

1 X

kD0

a 0 r

k

D a 0

1 X

kD0

r

k

D a 0

1

1 r

:

We can apply this method to bound the summation

P 1

kD1

.k=3 k /. In order to

start the summation at k D 0, we rewrite it as

P 1

kD0

..k C 1/=3 kC1 /. The ﬁrst

term (a 0 ) is 1=3, and the ratio (r) of consecutive terms is

.k C 2/=3 kC2

.k C 1/=3 kC1

D

1

3

k C 2

k C 1

2

3

for all k 0. Thus, we have

1 X

kD1

k

3 k

D

1 X

kD0

k C 1

3 kC1

1

3

1

1 2=3

D 1 :

1152 Appendix A Summations

A common bug in applying this method is to show that the ratio of consecu-

tive terms is less than 1 and then to assume that the summation is bounded by a

geometric series. An example is the inﬁnite harmonic series, which diverges since

1 X

kD1

1

k

D lim

n!1

n X

kD1

1

k

D lim

n!1

‚.lg n/

D 1 :

The ratio of the .kC1/st and kth terms in this series is k=.kC1/ < 1, but the series

is not bounded by a decreasing geometric series. To bound a series by a geometric

series, we must show that there is an r < 1, which is a constant, such that the ratio

of all pairs of consecutive terms never exceeds r. In the harmonic series, no such r

exists because the ratio becomes arbitrarily close to 1.

Splitting summations

One way to obtain bounds on a difﬁcult summation is to express the series as the

sum of two or more series by partitioning the range of the index and then to bound

each of the resulting series. For example, suppose we try to ﬁnd a lower bound

on the arithmetic series

P n

kD1

k, which we have already seen has an upper bound

of n 2

. We might attempt to bound each term in the summation by the smallest term,

but since that term is 1, we get a lower bound of n for the summation—far off from

our upper bound of n 2

.

We can obtain a better lower bound by ﬁrst splitting the summation. Assume for

convenience that n is even. We have

n X

kD1

k D

n=2 X

kD1

k C

n X

kDn=2C1

k

n=2 X

kD1

0 C

n X

kDn=2C1

.n=2/

D .n=2/

2

D .n

2

/ ;

which is an asymptotically tight bound, since

P n

kD1

k D O.n 2 /.

For a summation arising from the analysis of an algorithm, we can often split

the summation and ignore a constant number of the initial terms. Generally, this

technique applies when each term a k in a summation

P n

kD0

a k is independent of n.

A.2 Bounding summations 1153

Then for any constant k 0 > 0, we can write

n X

kD0

a k D

k 0 1 X

kD0

a k C

n X

kDk 0

a k

D ‚.1/ C

n X

kDk 0

a k ;

since the initial terms of the summation are all constant and there are a constant

number of them. We can then use other methods to bound

P n

kDk 0

a k . This tech-

nique applies to inﬁnite summations as well. For example, to ﬁnd an asymptotic

upper bound on

1 X

kD0

k 2

2 k

;

we observe that the ratio of consecutive terms is

.k C 1/ 2 =2 kC1

k 2 =2 k

D

.k C 1/ 2

2k 2

8

9

if k 3. Thus, the summation can be split into

1 X

kD0

k 2

2 k

D

2 X

kD0

k 2

2 k

C

1 X

kD3

k 2

2 k

2 X

kD0

k 2

2 k

C

9

8

1 X

kD0

8

9

k

D O.1/ ;

since the ﬁrst summation has a constant number of terms and the second summation

is a decreasing geometric series.

The technique of splitting summations can help us determine asymptotic bounds

in much more difﬁcult situations. For example, we can obtain a bound of O.lg n/

on the harmonic series (A.7):

H n D

n X

kD1

1

k

:

We do so by splitting the range 1 to n into blg nc C 1 pieces and upper-bounding

the contribution of each piece by 1. For i D 0; 1; : : : ; blg nc, the ith piece consists

1154 Appendix A Summations

of the terms starting at 1=2 i

and going up to but not including 1=2 iC1

. The last

piece might contain terms not in the original harmonic series, and thus we have

n X

kD1

1

k

blg nc X

iD0

2

i

1 X

j D0

1

2 i C j

blg nc X

iD0

2

i

1 X

j D0

1

2 i

D

blg nc X

iD0

1

lg n C 1 : (A.10)

Approximation by integrals

When a summation has the form

P n

kDm

f .k/, where f .k/ is a monotonically in-

creasing function, we can approximate it by integrals:

Z n

m1

f .x/ dx

n X

kDm

f .k/

Z nC1

m

f .x/ dx : (A.11)

Figure A.1 justiﬁes this approximation. The summation is represented as the area

of the rectangles in the ﬁgure, and the integral is the shaded region under the curve.

When f .k/ is a monotonically decreasing function, we can use a similar method

to provide the bounds

Z nC1

m

f .x/ dx

n X

kDm

f .k/

Z n

m1

f .x/ dx : (A.12)

The integral approximation (A.12) gives a tight estimate for the nth harmonic

number. For a lower bound, we obtain

n X

kD1

1

k

Z nC1

1

dx

x

D ln.n C 1/ : (A.13)

For the upper bound, we derive the inequality

n X

kD2

1

k

Z n

1

dx

x

D ln n ;

A.2 Bounding summations 1155

n+1 n–1 n–2 m+2 m m –1

f (m)

f (m+1)

f (m+2)

f (n–2)

f (n–1)

f (n)

f (x)

x

… …

n

… …

(a)

m+1

n+1 n–1 n–2 m+2 m m –1

f (m)

f (m+1)

f (m+2)

f (n–2)

f (n–1)

f (n)

f (x)

x

… …

n

… …

(b)

m+1

Figure A.1 Approximation of

P n

kDm

f .k/ by integrals. The area of each rectangle is shown

within the rectangle, and the total rectangle area represents the value of the summation. The in-

tegral is represented by the shaded area under the curve. By comparing areas in (a), we get R n

m1

f .x/ dx

P n

kDm

f .k/, and then by shifting the rectangles one unit to the right, we get

P n

kDm

f .k/

R nC1

m

f .x/ dx in (b).

1156 Appendix A Summations

which yields the bound

n X

kD1

1

k

ln n C 1 : (A.14)

Exercises

A.2-1

Show that

P n

kD1

1=k 2

is bounded above by a constant.

A.2-2

Find an asymptotic upper bound on the summation

blg nc X

kD0

˙

n=2

k

:

A.2-3

Show that the nth harmonic number is .lg n/ by splitting the summation.

A.2-4

Approximate

P n

kD1

k 3

with an integral.

A.2-5

Why didn’t we use the integral approximation (A.12) directly on

P n

kD1

1=k to

obtain an upper bound on the nth harmonic number?

Problems

A-1 Bounding summations

Give asymptotically tight bounds on the following summations. Assume that r 0

and s 0 are constants.

a.

n X

kD1

k

r

.

b.

n X

kD1

lg

s

k.

Notes for Appendix A 1157

c.

n X

kD1

k

r

lg

s

k.

Appendix notes

Knuth [209] provides an excellent reference for the material presented here. You

can ﬁnd basic properties of series in any good calculus book, such as Apostol [18]

or Thomas et al. [334].

B Sets, Etc.

Many chapters of this book touch on the elements of discrete mathematics. This

appendix reviews more completely the notations, deﬁnitions, and elementary prop-

erties of sets, relations, functions, graphs, and trees. If you are already well versed

in this material, you can probably just skim this chapter.

B.1 Sets

A set is a collection of distinguishable objects, called its members or elements. If

an object x is a member of a set S, we write x 2 S (read “x is a member of S”

or, more brieﬂy, “x is in S”). If x is not a member of S, we write x 62 S. We

can describe a set by explicitly listing its members as a list inside braces. For

example, we can deﬁne a set S to contain precisely the numbers 1, 2, and 3 by

writing S D f1; 2; 3g. Since 2 is a member of the set S, we can write 2 2 S, and

since 4 is not a member, we have 4 … S. A set cannot contain the same object more

than once, 1 and its elements are not ordered. Two sets A and B are equal, written

A D B, if they contain the same elements. For example, f1; 2; 3; 1g D f1; 2; 3g D

f3; 2; 1g.

We adopt special notations for frequently encountered sets:

; denotes the empty set, that is, the set containing no members.

Z denotes the set of integers, that is, the set f: : : ; 2; 1; 0; 1; 2; : : :g.

R denotes the set of real numbers.

N denotes the set of natural numbers, that is, the set f0; 1; 2; : : :g. 2

1

A variation of a set, which can contain the same object more than once, is called a multiset.

2

Some authors start the natural numbers with 1 instead of 0. The modern trend seems to be to start

with 0.

B.1 Sets 1159

If all the elements of a set A are contained in a set B, that is, if x 2 A implies

x 2 B, then we write A B and say that A is a subset of B. A set A is a

proper subset of B, written A B, if A B but A ¤ B. (Some authors use the

symbol “” to denote the ordinary subset relation, rather than the proper-subset

relation.) For any set A, we have A A. For two sets A and B, we have A D B

if and only if A B and B A. For any three sets A, B, and C , if A B

and B C , then A C . For any set A, we have ; A.

We sometimes deﬁne sets in terms of other sets. Given a set A, we can deﬁne a

set B A by stating a property that distinguishes the elements of B. For example,

we can deﬁne the set of even integers by fx W x 2 Z and x=2 is an integerg. The

colon in this notation is read “such that.” (Some authors use a vertical bar in place

of the colon.)

Given two sets A and B, we can also deﬁne new sets by applying set operations:

The intersection of sets A and B is the set

A \ B D fx W x 2 A and x 2 Bg :

The union of sets A and B is the set

A [ B D fx W x 2 A or x 2 Bg :

The difference between two sets A and B is the set

A B D fx W x 2 A and x … Bg :

Set operations obey the following laws:

Empty set laws:

A \ ; D ; ;

A [ ; D A :

Idempotency laws:

A \ A D A ;

A [ A D A :

Commutative laws:

A \ B D B \ A ;

A [ B D B [ A :

1160 Appendix B Sets, Etc.

A A A A A

A

B B B B B

.B \ C / [

[

D D

D D

A .B \ C / .A B/ .A C /

C C C C C

Figure B.1 A Venn diagram illustrating the ﬁrst of DeMorgan’s laws (B.2). Each of the sets A, B,

and C is represented as a circle.

Associative laws:

A \ .B \ C / D .A \ B/ \ C ;

A [ .B [ C / D .A [ B/ [ C :

Distributive laws:

A \ .B [ C / D .A \ B/ [ .A \ C / ;

A [ .B \ C / D .A [ B/ \ .A [ C / :

(B.1)

Absorption laws:

A \ .A [ B/ D A ;

A [ .A \ B/ D A :

DeMorgan’s laws:

A .B \ C / D .A B/ [ .A C / ;

A .B [ C / D .A B/ \ .A C / :

(B.2)

Figure B.1 illustrates the ﬁrst of DeMorgan’s laws, using a Venn diagram: a graph-

ical picture in which sets are represented as regions of the plane.

Often, all the sets under consideration are subsets of some larger set U called the

universe. For example, if we are considering various sets made up only of integers,

the set Z of integers is an appropriate universe. Given a universe U , we deﬁne the

complement of a set A as A D U A D fx W x 2 U and x 62 Ag. For any set

A U , we have the following laws:

A D A ;

A \ A D ; ;

A [ A D U :

B.1 Sets 1161

We can rewrite DeMorgan’s laws (B.2) with set complements. For any two sets

B; C U , we have

B \ C D B [ C ;

B [ C D B \ C :

Two sets A and B are disjoint if they have no elements in common, that is, if

A \ B D ;. A collection S D fS i g of nonempty sets forms a partition of a set S if

the sets are pairwise disjoint, that is, S i ; S j 2 S and i ¤ j imply S i \ S j D ;,

and

their union is S, that is,

S D

[

S i 2S

S i :

In other words, S forms a partition of S if each element of S appears in exactly

one S i 2 S .

The number of elements in a set is the cardinality (or size) of the set, denoted jSj.

Two sets have the same cardinality if their elements can be put into a one-to-one

correspondence. The cardinality of the empty set is j;j D 0. If the cardinality of a

set is a natural number, we say the set is ﬁnite; otherwise, it is inﬁnite. An inﬁnite

set that can be put into a one-to-one correspondence with the natural numbers N is

countably inﬁnite; otherwise, it is uncountable. For example, the integers Z are

countable, but the reals R are uncountable.

For any two ﬁnite sets A and B, we have the identity

jA [ Bj D jAj C jBj jA \ Bj ; (B.3)

from which we can conclude that

jA [ Bj jAj C jBj :

If A and B are disjoint, then jA \ Bj D 0 and thus jA [ Bj D jAj C jBj. If

A B, then jAj jBj.

A ﬁnite set of n elements is sometimes called an n-set. A 1-set is called a

singleton. A subset of k elements of a set is sometimes called a k-subset.

We denote the set of all subsets of a set S, including the empty set and S itself,

by 2 S

; we call 2 S

the power set of S. For example, 2 fa;bg D f;; fag ; fbg ; fa; bgg.

The power set of a ﬁnite set S has cardinality 2 jSj

(see Exercise B.1-5).

We sometimes care about setlike structures in which the elements are ordered.

An ordered pair of two elements a and b is denoted .a; b/ and is deﬁned formally

as the set .a; b/ D fa; fa; bgg. Thus, the ordered pair .a; b/ is not the same as the

ordered pair .b; a/.

1162 Appendix B Sets, Etc.

The Cartesian product of two sets A and B, denoted A B, is the set of all

ordered pairs such that the ﬁrst element of the pair is an element of A and the

second is an element of B. More formally,

A B D f.a; b/ W a 2 A and b 2 Bg :

For example, fa; bg fa; b; cg D f.a; a/; .a; b/; .a; c/; .b; a/; .b; b/; .b; c/g. When

A and B are ﬁnite sets, the cardinality of their Cartesian product is

jA Bj D jAj jBj : (B.4)

The Cartesian product of n sets A 1 ; A 2 ; : : : ; A n is the set of n-tuples

A 1 A 2 A n D f.a 1 ; a 2 ; : : : ; a n / W a i 2 A i for i D 1; 2; : : : ; ng ;

whose cardinality is

jA 1 A 2 A n j D jA 1 j jA 2 j jA n j

if all sets are ﬁnite. We denote an n-fold Cartesian product over a single set A by

the set

A

n

D A A A ;

whose cardinality is jA n j D jAj

n

if A is ﬁnite. We can also view an n-tuple as a

ﬁnite sequence of length n (see page 1166).

Exercises

B.1-1

Draw Venn diagrams that illustrate the ﬁrst of the distributive laws (B.1).

B.1-2

Prove the generalization of DeMorgan’s laws to any ﬁnite collection of sets:

A 1 \ A 2 \ \ A n D A 1 [ A 2 [ [ A n ;

A 1 [ A 2 [ [ A n D A 1 \ A 2 \ \ A n :

B.2 Relations 1163

B.1-3 ?

Prove the generalization of equation (B.3), which is called the principle of inclu-

sion and exclusion:

jA 1 [ A 2 [ [ A n j D

jA 1 j C jA 2 j C C jA n j

jA 1 \ A 2 j jA 1 \ A 3 j (all pairs)

C jA 1 \ A 2 \ A 3 j C (all triples)

:

:

:

C .1/

n1

jA 1 \ A 2 \ \ A n j :

B.1-4

Show that the set of odd natural numbers is countable.

B.1-5

Show that for any ﬁnite set S, the power set 2 S

has 2 jSj

elements (that is, there

are 2 jSj

distinct subsets of S).

B.1-6

Give an inductive deﬁnition for an n-tuple by extending the set-theoretic deﬁnition

for an ordered pair.

B.2 Relations

A binary relation R on two sets A and B is a subset of the Cartesian product A B.

If .a; b/ 2 R, we sometimes write a R b. When we say that R is a binary relation

on a set A, we mean that R is a subset of A A. For example, the “less than”

relation on the natural numbers is the set f.a; b/ W a; b 2 N and a < bg. An n-ary

relation on sets A 1 ; A 2 ; : : : ; A n is a subset of A 1 A 2 A n .

A binary relation R A A is reﬂexive if

a R a

for all a 2 A. For example, “D” and “” are reﬂexive relations on N , but “<” is

not. The relation R is symmetric if

a R b implies b R a

for all a; b 2 A. For example, “D” is symmetric, but “<” and “” are not. The

relation R is transitive if

a R b and b R c imply a R c

1164 Appendix B Sets, Etc.

for all a; b; c 2 A. For example, the relations “<,” “,” and “D” are transitive, but

the relation R D f.a; b/ W a; b 2 N and a D b 1g is not, since 3 R 4 and 4 R 5

do not imply 3 R 5.

A relation that is reﬂexive, symmetric, and transitive is an equivalence relation.

For example, “D” is an equivalence relation on the natural numbers, but “<” is not.

If R is an equivalence relation on a set A, then for a 2 A, the equivalence class

of a is the set Œa D fb 2 A W a R bg, that is, the set of all elements equivalent to a.

For example, if we deﬁne R D f.a; b/ W a; b 2 N and a C b is an even numberg,

then R is an equivalence relation, since a C a is even (reﬂexive), a C b is even

implies b C a is even (symmetric), and a C b is even and b C c is even imply

a C c is even (transitive). The equivalence class of 4 is Œ4 D f0; 2; 4; 6; : : :g, and

the equivalence class of 3 is Œ3 D f1; 3; 5; 7; : : :g. A basic theorem of equivalence

classes is the following.

Theorem B.1 (An equivalence relation is the same as a partition)

The equivalence classes of any equivalence relation R on a set A form a partition

of A, and any partition of A determines an equivalence relation on A for which the

sets in the partition are the equivalence classes.

Proof For the ﬁrst part of the proof, we must show that the equivalence classes

of R are nonempty, pairwise-disjoint sets whose union is A. Because R is reﬂex-

ive, a 2 Œa, and so the equivalence classes are nonempty; moreover, since every

element a 2 A belongs to the equivalence class Œa, the union of the equivalence

classes is A. It remains to show that the equivalence classes are pairwise disjoint,

that is, if two equivalence classes Œa and Œb have an element c in common, then

they are in fact the same set. Suppose that a R c and b R c. By symmetry, c R b,

and by transitivity, a R b. Thus, for any arbitrary element x 2 Œa, we have x R a

and, by transitivity, x R b, and thus Œa Œb. Similarly, Œb Œa, and thus

Œa D Œb.

For the second part of the proof, let A D fA i g be a partition of A, and deﬁne

R D f.a; b/ W there exists i such that a 2 A i and b 2 A i g. We claim that R is an

equivalence relation on A. Reﬂexivity holds, since a 2 A i implies a R a. Symme-

try holds, because if a R b, then a and b are in the same set A i , and hence b R a.

If a R b and b R c, then all three elements are in the same set A i , and thus a R c

and transitivity holds. To see that the sets in the partition are the equivalence

classes of R, observe that if a 2 A i , then x 2 Œa implies x 2 A i , and x 2 A i

implies x 2 Œa.

A binary relation R on a set A is antisymmetric if

a R b and b R a imply a D b :

B.2 Relations 1165

For example, the “” relation on the natural numbers is antisymmetric, since a b

and b a imply a D b. A relation that is reﬂexive, antisymmetric, and transitive

is a partial order, and we call a set on which a partial order is deﬁned a partially

ordered set. For example, the relation “is a descendant of” is a partial order on the

set of all people (if we view individuals as being their own descendants).

In a partially ordered set A, there may be no single “maximum” element a such

that b R a for all b 2 A. Instead, the set may contain several maximal elements a

such that for no b 2 A, where b ¤ a, is it the case that a R b. For example, a

collection of different-sized boxes may contain several maximal boxes that don’t

ﬁt inside any other box, yet it has no single “maximum” box into which any other

box will ﬁt. 3

A relation R on a set A is a total relation if for all a; b 2 A, we have a R b

or b R a (or both), that is, if every pairing of elements of A is related by R. A

partial order that is also a total relation is a total order or linear order. For example,

the relation “” is a total order on the natural numbers, but the “is a descendant

of” relation is not a total order on the set of all people, since there are individuals

neither of whom is descended from the other. A total relation that is transitive, but

not necessarily reﬂexive and antisymmetric, is a total preorder.

Exercises

B.2-1

Prove that the subset relation “” on all subsets of Z is a partial order but not a

total order.

B.2-2

Show that for any positive integer n, the relation “equivalent modulo n” is an equiv-

alence relation on the integers. (We say that a b .mod n/ if there exists an

integer q such that a b D qn.) Into what equivalence classes does this relation

partition the integers?

B.2-3

Give examples of relations that are

a. reﬂexive and symmetric but not transitive,

b. reﬂexive and transitive but not symmetric,

c. symmetric and transitive but not reﬂexive.

3

To be precise, in order for the “ﬁt inside” relation to be a partial order, we need to view a box as

ﬁtting inside itself.

1166 Appendix B Sets, Etc.

B.2-4

Let S be a ﬁnite set, and let R be an equivalence relation on S S. Show that if

in addition R is antisymmetric, then the equivalence classes of S with respect to R

are singletons.

B.2-5

Professor Narcissus claims that if a relation R is symmetric and transitive, then it is

also reﬂexive. He offers the following proof. By symmetry, a R b implies b R a.

Transitivity, therefore, implies a R a. Is the professor correct?

B.3 Functions

Given two sets A and B, a function f is a binary relation on A and B such that

for all a 2 A, there exists precisely one b 2 B such that .a; b/ 2 f . The set A is

called the domain of f , and the set B is called the codomain of f . We sometimes

write f W A ! B; and if .a; b/ 2 f , we write b D f .a/, since b is uniquely

determined by the choice of a.

Intuitively, the function f assigns an element of B to each element of A. No

element of A is assigned two different elements of B, but the same element of B

can be assigned to two different elements of A. For example, the binary relation

f D f.a; b/ W a; b 2 N and b D a mod 2g

is a function f W N ! f0; 1g, since for each natural number a, there is exactly one

value b in f0; 1g such that b D a mod 2. For this example, 0 D f .0/, 1 D f .1/,

0 D f .2/, etc. In contrast, the binary relation

g D f.a; b/ W a; b 2 N and a C b is eveng

is not a function, since .1; 3/ and .1; 5/ are both in g, and thus for the choice a D 1,

there is not precisely one b such that .a; b/ 2 g.

Given a function f W A ! B, if b D f .a/, we say that a is the argument of f

and that b is the value of f at a. We can deﬁne a function by stating its value for

every element of its domain. For example, we might deﬁne f .n/ D 2n for n 2 N ,

which means f D f.n; 2n/ W n 2 N g. Two functions f and g are equal if they

have the same domain and codomain and if, for all a in the domain, f .a/ D g.a/.

A ﬁnite sequence of length n is a function f whose domain is the set of n

integers f0; 1; : : : ; n 1g. We often denote a ﬁnite sequence by listing its values:

hf .0/; f .1/; : : : ; f .n 1/i. An inﬁnite sequence is a function whose domain is

the set N of natural numbers. For example, the Fibonacci sequence, deﬁned by

recurrence (3.22), is the inﬁnite sequence h0; 1; 1; 2; 3; 5; 8; 13; 21; : : :i.

B.3 Functions 1167

When the domain of a function f is a Cartesian product, we often omit the extra

parentheses surrounding the argument of f . For example, if we had a function

f W A 1 A 2 A n ! B, we would write b D f .a 1 ; a 2 ; : : : ; a n / instead

of b D f ..a 1 ; a 2 ; : : : ; a n //. We also call each a i an argument to the function f ,

though technically the (single) argument to f is the n-tuple .a 1 ; a 2 ; : : : ; a n /.

If f W A ! B is a function and b D f .a/, then we sometimes say that b is the

image of a under f . The image of a set A 0 A under f is deﬁned by

f .A

0

/ D fb 2 B W b D f .a/ for some a 2 A

0

g :

The range of f is the image of its domain, that is, f .A/. For example, the range

of the function f W N ! N deﬁned by f .n/ D 2n is f . N / D fm W m D 2n for

some n 2 N g, in other words, the set of nonnegative even integers.

A function is a surjection if its range is its codomain. For example, the function

f .n/ D bn=2c is a surjective function from N to N , since every element in N

appears as the value of f for some argument. In contrast, the function f .n/ D 2n

is not a surjective function from N to N , since no argument to f can produce 3 as a

value. The function f .n/ D 2n is, however, a surjective function from the natural

numbers to the even numbers. A surjection f W A ! B is sometimes described as

mapping A onto B. When we say that f is onto, we mean that it is surjective.

A function f W A ! B is an injection if distinct arguments to f produce

distinct values, that is, if a ¤ a 0

implies f .a/ ¤ f .a 0 /. For example, the function

f .n/ D 2n is an injective function from N to N , since each even number b is the

image under f of at most one element of the domain, namely b=2. The function

f .n/ D bn=2c is not injective, since the value 1 is produced by two arguments: 2

and 3. An injection is sometimes called a one-to-one function.

A function f W A ! B is a bijection if it is injective and surjective. For example,

the function f .n/ D .1/ n dn=2e is a bijection from N to Z :

0 ! 0 ;

1 ! 1 ;

2 ! 1 ;

3 ! 2 ;

4 ! 2 ;

:

:

:

The function is injective, since no element of Z is the image of more than one

element of N . It is surjective, since every element of Z appears as the image of

some element of N . Hence, the function is bijective. A bijection is sometimes

called a one-to-one correspondence, since it pairs elements in the domain and

codomain. A bijection from a set A to itself is sometimes called a permutation.

When a function f is bijective, we deﬁne its inverse f 1

as

f

1

.b/ D a if and only if f .a/ D b :

1168 Appendix B Sets, Etc.

For example, the inverse of the function f .n/ D .1/ n dn=2e is

f

1

.m/ D

(

2m if m 0 ;

2m 1 if m < 0 :

Exercises

B.3-1

Let A and B be ﬁnite sets, and let f W A ! B be a function. Show that

a. if f is injective, then jAj jBj;

b. if f is surjective, then jAj jBj.

B.3-2

Is the function f .x/ D x C 1 bijective when the domain and the codomain are N ?

Is it bijective when the domain and the codomain are Z ?

B.3-3

Give a natural deﬁnition for the inverse of a binary relation such that if a relation

is in fact a bijective function, its relational inverse is its functional inverse.

B.3-4 ?

Give a bijection from Z to Z Z .

B.4 Graphs

This section presents two kinds of graphs: directed and undirected. Certain def-

initions in the literature differ from those given here, but for the most part, the

differences are slight. Section 22.1 shows how we can represent graphs in com-

puter memory.

A directed graph (or digraph) G is a pair .V; E/, where V is a ﬁnite set and E

is a binary relation on V . The set V is called the vertex set of G, and its elements

are called vertices (singular: vertex). The set E is called the edge set of G, and its

elements are called edges. Figure B.2(a) is a pictorial representation of a directed

graph on the vertex set f1; 2; 3; 4; 5; 6g. Vertices are represented by circles in the

ﬁgure, and edges are represented by arrows. Note that self-loops—edges from a

vertex to itself—are possible.

In an undirected graph G D .V; E/, the edge set E consists of unordered

pairs of vertices, rather than ordered pairs. That is, an edge is a set fu; g, where

B.4 Graphs 1169

1 2 3

4 5 6

(a)

1 2 3

4 5 6

(b)

1 2 3

6

(c)

Figure B.2 Directed and undirected graphs. (a) A directed graph G D .V; E/, where V D

f1; 2; 3; 4; 5; 6g and E D f.1; 2/; .2; 2/; .2; 4/; .2; 5/; .4; 1/; .4; 5/; .5; 4/; .6; 3/g. The edge .2; 2/

is a self-loop. (b) An undirected graph G D .V; E/, where V D f1; 2; 3; 4; 5; 6g and E D

f.1; 2/; .1; 5/; .2; 5/; .3; 6/g. The vertex 4 is isolated. (c) The subgraph of the graph in part (a)

induced by the vertex set f1; 2; 3; 6g.

u; 2 V and u ¤ . By convention, we use the notation .u; / for an edge, rather

than the set notation fu; g, and we consider .u; / and .; u/ to be the same edge.

In an undirected graph, self-loops are forbidden, and so every edge consists of two

distinct vertices. Figure B.2(b) is a pictorial representation of an undirected graph

on the vertex set f1; 2; 3; 4; 5; 6g.

Many deﬁnitions for directed and undirected graphs are the same, although cer-

tain terms have slightly different meanings in the two contexts. If .u; / is an edge

in a directed graph G D .V; E/, we say that .u; / is incident from or leaves

vertex u and is incident to or enters vertex . For example, the edges leaving ver-

tex 2 in Figure B.2(a) are .2; 2/, .2; 4/, and .2; 5/. The edges entering vertex 2 are

.1; 2/ and .2; 2/. If .u; / is an edge in an undirected graph G D .V; E/, we say

that .u; / is incident on vertices u and . In Figure B.2(b), the edges incident on

vertex 2 are .1; 2/ and .2; 5/.

If .u; / is an edge in a graph G D .V; E/, we say that vertex is adjacent to

vertex u. When the graph is undirected, the adjacency relation is symmetric. When

the graph is directed, the adjacency relation is not necessarily symmetric. If is

adjacent to u in a directed graph, we sometimes write u ! . In parts (a) and (b)

of Figure B.2, vertex 2 is adjacent to vertex 1, since the edge .1; 2/ belongs to both

graphs. Vertex 1 is not adjacent to vertex 2 in Figure B.2(a), since the edge .2; 1/

does not belong to the graph.

The degree of a vertex in an undirected graph is the number of edges incident on

it. For example, vertex 2 in Figure B.2(b) has degree 2. A vertex whose degree is 0,

such as vertex 4 in Figure B.2(b), is isolated. In a directed graph, the out-degree

of a vertex is the number of edges leaving it, and the in-degree of a vertex is the

number of edges entering it. The degree of a vertex in a directed graph is its in-

1170 Appendix B Sets, Etc.

degree plus its out-degree. Vertex 2 in Figure B.2(a) has in-degree 2, out-degree 3,

and degree 5.

A path of length k from a vertex u to a vertex u 0

in a graph G D .V; E/

is a sequence h 0 ; 1 ; 2 ; : : : ; k i of vertices such that u D 0 , u 0 D k , and

. i1 ; i / 2 E for i D 1; 2; : : : ; k. The length of the path is the number of

edges in the path. The path contains the vertices 0 ; 1 ; : : : ; k and the edges

. 0 ; 1 /; . 1 ; 2 /; : : : ; . k1 ; k /. (There is always a 0-length path from u to u.) If

there is a path p from u to u 0

, we say that u 0

is reachable from u via p, which we

sometimes write as u

p u 0

if G is directed. A path is simple 4 if all vertices in the

path are distinct. In Figure B.2(a), the path h1; 2; 5; 4i is a simple path of length 3.

The path h2; 5; 4; 5i is not simple.

A subpath of path p D h 0 ; 1 ; : : : ; k i is a contiguous subsequence of its ver-

tices. That is, for any 0 i j k, the subsequence of vertices h i ; iC1 ; : : : ; j i

is a subpath of p.

In a directed graph, a path h 0 ; 1 ; : : : ; k i forms a cycle if 0 D k and the

path contains at least one edge. The cycle is simple if, in addition, 1 ; 2 ; : : : ; k

are distinct. A self-loop is a cycle of length 1. Two paths h 0 ; 1 ; 2 ; : : : ; k1 ; 0 i

and h 0

0

; 0

1

; 0

2

; : : : ; 0

k1

; 0

0

i form the same cycle if there exists an integer j such

that 0

i

D .iCj / mod k for i D 0; 1; : : : ; k 1. In Figure B.2(a), the path h1; 2; 4; 1i

forms the same cycle as the paths h2; 4; 1; 2i and h4; 1; 2; 4i. This cycle is simple,

but the cycle h1; 2; 4; 5; 4; 1i is not. The cycle h2; 2i formed by the edge .2; 2/ is

a self-loop. A directed graph with no self-loops is simple. In an undirected graph,

a path h 0 ; 1 ; : : : ; k i forms a cycle if k 3 and 0 D k ; the cycle is simple if

1 ; 2 ; : : : ; k are distinct. For example, in Figure B.2(b), the path h1; 2; 5; 1i is a

simple cycle. A graph with no cycles is acyclic.

An undirected graph is connected if every vertex is reachable from all other

vertices. The connected components of a graph are the equivalence classes of

vertices under the “is reachable from” relation. The graph in Figure B.2(b) has

three connected components: f1; 2; 5g, f3; 6g, and f4g. Every vertex in f1; 2; 5g is

reachable from every other vertex in f1; 2; 5g. An undirected graph is connected

if it has exactly one connected component. The edges of a connected component

are those that are incident on only the vertices of the component; in other words,

edge .u; / is an edge of a connected component only if both u and are vertices

of the component.

A directed graph is strongly connected if every two vertices are reachable from

each other. The strongly connected components of a directed graph are the equiv-

4

Some authors refer to what we call a path as a “walk” and to what we call a simple path as just a

“path.” We use the terms “path” and “simple path” throughout this book in a manner consistent with

their deﬁnitions.

B.4 Graphs 1171

1 2

3

4 5

6

u v w x y z

(a)

1 2

3

4

5

u v w x y

(b)

G

G ′

Figure B.3 (a) A pair of isomorphic graphs. The vertices of the top graph are mapped to the

vertices of the bottom graph by f .1/ D u; f .2/ D ; f .3/ D w; f .4/ D x; f .5/ D y; f .6/ D ´.

(b) Two graphs that are not isomorphic, since the top graph has a vertex of degree 4 and the bottom

graph does not.

alence classes of vertices under the “are mutually reachable” relation. A directed

graph is strongly connected if it has only one strongly connected component. The

graph in Figure B.2(a) has three strongly connected components: f1; 2; 4; 5g, f3g,

and f6g. All pairs of vertices in f1; 2; 4; 5g are mutually reachable. The ver-

tices f3; 6g do not form a strongly connected component, since vertex 6 cannot

be reached from vertex 3.

Two graphs G D .V; E/ and G 0 D .V 0 ; E 0 / are isomorphic if there exists a

bijection f W V ! V 0

such that .u; / 2 E if and only if .f .u/; f .// 2 E 0

.

In other words, we can relabel the vertices of G to be vertices of G 0

, maintain-

ing the corresponding edges in G and G 0

. Figure B.3(a) shows a pair of iso-

morphic graphs G and G 0

with respective vertex sets V D f1; 2; 3; 4; 5; 6g and

V 0 D fu; ; w; x; y; ´g. The mapping from V to V 0

given by f .1/ D u; f .2/ D ;

f .3/ D w; f .4/ D x; f .5/ D y; f .6/ D ´ provides the required bijective func-

tion. The graphs in Figure B.3(b) are not isomorphic. Although both graphs have

5 vertices and 7 edges, the top graph has a vertex of degree 4 and the bottom graph

does not.

We say that a graph G 0 D .V 0 ; E 0 / is a subgraph of G D .V; E/ if V 0 V

and E 0 E. Given a set V 0 V , the subgraph of G induced by V 0

is the graph

G 0 D .V 0 ; E 0 /, where

E

0

D f.u; / 2 E W u; 2 V

0

g :

1172 Appendix B Sets, Etc.

The subgraph induced by the vertex set f1; 2; 3; 6g in Figure B.2(a) appears in

Figure B.2(c) and has the edge set f.1; 2/; .2; 2/; .6; 3/g.

Given an undirected graph G D .V; E/, the directed version of G is the directed

graph G 0 D .V; E 0 /, where .u; / 2 E 0

if and only if .u; / 2 E. That is, we

replace each undirected edge .u; / in G by the two directed edges .u; / and .; u/

in the directed version. Given a directed graph G D .V; E/, the undirected version

of G is the undirected graph G 0 D .V; E 0 /, where .u; / 2 E 0

if and only if u ¤

and .u; / 2 E. That is, the undirected version contains the edges of G “with

their directions removed” and with self-loops eliminated. (Since .u; / and .; u/

are the same edge in an undirected graph, the undirected version of a directed

graph contains it only once, even if the directed graph contains both edges .u; /

and .; u/.) In a directed graph G D .V; E/, a neighbor of a vertex u is any vertex

that is adjacent to u in the undirected version of G. That is, is a neighbor of u if

u ¤ and either .u; / 2 E or .; u/ 2 E. In an undirected graph, u and are

neighbors if they are adjacent.

Several kinds of graphs have special names. A complete graph is an undirected

graph in which every pair of vertices is adjacent. A bipartite graph is an undirected

graph G D .V; E/ in which V can be partitioned into two sets V 1 and V 2 such that

.u; / 2 E implies either u 2 V 1 and 2 V 2 or u 2 V 2 and 2 V 1 . That is, all

edges go between the two sets V 1 and V 2 . An acyclic, undirected graph is a forest,

and a connected, acyclic, undirected graph is a (free) tree (see Section B.5). We

often take the ﬁrst letters of “directed acyclic graph” and call such a graph a dag.

There are two variants of graphs that you may occasionally encounter. A multi-

graph is like an undirected graph, but it can have both multiple edges between ver-

tices and self-loops. A hypergraph is like an undirected graph, but each hyperedge,

rather than connecting two vertices, connects an arbitrary subset of vertices. Many

algorithms written for ordinary directed and undirected graphs can be adapted to

run on these graphlike structures.

The contraction of an undirected graph G D .V; E/ by an edge e D .u; / is a

graph G 0 D .V 0 ; E 0 /, where V 0 D V fu; g [ fxg and x is a new vertex. The set

of edges E 0

is formed from E by deleting the edge .u; / and, for each vertex w

incident on u or , deleting whichever of .u; w/ and .; w/ is in E and adding the

new edge .x; w/. In effect, u and are “contracted” into a single vertex.

Exercises

B.4-1

Attendees of a faculty party shake hands to greet each other, and each professor

remembers how many times he or she shook hands. At the end of the party, the

department head adds up the number of times that each professor shook hands.

B.5 Trees 1173

Show that the result is even by proving the handshaking lemma: if G D .V; E/ is

an undirected graph, then

X

2V

degree./ D 2 jEj :

B.4-2

Show that if a directed or undirected graph contains a path between two vertices u

and , then it contains a simple path between u and . Show that if a directed graph

contains a cycle, then it contains a simple cycle.

B.4-3

Show that any connected, undirected graph G D .V; E/ satisﬁes jEj jV j 1.

B.4-4

Verify that in an undirected graph, the “is reachable from” relation is an equiv-

alence relation on the vertices of the graph. Which of the three properties of an

equivalence relation hold in general for the “is reachable from” relation on the

vertices of a directed graph?

B.4-5

What is the undirected version of the directed graph in Figure B.2(a)? What is the

directed version of the undirected graph in Figure B.2(b)?

B.4-6 ?

Show that we can represent a hypergraph by a bipartite graph if we let incidence in

the hypergraph correspond to adjacency in the bipartite graph. (Hint: Let one set

of vertices in the bipartite graph correspond to vertices of the hypergraph, and let

the other set of vertices of the bipartite graph correspond to hyperedges.)

B.5 Trees

As with graphs, there are many related, but slightly different, notions of trees. This

section presents deﬁnitions and mathematical properties of several kinds of trees.

Sections 10.4 and 22.1 describe how we can represent trees in computer memory.

B.5.1 Free trees

As deﬁned in Section B.4, a free tree is a connected, acyclic, undirected graph. We

often omit the adjective “free” when we say that a graph is a tree. If an undirected

graph is acyclic but possibly disconnected, it is a forest. Many algorithms that work

1174 Appendix B Sets, Etc.

(a) (b) (c)

Figure B.4 (a) A free tree. (b) A forest. (c) A graph that contains a cycle and is therefore neither

a tree nor a forest.

for trees also work for forests. Figure B.4(a) shows a free tree, and Figure B.4(b)

shows a forest. The forest in Figure B.4(b) is not a tree because it is not connected.

The graph in Figure B.4(c) is connected but neither a tree nor a forest, because it

contains a cycle.

The following theorem captures many important facts about free trees.

Theorem B.2 (Properties of free trees)

Let G D .V; E/ be an undirected graph. The following statements are equivalent.

1. G is a free tree.

2. Any two vertices in G are connected by a unique simple path.

3. G is connected, but if any edge is removed from E, the resulting graph is dis-

connected.

4. G is connected, and jEj D jV j 1.

5. G is acyclic, and jEj D jV j 1.

6. G is acyclic, but if any edge is added to E, the resulting graph contains a cycle.

Proof (1) ) (2): Since a tree is connected, any two vertices in G are connected

by at least one simple path. Suppose, for the sake of contradiction, that vertices u

and are connected by two distinct simple paths p 1 and p 2 , as shown in Figure B.5.

Let w be the vertex at which the paths ﬁrst diverge; that is, w is the ﬁrst vertex

on both p 1 and p 2 whose successor on p 1 is x and whose successor on p 2 is y,

where x ¤ y. Let ´ be the ﬁrst vertex at which the paths reconverge; that is, ´ is

the ﬁrst vertex following w on p 1 that is also on p 2 . Let p 0

be the subpath of p 1

from w through x to ´, and let p 00

be the subpath of p 2 from w through y to ´.

Paths p 0

and p 00

share no vertices except their endpoints. Thus, the path obtained by

concatenating p 0

and the reverse of p 00

is a cycle, which contradicts our assumption

B.5 Trees 1175

u

w

z

v

x

y

p ′

p ′′

Figure B.5 A step in the proof of Theorem B.2: if (1) G is a free tree, then (2) any two vertices

in G are connected by a unique simple path. Assume for the sake of contradiction that vertices u

and are connected by two distinct simple paths p1 and p2. These paths ﬁrst diverge at vertex w,

and they ﬁrst reconverge at vertex ´. The path p

0

concatenated with the reverse of the path p

00

forms

a cycle, which yields the contradiction.

that G is a tree. Thus, if G is a tree, there can be at most one simple path between

two vertices.

(2) ) (3): If any two vertices in G are connected by a unique simple path,

then G is connected. Let .u; / be any edge in E. This edge is a path from u to ,

and so it must be the unique path from u to . If we remove .u; / from G, there

is no path from u to , and hence its removal disconnects G.

(3) ) (4): By assumption, the graph G is connected, and by Exercise B.4-3, we

have jEj jV j 1. We shall prove jEj jV j 1 by induction. A connected

graph with n D 1 or n D 2 vertices has n 1 edges. Suppose that G has n 3

vertices and that all graphs satisfying (3) with fewer than n vertices also satisfy

jEj jV j 1. Removing an arbitrary edge from G separates the graph into k 2

connected components (actually k D 2). Each component satisﬁes (3), or else G

would not satisfy (3). If we view each connected component V i , with edge set E i ,

as its own free tree, then because each component has fewer than jV j vertices, by

the inductive hypothesis we have jE i j jV i j 1. Thus, the number of edges in all

components combined is at most jV j k jV j 2. Adding in the removed edge

yields jEj jV j 1.

(4) ) (5): Suppose that G is connected and that jEj D jV j 1. We must show

that G is acyclic. Suppose that G has a cycle containing k vertices 1 ; 2 ; : : : ; k ,

and without loss of generality assume that this cycle is simple. Let G k D .V k ; E k /

be the subgraph of G consisting of the cycle. Note that jV k j D jE k j D k.

If k < jV j, there must be a vertex kC1 2 V V k that is adjacent to some ver-

tex i 2 V k , since G is connected. Deﬁne G kC1 D .V kC1 ; E kC1 / to be the sub-

graph of G with V kC1 D V k [ f kC1 g and E kC1 D E k [ f. i ; kC1 /g. Note that

jV kC1 j D jE kC1 j D k C 1. If k C 1 < jV j, we can continue, deﬁning G kC2 in

the same manner, and so forth, until we obtain G n D .V n ; E n /, where n D jV j,

1176 Appendix B Sets, Etc.

V n D V , and jE n j D jV n j D jV j. Since G n is a subgraph of G, we have E n E,

and hence jEj jV j, which contradicts the assumption that jEj D jV j 1. Thus,

G is acyclic.

(5) ) (6): Suppose that G is acyclic and that jEj D jV j 1. Let k be the

number of connected components of G. Each connected component is a free tree

by deﬁnition, and since (1) implies (5), the sum of all edges in all connected com-

ponents of G is jV j k. Consequently, we must have k D 1, and G is in fact a

tree. Since (1) implies (2), any two vertices in G are connected by a unique simple

path. Thus, adding any edge to G creates a cycle.

(6) ) (1): Suppose that G is acyclic but that adding any edge to E creates a

cycle. We must show that G is connected. Let u and be arbitrary vertices in G.

If u and are not already adjacent, adding the edge .u; / creates a cycle in which

all edges but .u; / belong to G. Thus, the cycle minus edge .u; / must contain a

path from u to , and since u and were chosen arbitrarily, G is connected.

B.5.2 Rooted and ordered trees

A rooted tree is a free tree in which one of the vertices is distinguished from the

others. We call the distinguished vertex the root of the tree. We often refer to a

vertex of a rooted tree as a node 5 of the tree. Figure B.6(a) shows a rooted tree on

a set of 12 nodes with root 7.

Consider a node x in a rooted tree T with root r. We call any node y on the

unique simple path from r to x an ancestor of x. If y is an ancestor of x, then x is

a descendant of y. (Every node is both an ancestor and a descendant of itself.) If y

is an ancestor of x and x ¤ y, then y is a proper ancestor of x and x is a proper

descendant of y. The subtree rooted at x is the tree induced by descendants of x,

rooted at x. For example, the subtree rooted at node 8 in Figure B.6(a) contains

nodes 8, 6, 5, and 9.

If the last edge on the simple path from the root r of a tree T to a node x is .y; x/,

then y is the parent of x, and x is a child of y. The root is the only node in T with

no parent. If two nodes have the same parent, they are siblings. A node with no

children is a leaf or external node. A nonleaf node is an internal node.

5

The term “node” is often used in the graph theory literature as a synonym for “vertex.” We reserve

the term “node” to mean a vertex of a rooted tree.

B.5 Trees 1177

9

6 5

8

1

12

3 10

7

11 2

4

height = 4

depth 0

depth 1

depth 2

depth 3

depth 4

(a)

9

6 5

8 12

3 10

7

11 2

4

(b)

1

Figure B.6 Rooted and ordered trees. (a) A rooted tree with height 4. The tree is drawn in a

standard way: the root (node 7) is at the top, its children (nodes with depth 1) are beneath it, their

children (nodes with depth 2) are beneath them, and so forth. If the tree is ordered, the relative left-

to-right order of the children of a node matters; otherwise it doesn’t. (b) Another rooted tree. As a

rooted tree, it is identical to the tree in (a), but as an ordered tree it is different, since the children of

node 3 appear in a different order.

The number of children of a node x in a rooted tree T equals the degree of x. 6

The length of the simple path from the root r to a node x is the depth of x in T .

A level of a tree consists of all nodes at the same depth. The height of a node in a

tree is the number of edges on the longest simple downward path from the node to

a leaf, and the height of a tree is the height of its root. The height of a tree is also

equal to the largest depth of any node in the tree.

An ordered tree is a rooted tree in which the children of each node are ordered.

That is, if a node has k children, then there is a ﬁrst child, a second child, . . . ,

and a kth child. The two trees in Figure B.6 are different when considered to be

ordered trees, but the same when considered to be just rooted trees.

B.5.3 Binary and positional trees

We deﬁne binary trees recursively. A binary tree T is a structure deﬁned on a ﬁnite

set of nodes that either

contains no nodes, or

6

Notice that the degree of a node depends on whether we consider T to be a rooted tree or a free tree.

The degree of a vertex in a free tree is, as in any undirected graph, the number of adjacent vertices.

In a rooted tree, however, the degree is the number of children—the parent of a node does not count

toward its degree.

1178 Appendix B Sets, Etc.

3

2

4 1

6

7

5

(a)

3

2

4 1

6

7

5

(b)

3

2

4 1

6

7

5

(c)

Figure B.7 Binary trees. (a) A binary tree drawn in a standard way. The left child of a node is

drawn beneath the node and to the left. The right child is drawn beneath and to the right. (b) A binary

tree different from the one in (a). In (a), the left child of node 7 is 5 and the right child is absent.

In (b), the left child of node 7 is absent and the right child is 5. As ordered trees, these trees are

the same, but as binary trees, they are distinct. (c) The binary tree in (a) represented by the internal

nodes of a full binary tree: an ordered tree in which each internal node has degree 2. The leaves in

the tree are shown as squares.

is composed of three disjoint sets of nodes: a root node, a binary tree called its

left subtree, and a binary tree called its right subtree.

The binary tree that contains no nodes is called the empty tree or null tree, some-

times denoted NIL. If the left subtree is nonempty, its root is called the left child of

the root of the entire tree. Likewise, the root of a nonnull right subtree is the right

child of the root of the entire tree. If a subtree is the null tree NIL, we say that the

child is absent or missing. Figure B.7(a) shows a binary tree.

A binary tree is not simply an ordered tree in which each node has degree at

most 2. For example, in a binary tree, if a node has just one child, the position

of the child—whether it is the left child or the right child—matters. In an or-

dered tree, there is no distinguishing a sole child as being either left or right. Fig-

ure B.7(b) shows a binary tree that differs from the tree in Figure B.7(a) because of

the position of one node. Considered as ordered trees, however, the two trees are

identical.

We can represent the positioning information in a binary tree by the internal

nodes of an ordered tree, as shown in Figure B.7(c). The idea is to replace each

missing child in the binary tree with a node having no children. These leaf nodes

are drawn as squares in the ﬁgure. The tree that results is a full binary tree: each

node is either a leaf or has degree exactly 2. There are no degree-1 nodes. Conse-

quently, the order of the children of a node preserves the position information.

We can extend the positioning information that distinguishes binary trees from

ordered trees to trees with more than 2 children per node. In a positional tree, the

B.5 Trees 1179

height = 3

depth 0

depth 1

depth 2

depth 3

Figure B.8 A complete binary tree of height 3 with 8 leaves and 7 internal nodes.

children of a node are labeled with distinct positive integers. The ith child of a

node is absent if no child is labeled with integer i. A k-ary tree is a positional tree

in which for every node, all children with labels greater than k are missing. Thus,

a binary tree is a k-ary tree with k D 2.

A complete k-ary tree is a k-ary tree in which all leaves have the same depth

and all internal nodes have degree k. Figure B.8 shows a complete binary tree of

height 3. How many leaves does a complete k-ary tree of height h have? The root

has k children at depth 1, each of which has k children at depth 2, etc. Thus, the

number of leaves at depth h is k h

. Consequently, the height of a complete k-ary

tree with n leaves is log k n. The number of internal nodes of a complete k-ary tree

of height h is

1 C k C k

2

C C k

h1

D

h1 X

iD0

k

i

D

k h 1

k 1

by equation (A.5). Thus, a complete binary tree has 2 h 1 internal nodes.

Exercises

B.5-1

Draw all the free trees composed of the three vertices x, y, and ´. Draw all the

rooted trees with nodes x, y, and ´ with x as the root. Draw all the ordered trees

with nodes x, y, and ´ with x as the root. Draw all the binary trees with nodes x,

y, and ´ with x as the root.

1180 Appendix B Sets, Etc.

B.5-2

Let G D .V; E/ be a directed acyclic graph in which there is a vertex 0 2 V

such that there exists a unique path from 0 to every vertex 2 V . Prove that the

undirected version of G forms a tree.

B.5-3

Show by induction that the number of degree-2 nodes in any nonempty binary tree

is 1 fewer than the number of leaves. Conclude that the number of internal nodes

in a full binary tree is 1 fewer than the number of leaves.

B.5-4

Use induction to show that a nonempty binary tree with n nodes has height at

least blg nc.

B.5-5 ?

The internal path length of a full binary tree is the sum, taken over all internal

nodes of the tree, of the depth of each node. Likewise, the external path length is

the sum, taken over all leaves of the tree, of the depth of each leaf. Consider a full

binary tree with n internal nodes, internal path length i, and external path length e.

Prove that e D i C 2n.

B.5-6 ?

Let us associate a “weight” w.x/ D 2 d

with each leaf x of depth d in a binary

tree T , and let L be the set of leaves of T . Prove that

P

x2L

w.x/ 1. (This is

known as the Kraft inequality.)

B.5-7 ?

Show that if L 2, then every binary tree with L leaves contains a subtree having

between L=3 and 2L=3 leaves, inclusive.

Problems

B-1 Graph coloring

Given an undirected graph G D .V; E/, a k-coloring of G is a function c W V !

f0; 1; : : : ; k 1g such that c.u/ ¤ c./ for every edge .u; / 2 E. In other words,

the numbers 0; 1; : : : ; k 1 represent the k colors, and adjacent vertices must have

different colors.

a. Show that any tree is 2-colorable.

Problems for Appendix B 1181

b. Show that the following are equivalent:

1. G is bipartite.

2. G is 2-colorable.

3. G has no cycles of odd length.

c. Let d be the maximum degree of any vertex in a graph G. Prove that we can

color G with d C 1 colors.

d. Show that if G has O.jV j/ edges, then we can color G with O.

p

jV j/ colors.

B-2 Friendly graphs

Reword each of the following statements as a theorem about undirected graphs,

and then prove it. Assume that friendship is symmetric but not reﬂexive.

a. Any group of at least two people contains at least two people with the same

number of friends in the group.

b. Every group of six people contains either at least three mutual friends or at least

three mutual strangers.

c. Any group of people can be partitioned into two subgroups such that at least

half the friends of each person belong to the subgroup of which that person is

not a member.

d. If everyone in a group is the friend of at least half the people in the group, then

the group can be seated around a table in such a way that everyone is seated

between two friends.

B-3 Bisecting trees

Many divide-and-conquer algorithms that operate on graphs require that the graph

be bisected into two nearly equal-sized subgraphs, which are induced by a partition

of the vertices. This problem investigates bisections of trees formed by removing a

small number of edges. We require that whenever two vertices end up in the same

subtree after removing edges, then they must be in the same partition.

a. Show that we can partition the vertices of any n-vertex binary tree into two

sets A and B, such that jAj 3n=4 and jBj 3n=4, by removing a single

edge.

b. Show that the constant 3=4 in part (a) is optimal in the worst case by giving

an example of a simple binary tree whose most evenly balanced partition upon

removal of a single edge has jAj D 3n=4.

1182 Appendix B Sets, Etc.

c. Show that by removing at most O.lg n/ edges, we can partition the vertices

of any n-vertex binary tree into two sets A and B such that jAj D bn=2c

and jBj D dn=2e.

Appendix notes

G. Boole pioneered the development of symbolic logic, and he introduced many of

the basic set notations in a book published in 1854. Modern set theory was created

by G. Cantor during the period 1874–1895. Cantor focused primarily on sets of

inﬁnite cardinality. The term “function” is attributed to G. W. Leibniz, who used it

to refer to several kinds of mathematical formulas. His limited deﬁnition has been

generalized many times. Graph theory originated in 1736, when L. Euler proved

that it was impossible to cross each of the seven bridges in the city of K¨onigsberg

exactly once and return to the starting point.

The book by Harary [160] provides a useful compendium of many deﬁnitions

and results from graph theory.

C Counting and Probability

This appendix reviews elementary combinatorics and probability theory. If you

have a good background in these areas, you may want to skim the beginning of this

appendix lightly and concentrate on the later sections. Most of this book’s chapters

do not require probability, but for some chapters it is essential.

Section C.1 reviews elementary results in counting theory, including standard

formulas for counting permutations and combinations. The axioms of probability

and basic facts concerning probability distributions form Section C.2. Random

variables are introduced in Section C.3, along with the properties of expectation

and variance. Section C.4 investigates the geometric and binomial distributions

that arise from studying Bernoulli trials. The study of the binomial distribution

continues in Section C.5, an advanced discussion of the “tails” of the distribution.

C.1 Counting

Counting theory tries to answer the question “How many?” without actually enu-

merating all the choices. For example, we might ask, “How many different n-bit

numbers are there?” or “How many orderings of n distinct elements are there?” In

this section, we review the elements of counting theory. Since some of the material

assumes a basic understanding of sets, you might wish to start by reviewing the

material in Section B.1.

Rules of sum and product

We can sometimes express a set of items that we wish to count as a union of disjoint

sets or as a Cartesian product of sets.

The rule of sum says that the number of ways to choose one element from one

of two disjoint sets is the sum of the cardinalities of the sets. That is, if A and B

are two ﬁnite sets with no members in common, then jA [ Bj D jAj C jBj, which

1184 Appendix C Counting and Probability

follows from equation (B.3). For example, each position on a car’s license plate

is a letter or a digit. The number of possibilities for each position is therefore

26 C 10 D 36, since there are 26 choices if it is a letter and 10 choices if it is a

digit.

The rule of product says that the number of ways to choose an ordered pair is the

number of ways to choose the ﬁrst element times the number of ways to choose the

second element. That is, if A and B are two ﬁnite sets, then jA Bj D jAj jBj,

which is simply equation (B.4). For example, if an ice-cream parlor offers 28

ﬂavors of ice cream and 4 toppings, the number of possible sundaes with one scoop

of ice cream and one topping is 28 4 D 112.

Strings

A string over a ﬁnite set S is a sequence of elements of S. For example, there are 8

binary strings of length 3:

000; 001; 010; 011; 100; 101; 110; 111 :

We sometimes call a string of length k a k-string. A substring s 0

of a string s

is an ordered sequence of consecutive elements of s. A k-substring of a string

is a substring of length k. For example, 010 is a 3-substring of 01101001 (the

3-substring that begins in position 4), but 111 is not a substring of 01101001.

We can view a k-string over a set S as an element of the Cartesian product S k

of k-tuples; thus, there are jSj

k

strings of length k. For example, the number of

binary k-strings is 2 k

. Intuitively, to construct a k-string over an n-set, we have n

ways to pick the ﬁrst element; for each of these choices, we have n ways to pick the

second element; and so forth k times. This construction leads to the k-fold product

n n n D n k

as the number of k-strings.

Permutations

A permutation of a ﬁnite set S is an ordered sequence of all the elements of S,

with each element appearing exactly once. For example, if S D fa; b; cg, then S

has 6 permutations:

abc; acb; bac; bca; cab; cba :

There are nŠ permutations of a set of n elements, since we can choose the ﬁrst

element of the sequence in n ways, the second in n 1 ways, the third in n 2

ways, and so on.

A k-permutation of S is an ordered sequence of k elements of S, with no ele-

ment appearing more than once in the sequence. (Thus, an ordinary permutation is

an n-permutation of an n-set.) The twelve 2-permutations of the set fa; b; c; dg are

C.1 Counting 1185

ab; ac; ad; ba; bc; bd; ca; cb; cd; da; db; dc :

The number of k-permutations of an n-set is

n.n 1/.n 2/ .n k C 1/ D

nŠ

.n k/Š

; (C.1)

since we have n ways to choose the ﬁrst element, n 1 ways to choose the second

element, and so on, until we have selected k elements, the last being a selection

from the remaining n k C 1 elements.

Combinations

A k-combination of an n-set S is simply a k-subset of S. For example, the 4-set

fa; b; c; dg has six 2-combinations:

ab; ac; ad; bc; bd; cd :

(Here we use the shorthand of denoting the 2-subset fa; bg by ab, and so on.)

We can construct a k-combination of an n-set by choosing k distinct (different)

elements from the n-set. The order in which we select the elements does not matter.

We can express the number of k-combinations of an n-set in terms of the number

of k-permutations of an n-set. Every k-combination has exactly kŠ permutations

of its elements, each of which is a distinct k-permutation of the n-set. Thus, the

number of k-combinations of an n-set is the number of k-permutations divided

by kŠ; from equation (C.1), this quantity is

nŠ

kŠ .n k/Š

: (C.2)

For k D 0, this formula tells us that the number of ways to choose 0 elements from

an n-set is 1 (not 0), since 0Š D 1.

Binomial coefﬁcients

The notation

n

k

(read “n choose k”) denotes the number of k-combinations of

an n-set. From equation (C.2), we have

n

k

!

D

nŠ

kŠ .n k/Š

:

This formula is symmetric in k and n k:

n

k

!

D

n

n k

!

: (C.3)

1186 Appendix C Counting and Probability

These numbers are also known as binomial coefﬁcients, due to their appearance in

the binomial expansion:

.x C y/

n

D

n X

kD0

n

k

!

x

k

y

nk

: (C.4)

A special case of the binomial expansion occurs when x D y D 1:

2

n

D

n X

kD0

n

k

!

:

This formula corresponds to counting the 2 n

binary n-strings by the number of 1s

they contain:

n

k

binary n-strings contain exactly k 1s, since we have

n

k

ways to

choose k out of the n positions in which to place the 1s.

Many identities involve binomial coefﬁcients. The exercises at the end of this

section give you the opportunity to prove a few.

Binomial bounds

We sometimes need to bound the size of a binomial coefﬁcient. For 1 k n,

we have the lower bound

n

k

!

D

n.n 1/ .n k C 1/

k.k 1/ 1

D

n

k

n 1

k 1

n k C 1

1

n

k

k

:

Taking advantage of the inequality kŠ .k=e/ k

derived from Stirling’s approxi-

mation (3.18), we obtain the upper bounds

n

k

!

D

n.n 1/ .n k C 1/

k.k 1/ 1

n k

kŠ

en

k

k

: (C.5)

For all integers k such that 0 k n, we can use induction (see Exercise C.1-12)

to prove the bound

C.1 Counting 1187

n

k

!

n n

k k .n k/ nk

; (C.6)

where for convenience we assume that 0 0 D 1. For k D n, where 0 1, we

can rewrite this bound as

n

n

!

n n

.n/ n ..1 /n/ .1 /n

D

1

1

1

1

! n

D 2

n H. /

;

where

H./ D lg .1 / lg.1 / (C.7)

is the (binary) entropy function and where, for convenience, we assume that

0 lg 0 D 0, so that H.0/ D H.1/ D 0.

Exercises

C.1-1

How many k-substrings does an n-string have? (Consider identical k-substrings at

different positions to be different.) How many substrings does an n-string have in

total?

C.1-2

An n-input, m-output boolean function is a function from fTRUE; FALSEg

n

to

fTRUE; FALSEg

m

. How many n-input, 1-output boolean functions are there? How

many n-input, m-output boolean functions are there?

C.1-3

In how many ways can n professors sit around a circular conference table? Con-

sider two seatings to be the same if one can be rotated to form the other.

C.1-4

In how many ways can we choose three distinct numbers from the set f1; 2; : : : ; 99g

so that their sum is even?

1188 Appendix C Counting and Probability

C.1-5

Prove the identity

n

k

!

D

n

k

n 1

k 1

!

(C.8)

for 0 < k n.

C.1-6

Prove the identity

n

k

!

D

n

n k

n 1

k

!

for 0 k < n.

C.1-7

To choose k objects from n, you can make one of the objects distinguished and

consider whether the distinguished object is chosen. Use this approach to prove

that

n

k

!

D

n 1

k

!

C

n 1

k 1

!

:

C.1-8

Using the result of Exercise C.1-7, make a table for n D 0; 1; : : : ; 6 and 0 k n

of the binomial coefﬁcients

n

k

with

0

0

at the top,

1

0

and

1

1

on the next line, and

so forth. Such a table of binomial coefﬁcients is called Pascal’s triangle.

C.1-9

Prove that

n X

iD1

i D

n C 1

2

!

:

C.1-10

Show that for any integers n 0 and 0 k n, the expression

n

k

achieves its

maximum value when k D bn=2c or k D dn=2e.

C.1-11 ?

Argue that for any integers n 0, j 0, k 0, and j C k n,

n

j C k

!

n

j

!

n j

k

!

: (C.9)

C.2 Probability 1189

Provide both an algebraic proof and an argument based on a method for choosing

j C k items out of n. Give an example in which equality does not hold.

C.1-12 ?

Use induction on all integers k such that 0 k n=2 to prove inequality (C.6),

and use equation (C.3) to extend it to all integers k such that 0 k n.

C.1-13 ?

Use Stirling’s approximation to prove that

2n

n

!

D

2 2n

p

n

.1 C O.1=n// : (C.10)

C.1-14 ?

By differentiating the entropy function H./, show that it achieves its maximum

value at D 1=2. What is H.1=2/?

C.1-15 ?

Show that for any integer n 0,

n X

kD0

n

k

!

k D n2

n1

: (C.11)

C.2 Probability

Probability is an essential tool for the design and analysis of probabilistic and ran-

domized algorithms. This section reviews basic probability theory.

We deﬁne probability in terms of a sample space S, which is a set whose ele-

ments are called elementary events. We can think of each elementary event as a

possible outcome of an experiment. For the experiment of ﬂipping two distinguish-

able coins, with each individual ﬂip resulting in a head (H) or a tail (T), we can view

the sample space as consisting of the set of all possible 2-strings over fH; Tg:

S D fHH; HT; TH; TTg :

1190 Appendix C Counting and Probability

An event is a subset 1 of the sample space S. For example, in the experiment of

ﬂipping two coins, the event of obtaining one head and one tail is fHT; THg. The

event S is called the certain event, and the event ; is called the null event. We say

that two events A and B are mutually exclusive if A \ B D ;. We sometimes treat

an elementary event s 2 S as the event fsg. By deﬁnition, all elementary events

are mutually exclusive.

Axioms of probability

A probability distribution Pr fg on a sample space S is a mapping from events of S

to real numbers satisfying the following probability axioms:

1. Pr fAg 0 for any event A.

2. Pr fSg D 1.

3. Pr fA [ Bg D Pr fAg C Pr fBg for any two mutually exclusive events A

and B. More generally, for any (ﬁnite or countably inﬁnite) sequence of events

A 1 ; A 2 ; : : : that are pairwise mutually exclusive,

Pr

(

[

i

A i

)

D

X

i

Pr fA i g :

We call Pr fAg the probability of the event A. We note here that axiom 2 is a

normalization requirement: there is really nothing fundamental about choosing 1

as the probability of the certain event, except that it is natural and convenient.

Several results follow immediately from these axioms and basic set theory (see

Section B.1). The null event ; has probability Pr f;g D 0. If A B, then

Pr fAg Pr fBg. Using A to denote the event S A (the complement of A),

we have Pr

˚

A

D 1 Pr fAg. For any two events A and B,

Pr fA [ Bg D Pr fAg C Pr fBg Pr fA \ Bg (C.12)

Pr fAg C Pr fBg : (C.13)

1

For a general probability distribution, there may be some subsets of the sample space S that are not

considered to be events. This situation usually arises when the sample space is uncountably inﬁnite.

The main requirement for what subsets are events is that the set of events of a sample space be closed

under the operations of taking the complement of an event, forming the union of a ﬁnite or countable

number of events, and taking the intersection of a ﬁnite or countable number of events. Most of

the probability distributions we shall see are over ﬁnite or countable sample spaces, and we shall

generally consider all subsets of a sample space to be events. A notable exception is the continuous

uniform probability distribution, which we shall see shortly.

C.2 Probability 1191

In our coin-ﬂipping example, suppose that each of the four elementary events

has probability 1=4. Then the probability of getting at least one head is

Pr fHH; HT; THg D Pr fHHg C Pr fHTg C Pr fTHg

D 3=4 :

Alternatively, since the probability of getting strictly less than one head is

Pr fTTg D 1=4, the probability of getting at least one head is 1 1=4 D 3=4.

Discrete probability distributions

A probability distribution is discrete if it is deﬁned over a ﬁnite or countably inﬁnite

sample space. Let S be the sample space. Then for any event A,

Pr fAg D

X

s2A

Pr fsg ;

since elementary events, speciﬁcally those in A, are mutually exclusive. If S is

ﬁnite and every elementary event s 2 S has probability

Pr fsg D 1= jSj ;

then we have the uniform probability distribution on S. In such a case the experi-

ment is often described as “picking an element of S at random.”

As an example, consider the process of ﬂipping a fair coin, one for which the

probability of obtaining a head is the same as the probability of obtaining a tail, that

is, 1=2. If we ﬂip the coin n times, we have the uniform probability distribution

deﬁned on the sample space S D fH; Tg

n

, a set of size 2 n

. We can represent each

elementary event in S as a string of length n over fH; Tg, each string occurring with

probability 1=2 n

. The event

A D fexactly k heads and exactly n k tails occurg

is a subset of S of size jAj D

n

k

, since

n

k

strings of length n over fH; Tg contain

exactly k H’s. The probability of event A is thus Pr fAg D

n

k

=2 n

.

Continuous uniform probability distribution

The continuous uniform probability distribution is an example of a probability

distribution in which not all subsets of the sample space are considered to be

events. The continuous uniform probability distribution is deﬁned over a closed

interval Œa; b of the reals, where a < b. Our intuition is that each point in the in-

terval Œa; b should be “equally likely.” There are an uncountable number of points,

however, so if we give all points the same ﬁnite, positive probability, we cannot si-

multaneously satisfy axioms 2 and 3. For this reason, we would like to associate a

1192 Appendix C Counting and Probability

probability only with some of the subsets of S, in such a way that the axioms are

satisﬁed for these events.

For any closed interval Œc; d, where a c d b, the continuous uniform

probability distribution deﬁnes the probability of the event Œc; d to be

Pr fŒc; dg D

d c

b a

:

Note that for any point x D Œx; x, the probability of x is 0. If we remove

the endpoints of an interval Œc; d, we obtain the open interval .c; d/. Since

Œc; d D Œc; c [ .c; d/ [ Œd; d, axiom 3 gives us Pr fŒc; dg D Pr f.c; d/g. Gen-

erally, the set of events for the continuous uniform probability distribution contains

any subset of the sample space Œa; b that can be obtained by a ﬁnite or countable

union of open and closed intervals, as well as certain more complicated sets.

Conditional probability and independence

Sometimes we have some prior partial knowledge about the outcome of an exper-

iment. For example, suppose that a friend has ﬂipped two fair coins and has told

you that at least one of the coins showed a head. What is the probability that both

coins are heads? The information given eliminates the possibility of two tails. The

three remaining elementary events are equally likely, so we infer that each occurs

with probability 1=3. Since only one of these elementary events shows two heads,

the answer to our question is 1=3.

Conditional probability formalizes the notion of having prior partial knowledge

of the outcome of an experiment. The conditional probability of an event A given

that another event B occurs is deﬁned to be

Pr fA j Bg D

Pr fA \ Bg

Pr fBg

(C.14)

whenever Pr fBg ¤ 0. (We read “Pr fA j Bg” as “the probability of A given B.”)

Intuitively, since we are given that event B occurs, the event that A also occurs

is A \ B. That is, A \ B is the set of outcomes in which both A and B occur.

Because the outcome is one of the elementary events in B, we normalize the prob-

abilities of all the elementary events in B by dividing them by Pr fBg, so that they

sum to 1. The conditional probability of A given B is, therefore, the ratio of the

probability of event A \ B to the probability of event B. In the example above, A

is the event that both coins are heads, and B is the event that at least one coin is a

head. Thus, Pr fA j Bg D .1=4/=.3=4/ D 1=3.

Two events are independent if

Pr fA \ Bg D Pr fAg Pr fBg ; (C.15)

which is equivalent, if Pr fBg ¤ 0, to the condition

C.2 Probability 1193

Pr fA j Bg D Pr fAg :

For example, suppose that we ﬂip two fair coins and that the outcomes are inde-

pendent. Then the probability of two heads is .1=2/.1=2/ D 1=4. Now suppose

that one event is that the ﬁrst coin comes up heads and the other event is that the

coins come up differently. Each of these events occurs with probability 1=2, and

the probability that both events occur is 1=4; thus, according to the deﬁnition of

independence, the events are independent—even though you might think that both

events depend on the ﬁrst coin. Finally, suppose that the coins are welded to-

gether so that they both fall heads or both fall tails and that the two possibilities are

equally likely. Then the probability that each coin comes up heads is 1=2, but the

probability that they both come up heads is 1=2 ¤ .1=2/.1=2/. Consequently, the

event that one comes up heads and the event that the other comes up heads are not

independent.

A collection A 1 ; A 2 ; : : : ; A n of events is said to be pairwise independent if

Pr fA i \ A j g D Pr fA i g Pr fA j g

for all 1 i < j n. We say that the events of the collection are (mutually)

independent if every k-subset A i 1 ; A i 2 ; : : : ; A i k of the collection, where 2 k n

and 1 i 1 < i 2 < < i k n, satisﬁes

Pr fA i 1 \ A i 2 \ \ A i k g D Pr fA i 1 g Pr fA i 2 g Pr fA i k g :

For example, suppose we ﬂip two fair coins. Let A 1 be the event that the ﬁrst coin

is heads, let A 2 be the event that the second coin is heads, and let A 3 be the event

that the two coins are different. We have

Pr fA 1 g D 1=2 ;

Pr fA 2 g D 1=2 ;

Pr fA 3 g D 1=2 ;

Pr fA 1 \ A 2 g D 1=4 ;

Pr fA 1 \ A 3 g D 1=4 ;

Pr fA 2 \ A 3 g D 1=4 ;

Pr fA 1 \ A 2 \ A 3 g D 0 :

Since for 1 i < j 3, we have Pr fA i \ A j g D Pr fA i g Pr fA j g D 1=4, the

events A 1 , A 2 , and A 3 are pairwise independent. The events are not mutually inde-

pendent, however, because Pr fA 1 \ A 2 \ A 3 g D 0 and Pr fA 1 g Pr fA 2 g Pr fA 3 g D

1=8 ¤ 0.

1194 Appendix C Counting and Probability

Bayes’s theorem

From the deﬁnition of conditional probability (C.14) and the commutative law

A \ B D B \ A, it follows that for two events A and B, each with nonzero

probability,

Pr fA \ Bg D Pr fBg Pr fA j Bg (C.16)

D Pr fAg Pr fB j Ag :

Solving for Pr fA j Bg, we obtain

Pr fA j Bg D

Pr fAg Pr fB j Ag

Pr fBg

; (C.17)

which is known as Bayes’s theorem. The denominator Pr fBg is a normalizing

constant, which we can reformulate as follows. Since B D .B \ A/ [ .B \ A/,

and since B \ A and B \ A are mutually exclusive events,

Pr fBg D Pr fB \ Ag C Pr

˚

B \ A

D Pr fAg Pr fB j Ag C Pr

˚

A

Pr

˚

B j A

:

Substituting into equation (C.17), we obtain an equivalent form of Bayes’s theo-

rem:

Pr fA j Bg D

Pr fAg Pr fB j Ag

Pr fAg Pr fB j Ag C Pr

˚

A

Pr

˚

B j A

: (C.18)

Bayes’s theorem can simplify the computing of conditional probabilities. For

example, suppose that we have a fair coin and a biased coin that always comes up

heads. We run an experiment consisting of three independent events: we choose

one of the two coins at random, we ﬂip that coin once, and then we ﬂip it again.

Suppose that the coin we have chosen comes up heads both times. What is the

probability that it is biased?

We solve this problem using Bayes’s theorem. Let A be the event that we choose

the biased coin, and let B be the event that the chosen coin comes up heads both

times. We wish to determine Pr fA j Bg. We have Pr fAg D 1=2, Pr fB j Ag D 1,

Pr

˚

A

D 1=2, and Pr

˚

B j A

D 1=4; hence,

Pr fA j Bg D

.1=2/ 1

.1=2/ 1 C .1=2/ .1=4/

D 4=5 :

Exercises

C.2-1

Professor Guildenstern ﬂips a fair

?

coin twice. What is the probability that Professor Rosencrantz obtains more heads

Professor Rosencrantz ﬂips a fair coin once.

than Professor Guildenstern

C.2 Probability 1195

C.2-2

Prove Boole’s inequality: For any ﬁnite or countably inﬁnite sequence of events

A 1 ; A 2 ; : : :,

Pr fA 1 [ A 2 [ g Pr fA 1 g C Pr fA 2 g C : (C.19)

C.2-3

Suppose we shufﬂe a deck of 10 cards, each bearing a distinct number from 1 to 10,

to mix the cards thoroughly. We then remove three cards, one at a time, from the

deck. What is the probability that we select the three cards in sorted (increasing)

order?

C.2-4

Prove that

Pr fA j Bg C Pr

˚

A j B

D 1 :

C.2-5

Prove that for any collection of events A 1 ; A 2 ; : : : ; A n ,

Pr fA 1 \ A 2 \ \ A n g D Pr fA 1 g Pr fA 2 j A 1 g Pr fA 3 j A 1 \ A 2 g

Pr fA n j A 1 \ A 2 \ \ A n1 g :

C.2-6 ?

Describe a procedure that takes as input two integers a and b such that 0 < a < b

and, using fair coin ﬂips, produces as output heads with probability a=b and tails

with probability .b a/=b. Give a bound on the expected number of coin ﬂips,

which should be O.1/. (Hint: Represent a=b in binary.)

C.2-7 ?

Show how to construct a set of n events that are pairwise independent but such that

no subset of k > 2 of them is mutually independent.

C.2-8 ?

Two events A and B are conditionally independent, given C , if

Pr fA \ B j C g D Pr fA j C g Pr fB j C g :

Give a simple but nontrivial example of two events that are not independent but are

conditionally independent given a third event.

C.2-9 ?

You are a contestant in a game show in which a prize is hidden behind one of

three curtains. You will win the prize if you select the correct curtain. After you

1196 Appendix C Counting and Probability

have picked one curtain but before the curtain is lifted, the emcee lifts one of the

other curtains, knowing that it will reveal an empty stage, and asks if you would

like to switch from your current selection to the remaining curtain. How would

your chances change if you switch? (This question is the celebrated Monty Hall

problem, named after a game-show host who often presented contestants with just

this dilemma.)

C.2-10 ?

A prison warden has randomly picked one prisoner among three to go free. The

other two will be executed. The guard knows which one will go free but is forbid-

den to give any prisoner information regarding his status. Let us call the prisoners

X, Y , and Z. Prisoner X asks the guard privately which of Y or Z will be exe-

cuted, arguing that since he already knows that at least one of them must die, the

guard won’t be revealing any information about his own status. The guard tells X

that Y is to be executed. Prisoner X feels happier now, since he ﬁgures that either

he or prisoner Z will go free, which means that his probability of going free is

now 1=2. Is he right, or are his chances still 1=3? Explain.

C.3 Discrete random variables

A (discrete) random variable X is a function from a ﬁnite or countably inﬁnite

sample space S to the real numbers. It associates a real number with each possible

outcome of an experiment, which allows us to work with the probability distribu-

tion induced on the resulting set of numbers. Random variables can also be deﬁned

for uncountably inﬁnite sample spaces, but they raise technical issues that are un-

necessary to address for our purposes. Henceforth, we shall assume that random

variables are discrete.

For a random variable X and a real number x, we deﬁne the event X D x to be

fs 2 S W X.s/ D xg; thus,

Pr fX D xg D

X

s2SWX.s/Dx

Pr fsg :

The function

f .x/ D Pr fX D xg

is the probability density function of the random variable X. From the probability

axioms, Pr fX D xg 0 and

P

x

Pr fX D xg D 1.

As an example, consider the experiment of rolling a pair of ordinary, 6-sided

dice. There are 36 possible elementary events in the sample space. We assume

C.3 Discrete random variables 1197

that the probability distribution is uniform, so that each elementary event s 2 S is

equally likely: Pr fsg D 1=36. Deﬁne the random variable X to be the maximum of

the two values showing on the dice. We have Pr fX D 3g D 5=36, since X assigns

a value of 3 to 5 of the 36 possible elementary events, namely, .1; 3/, .2; 3/, .3; 3/,

.3; 2/, and .3; 1/.

We often deﬁne several random variables on the same sample space. If X and Y

are random variables, the function

f .x; y/ D Pr fX D x and Y D yg

is the joint probability density function of X and Y . For a ﬁxed value y,

Pr fY D yg D

X

x

Pr fX D x and Y D yg ;

and similarly, for a ﬁxed value x,

Pr fX D xg D

X

y

Pr fX D x and Y D yg :

Using the deﬁnition (C.14) of conditional probability, we have

Pr fX D x j Y D yg D

Pr fX D x and Y D yg

Pr fY D yg

:

We deﬁne two random variables X and Y to be independent if for all x and y, the

events X D x and Y D y are independent or, equivalently, if for all x and y, we

have Pr fX D x and Y D yg D Pr fX D xg Pr fY D yg.

Given a set of random variables deﬁned over the same sample space, we can

deﬁne new random variables as sums, products, or other functions of the original

variables.

Expected value of a random variable

The simplest and most useful summary of the distribution of a random variable is

the “average” of the values it takes on. The expected value (or, synonymously,

expectation or mean) of a discrete random variable X is

E ŒX D

X

x

x Pr fX D xg ; (C.20)

which is well deﬁned if the sum is ﬁnite or converges absolutely. Sometimes the

expectation of X is denoted by X or, when the random variable is apparent from

context, simply by .

Consider a game in which you ﬂip two fair coins. You earn $3 for each head but

lose $2 for each tail. The expected value of the random variable X representing

1198 Appendix C Counting and Probability

your earnings is

E ŒX D 6 Pr f2 H’sg C 1 Pr f1 H, 1 Tg 4 Pr f2 T’sg

D 6.1=4/ C 1.1=2/ 4.1=4/

D 1 :

The expectation of the sum of two random variables is the sum of their expecta-

tions, that is,

E ŒX C Y D E ŒX C E ŒY ; (C.21)

whenever E ŒX and E ŒY are deﬁned. We call this property linearity of expecta-

tion, and it holds even if X and Y are not independent. It also extends to ﬁnite and

absolutely convergent summations of expectations. Linearity of expectation is the

key property that enables us to perform probabilistic analyses by using indicator

random variables (see Section 5.2).

If X is any random variable, any function g.x/ deﬁnes a new random vari-

able g.X/. If the expectation of g.X/ is deﬁned, then

E Œg.X/ D

X

x

g.x/ Pr fX D xg :

Letting g.x/ D ax, we have for any constant a,

E ŒaX D aE ŒX : (C.22)

Consequently, expectations are linear: for any two random variables X and Y and

any constant a,

E ŒaX C Y D aE ŒX C E ŒY : (C.23)

When two random variables X and Y are independent and each has a deﬁned

expectation,

E ŒXY D

X

x

X

y

xy Pr fX D x and Y D yg

D

X

x

X

y

xy Pr fX D xg Pr fY D yg

D

X

x

x Pr fX D xg

!

X

y

y Pr fY D yg

!

D E ŒX E ŒY :

In general, when n random variables X 1 ; X 2 ; : : : ; X n are mutually independent,

E ŒX 1 X 2 X n D E ŒX 1 E ŒX 2 E ŒX n : (C.24)

C.3 Discrete random variables 1199

When a random variable X takes on values from the set of natural numbers

N D f0; 1; 2; : : :g, we have a nice formula for its expectation:

E ŒX D

1 X

iD0

i Pr fX D ig

D

1 X

iD0

i.Pr fX ig Pr fX i C 1g/

D

1 X

iD1

Pr fX ig ; (C.25)

since each term Pr fX ig is added in i times and subtracted out i 1 times

(except Pr fX 0g, which is added in 0 times and not subtracted out at all).

When we apply a convex function f .x/ to a random variable X, Jensen’s in-

equality gives us

E Œf .X/ f .E ŒX/ ; (C.26)

provided that the expectations exist and are ﬁnite. (A function f .x/ is convex

if for all x and y and for all 0 1, we have f .x C .1 /y/

f .x/ C .1 /f .y/.)

Variance and standard deviation

The expected value of a random variable does not tell us how “spread out” the

variable’s values are. For example, if we have random variables X and Y for which

Pr fX D 1=4g D Pr fX D 3=4g D 1=2 and Pr fY D 0g D Pr fY D 1g D 1=2,

then both E ŒX and E ŒY are 1=2, yet the actual values taken on by Y are farther

from the mean than the actual values taken on by X.

The notion of variance mathematically expresses how far from the mean a ran-

dom variable’s values are likely to be. The variance of a random variable X with

mean E ŒX is

Var ŒX D E

.X E ŒX/

2

D E

X

2

2XE ŒX C E

2

ŒX

D E

X

2

2E ŒXE ŒX C E

2

ŒX

D E

X

2

2E

2

ŒX C E

2

ŒX

D E

X

2

E

2

ŒX : (C.27)

To justify the equality E ŒE

2 ŒX D E

2 ŒX, note that because E ŒX is a real num-

ber and not a random variable, so is E

2 ŒX. The equality E ŒXE ŒX D E

2 ŒX

1200 Appendix C Counting and Probability

follows from equation (C.22), with a D E ŒX. Rewriting equation (C.27) yields

an expression for the expectation of the square of a random variable:

E

X

2

D Var ŒX C E

2

ŒX : (C.28)

The variance of a random variable X and the variance of aX are related (see

Exercise C.3-10):

Var ŒaX D a

2

Var ŒX :

When X and Y are independent random variables,

Var ŒX C Y D Var ŒX C Var ŒY :

In general, if n random variables X 1 ; X 2 ; : : : ; X n are pairwise independent, then

Var

"

n X

iD1

X i

#

D

n X

iD1

Var ŒX i : (C.29)

The standard deviation of a random variable X is the nonnegative square root

of the variance of X. The standard deviation of a random variable X is sometimes

denoted X or simply when the random variable X is understood from context.

With this notation, the variance of X is denoted 2

.

Exercises

C.3-1

Suppose we roll two ordinary, 6-sided dice. What is the expectation of the sum

of the two values showing? What is the expectation of the maximum of the two

values showing?

C.3-2

An array AŒ1 : : n contains n distinct numbers that are randomly ordered, with each

permutation of the n numbers being equally likely. What is the expectation of the

index of the maximum element in the array? What is the expectation of the index

of the minimum element in the array?

C.3-3

A carnival game consists of three dice in a cage. A player can bet a dollar on any

of the numbers 1 through 6. The cage is shaken, and the payoff is as follows. If the

player’s number doesn’t appear on any of the dice, he loses his dollar. Otherwise,

if his number appears on exactly k of the three dice, for k D 1; 2; 3, he keeps his

dollar and wins k more dollars. What is his expected gain from playing the carnival

game once?

C.4 The geometric and binomial distributions 1201

C.3-4

Argue that if X and Y are nonnegative random variables, then

E Œmax.X; Y / E ŒX C E ŒY :

C.3-5 ?

Let X and Y be independent random variables. Prove that f .X/ and g.Y / are

independent for any choice of functions f and g.

C.3-6 ?

Let X be a nonnegative random variable, and suppose that E ŒX is well deﬁned.

Prove Markov’s inequality:

Pr fX tg E ŒX =t (C.30)

for all t > 0.

C.3-7 ?

Let S be a sample space, and let X and X 0

be random variables such that

X.s/ X 0 .s/ for all s 2 S. Prove that for any real constant t,

Pr fX tg Pr fX

0

tg :

C.3-8

Which is larger: the expectation of the square of a random variable, or the square

of its expectation?

C.3-9

Show that for any random variable X that takes on only the values 0 and 1, we have

Var ŒX D E ŒX E Œ1 X.

C.3-10

Prove that Var ŒaX D a 2

Var ŒX from the deﬁnition (C.27) of variance.

C.4 The geometric and binomial distributions

We can think of a coin ﬂip as an instance of a Bernoulli trial, which is an experi-

ment with only two possible outcomes: success, which occurs with probability p,

and failure, which occurs with probability q D 1p. When we speak of Bernoulli

trials collectively, we mean that the trials are mutually independent and, unless we

speciﬁcally say otherwise, that each has the same probability p for success. Two

1202 Appendix C Counting and Probability

0.05

0.10

0.15

0.20

0.25

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

0.30

0.35

k

2

3

k1

1

3

Figure C.1 A geometric distribution with probability p D 1=3 of success and a probability

q D 1 p of failure. The expectation of the distribution is 1=p D 3.

important distributions arise from Bernoulli trials: the geometric distribution and

the binomial distribution.

The geometric distribution

Suppose we have a sequence of Bernoulli trials, each with a probability p of suc-

cess and a probability q D 1p of failure. How many trials occur before we obtain

a success? Let us deﬁne the random variable X be the number of trials needed to

obtain a success. Then X has values in the range f1; 2; : : :g, and for k 1,

Pr fX D kg D q

k1

p ; (C.31)

since we have k 1 failures before the one success. A probability distribution sat-

isfying equation (C.31) is said to be a geometric distribution. Figure C.1 illustrates

such a distribution.

C.4 The geometric and binomial distributions 1203

Assuming that q < 1, we can calculate the expectation of a geometric distribu-

tion using identity (A.8):

E ŒX D

1 X

kD1

kq

k1

p

D

p

q

1 X

kD0

kq

k

D

p

q

q

.1 q/ 2

D

p

q

q

p 2

D 1=p : (C.32)

Thus, on average, it takes 1=p trials before we obtain a success, an intuitive result.

The variance, which can be calculated similarly, but using Exercise A.1-3, is

Var ŒX D q=p

2

: (C.33)

As an example, suppose we repeatedly roll two dice until we obtain either a

seven or an eleven. Of the 36 possible outcomes, 6 yield a seven and 2 yield an

eleven. Thus, the probability of success is p D 8=36 D 2=9, and we must roll

1=p D 9=2 D 4:5 times on average to obtain a seven or eleven.

The binomial distribution

How many successes occur during n Bernoulli trials, where a success occurs with

probability p and a failure with probability q D 1 p? Deﬁne the random vari-

able X to be the number of successes in n trials. Then X has values in the range

f0; 1; : : : ; ng, and for k D 0; 1; : : : ; n,

Pr fX D kg D

n

k

!

p

k

q

nk

; (C.34)

since there are

n

k

ways to pick which k of the n trials are successes, and the

probability that each occurs is p k q nk

. A probability distribution satisfying equa-

tion (C.34) is said to be a binomial distribution. For convenience, we deﬁne the

family of binomial distributions using the notation

b.kI n; p/ D

n

k

!

p

k

.1 p/

nk

: (C.35)

Figure C.2 illustrates a binomial distribution. The name “binomial” comes from the

right-hand side of equation (C.34) being the kth term of the expansion of .p Cq/ n

.

Consequently, since p C q D 1,

1204 Appendix C Counting and Probability

0.05

0.10

0.15

0.20

0.25

k

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

b (k; 15, 1/3)

Figure C.2 The binomial distribution b.kI 15; 1=3/ resulting from n D 15 Bernoulli trials, each

with probability p D 1=3 of success. The expectation of the distribution is np D 5.

n X

kD0

b.kI n; p/ D 1 ; (C.36)

as axiom 2 of the probability axioms requires.

We can compute the expectation of a random variable having a binomial distri-

bution from equations (C.8) and (C.36). Let X be a random variable that follows

the binomial distribution b.kI n; p/, and let q D 1 p. By the deﬁnition of expec-

tation, we have

E ŒX D

n X

kD0

k Pr fX D kg

D

n X

kD0

k b.kI n; p/

D

n X

kD1

k

n

k

!

p

k

q

nk

D np

n X

kD1

n 1

k 1

!

p

k1

q

nk

(by equation (C.8))

D np

n1 X

kD0

n 1

k

!

p

k

q

.n1/k

C.4 The geometric and binomial distributions 1205

D np

n1 X

kD0

b.kI n 1; p/

D np (by equation (C.36)) . (C.37)

By using the linearity of expectation, we can obtain the same result with sub-

stantially less algebra. Let X i be the random variable describing the number of

successes in the ith trial. Then E ŒX i D p 1 C q 0 D p, and by linearity of

expectation (equation (C.21)), the expected number of successes for n trials is

E ŒX D E

"

n X

iD1

X i

#

D

n X

iD1

E ŒX i

D

n X

iD1

p

D np : (C.38)

We can use the same approach to calculate the variance of the distribution. Using

equation (C.27), we have Var ŒX i D E ŒX 2

i

E

2 ŒX i . Since X i only takes on the

values 0 and 1, we have X 2

i

D X i , which implies E ŒX 2

i

D E ŒX i D p. Hence,

Var ŒX i D p p

2

D p.1 p/ D pq : (C.39)

To compute the variance of X, we take advantage of the independence of the n

trials; thus, by equation (C.29),

Var ŒX D Var

"

n X

iD1

X i

#

D

n X

iD1

Var ŒX i

D

n X

iD1

pq

D npq : (C.40)

As Figure C.2 shows, the binomial distribution b.kI n; p/ increases with k until

it reaches the mean np, and then it decreases. We can prove that the distribution

always behaves in this manner by looking at the ratio of successive terms:

1206 Appendix C Counting and Probability

b.kI n; p/

b.k 1I n; p/

D

n

k

p k q nk

n

k1

p k1 q nkC1

D

nŠ.k 1/Š.n k C 1/Šp

kŠ.n k/ŠnŠq

D

.n k C 1/p

kq

(C.41)

D 1 C

.n C 1/p k

kq

:

This ratio is greater than 1 precisely when .n C 1/p k is positive. Conse-

quently, b.kI n; p/ > b.k 1I n; p/ for k < .n C 1/p (the distribution increases),

and b.kI n; p/ < b.k 1I n; p/ for k > .n C 1/p (the distribution decreases).

If k D .n C 1/p is an integer, then b.kI n; p/ D b.k 1I n; p/, and so the distri-

bution then has two maxima: at k D .nC1/p and at k1 D .nC1/p1 D np q.

Otherwise, it attains a maximum at the unique integer k that lies in the range

np q < k < .n C 1/p.

The following lemma provides an upper bound on the binomial distribution.

Lemma C.1

Let n 0, let 0 < p < 1, let q D 1 p, and let 0 k n. Then

b.kI n; p/

np

k

k

nq

n k

nk

:

Proof Using equation (C.6), we have

b.kI n; p/ D

n

k

!

p

k

q

nk

n

k

k

n

n k

nk

p

k

q

nk

D

np

k

k

nq

n k

nk

:

Exercises

C.4-1

Verify axiom 2 of the probability axioms for the geometric distribution.

C.4-2

How many times on average must we ﬂip 6 fair coins before we obtain 3 heads

and 3 tails?

C.4 The geometric and binomial distributions 1207

C.4-3

Show that b.kI n; p/ D b.n kI n; q/, where q D 1 p.

C.4-4

Show that value of the maximum of the binomial distribution b.kI n; p/ is approx-

imately 1=

p

2npq, where q D 1 p.

C.4-5 ?

Show that the probability of no successes in n Bernoulli trials, each with probability

p D 1=n, is approximately 1=e. Show that the probability of exactly one success

is also approximately 1=e.

C.4-6 ?

Professor Rosencrantz ﬂips a fair coin n times, and so does Professor Guildenstern.

Show that the probability that they get the same number of heads is

2n

n

=4 n

. (Hint:

For Professor Rosencrantz, call a head a success; for Professor Guildenstern, call

a tail a success.) Use your argument to verify the identity

n X

kD0

n

k

! 2

D

2n

n

!

:

C.4-7 ?

Show that for 0 k n,

b.kI n; 1=2/ 2

n H.k=n/n

;

where H.x/ is the entropy function (C.7).

C.4-8 ?

Consider n Bernoulli trials, where for i D 1; 2; : : : ; n, the ith trial has probabil-

ity p i of success, and let X be the random variable denoting the total number of

successes. Let p p i for all i D 1; 2; : : : ; n. Prove that for 1 k n,

Pr fX < kg

k1 X

iD0

b.iI n; p/ :

C.4-9 ?

Let X be the random variable for the total number of successes in a set A of n

Bernoulli trials, where the ith trial has a probability p i of success, and let X 0

be the random variable for the total number of successes in a second set A 0

of n

Bernoulli trials, where the ith trial has a probability p 0

i

p i of success. Prove that

for 0 k n,

1208 Appendix C Counting and Probability

Pr fX

0

kg Pr fX kg :

(Hint: Show how to obtain the Bernoulli trials in A 0

by an experiment involving

the trials of A, and use the result of Exercise C.3-7.)

? C.5 The tails of the binomial distribution

The probability of having at least, or at most, k successes in n Bernoulli trials,

each with probability p of success, is often of more interest than the probability of

having exactly k successes. In this section, we investigate the tails of the binomial

distribution: the two regions of the distribution b.kI n; p/ that are far from the

mean np. We shall prove several important bounds on (the sum of all terms in) a

tail.

We ﬁrst provide a bound on the right tail of the distribution b.kI n; p/. We can

determine bounds on the left tail by inverting the roles of successes and failures.

Theorem C.2

Consider a sequence of n Bernoulli trials, where success occurs with probability p.

Let X be the random variable denoting the total number of successes. Then for

0 k n, the probability of at least k successes is

Pr fX kg D

n X

iDk

b.iI n; p/

n

k

!

p

k

:

Proof For S f1; 2; : : : ; ng, we let A S denote the event that the ith trial is a

success for every i 2 S. Clearly Pr fA S g D p k

if jSj D k. We have

Pr fX kg D Pr fthere exists S f1; 2; : : : ; ng W jSj D k and A S g

D Pr

[

Sf1;2;:::;ngWjSjDk

A S

X

Sf1;2;:::;ngWjSjDk

Pr fA S g (by inequality (C.19))

D

n

k

!

p

k

:

C.5 The tails of the binomial distribution 1209

The following corollary restates the theorem for the left tail of the binomial

distribution. In general, we shall leave it to you to adapt the proofs from one tail to

the other.

Corollary C.3

Consider a sequence of n Bernoulli trials, where success occurs with probabil-

ity p. If X is the random variable denoting the total number of successes, then for

0 k n, the probability of at most k successes is

Pr fX kg D

k X

iD0

b.iI n; p/

n

n k

!

.1 p/

nk

D

n

k

!

.1 p/

nk

:

Our next bound concerns the left tail of the binomial distribution. Its corollary

shows that, far from the mean, the left tail diminishes exponentially.

Theorem C.4

Consider a sequence of n Bernoulli trials, where success occurs with probability p

and failure with probability q D 1 p. Let X be the random variable denoting the

total number of successes. Then for 0 < k < np, the probability of fewer than k

successes is

Pr fX < kg D

k1 X

iD0

b.iI n; p/

<

kq

np k

b.kI n; p/ :

Proof We bound the series

P k1

iD0

b.iI n; p/ by a geometric series using the tech-

nique from Section A.2, page 1151. For i D 1; 2; : : : ; k, we have from equa-

tion (C.41),

b.i 1I n; p/

b.iI n; p/

D

iq

.n i C 1/p

<

iq

.n i/p

kq

.n k/p

:

1210 Appendix C Counting and Probability

If we let

x D

kq

.n k/p

<

kq

.n np/p

D

kq

nqp

D

k

np

< 1 ;

it follows that

b.i 1I n; p/ < x b.iI n; p/

for 0 < i k. Iteratively applying this inequality k i times, we obtain

b.iI n; p/ < x

ki

b.kI n; p/

for 0 i < k, and hence

k1 X

iD0

b.iI n; p/ <

k1 X

iD0

x

ki

b.kI n; p/

< b.kI n; p/

1 X

iD0

x

i

D

x

1 x

b.kI n; p/

D

kq

np k

b.kI n; p/ :

Corollary C.5

Consider a sequence of n Bernoulli trials, where success occurs with probability p

and failure with probability q D 1 p. Then for 0 < k np=2, the probability of

fewer than k successes is less than one half of the probability of fewer than k C 1

successes.

Proof Because k np=2, we have

kq

np k

.np=2/q

np .np=2/

C.5 The tails of the binomial distribution 1211

D

.np=2/q

np=2

1 ; (C.42)

since q 1. Letting X be the random variable denoting the number of successes,

Theorem C.4 and inequality (C.42) imply that the probability of fewer than k suc-

cesses is

Pr fX < kg D

k1 X

iD0

b.iI n; p/ < b.kI n; p/ :

Thus we have

Pr fX < kg

Pr fX < k C 1g

D

P k1

iD0

b.iI n; p/

P k

iD0

b.iI n; p/

D

P k1

iD0

b.iI n; p/

P k1

iD0

b.iI n; p/ C b.kI n; p/

< 1=2 ;

since

P k1

iD0

b.iI n; p/ < b.kI n; p/.

Bounds on the right tail follow similarly. Exercise C.5-2 asks you to prove them.

Corollary C.6

Consider a sequence of n Bernoulli trials, where success occurs with probability p.

Let X be the random variable denoting the total number of successes. Then for

np < k < n, the probability of more than k successes is

Pr fX > kg D

n X

iDkC1

b.iI n; p/

<

.n k/p

k np

b.kI n; p/ :

Corollary C.7

Consider a sequence of n Bernoulli trials, where success occurs with probability p

and failure with probability q D 1 p. Then for .np C n/=2 < k < n, the

probability of more than k successes is less than one half of the probability of

more than k 1 successes.

The next theorem considers n Bernoulli trials, each with a probability p i of

success, for i D 1; 2; : : : ; n. As the subsequent corollary shows, we can use the

1212 Appendix C Counting and Probability

theorem to provide a bound on the right tail of the binomial distribution by setting

p i D p for each trial.

Theorem C.8

Consider a sequence of n Bernoulli trials, where in the ith trial, for i D 1; 2; : : : ; n,

success occurs with probability p i and failure occurs with probability q i D 1 p i .

Let X be the random variable describing the total number of successes, and let

D E ŒX. Then for r > ,

Pr fX rg

e

r

r

:

Proof Since for any ˛ > 0, the function e ˛x

is strictly increasing in x,

Pr fX rg D Pr

˚

e

˛.X

/

e

˛r

; (C.43)

where we will determine ˛ later. Using Markov’s inequality (C.30), we obtain

Pr

˚

e

˛.X

/

e

˛r

E

e

˛.X

/

e

˛r

: (C.44)

The bulk of the proof consists of bounding E

e ˛.X

/

and substituting a suit-

able value for ˛ in inequality (C.44). First, we evaluate E

e ˛.X

/

. Using the

technique of indicator random variables (see Section 5.2), let X i D I fthe ith

Bernoulli trial is a successg for i D 1; 2; : : : ; n; that is, X i is the random vari-

able that is 1 if the ith Bernoulli trial is a success and 0 if it is a failure. Thus,

X D

n X

iD1

X i ;

and by linearity of expectation,

D E ŒX D E

"

n X

iD1

X i

#

D

n X

iD1

E ŒX i D

n X

iD1

p i ;

which implies

X D

n X

iD1

.X i p i / :

To evaluate E

e ˛.X

/

, we substitute for X , obtaining

E

e

˛.X

/

D E

e

˛

P n

iD1 .X i p i /

D E

"

n Y

iD1

e

˛.X i p i /

#

D

n Y

iD1

E

e

˛.X i p i /

;

C.5 The tails of the binomial distribution 1213

which follows from (C.24), since the mutual independence of the random vari-

ables X i implies the mutual independence of the random variables e ˛.X i p i /

(see

Exercise C.3-5). By the deﬁnition of expectation,

E

e

˛.X i p i /

D e

˛.1p i /

p i C e

˛.0p i /

q i

D p i e

˛q i C q i e

˛p i

p i e

˛

C 1 (C.45)

exp.p i e

˛

/ ;

where exp.x/ denotes the exponential function: exp.x/ D e x

. (Inequality (C.45)

follows from the inequalities ˛ > 0, q i 1, e ˛q i e ˛

, and e ˛p i 1, and the last

line follows from inequality (3.12).) Consequently,

E

e

˛.X

/

D

n Y

iD1

E

e

˛.X i p i /

n Y

iD1

exp.p i e

˛

/

D exp

n X

iD1

p i e

˛

!

D exp.e

˛

/ ; (C.46)

since D

P n

iD1

p i . Therefore, from equation (C.43) and inequalities (C.44)

and (C.46), it follows that

Pr fX rg exp.e

˛

˛r/ : (C.47)

Choosing ˛ D ln.r=/ (see Exercise C.5-7), we obtain

Pr fX rg exp.e ln.r=

/

r ln.r=//

D exp.r r ln.r=//

D

e r

.r=/ r

D

e

r

r

:

When applied to Bernoulli trials in which each trial has the same probability of

success, Theorem C.8 yields the following corollary bounding the right tail of a

binomial distribution.

1214 Appendix C Counting and Probability

Corollary C.9

Consider a sequence of n Bernoulli trials, where in each trial success occurs with

probability p and failure occurs with probability q D 1 p. Then for r > np,

Pr fX np rg D

n X

kDdnpCre

b.kI n; p/

npe

r

r

:

Proof By equation (C.37), we have D E ŒX D np.

Exercises

C.5-1 ?

Which is less likely: obtaining no heads when you ﬂip a fair coin n times, or

obtaining fewer than n heads when you ﬂip the coin 4n times?

C.5-2 ?

Prove Corollaries C.6 and C.7.

C.5-3 ?

Show that

k1 X

iD0

n

i

!

a

i

< .a C 1/

n

k

na k.a C 1/

b.kI n; a=.a C 1//

for all a > 0 and all k such that 0 < k < na=.a C 1/.

C.5-4 ?

Prove that if 0 < k < np, where 0 < p < 1 and q D 1 p, then

k1 X

iD0

p

i

q

ni

<

kq

np k

np

k

k

nq

n k

nk

:

C.5-5 ?

Show that the conditions of Theorem C.8 imply that

Pr f X rg

.n /e

r

r

:

Similarly, show that the conditions of Corollary C.9 imply that

Pr fnp X rg

nqe

r

r

:

Problems for Appendix C 1215

C.5-6 ?

Consider a sequence of n Bernoulli trials, where in the ith trial, for i D 1; 2; : : : ; n,

success occurs with probability p i and failure occurs with probability q i D 1 p i .

Let X be the random variable describing the total number of successes, and let

D E ŒX. Show that for r 0,

Pr fX rg e

r

2

=2n

:

(Hint: Prove that p i e ˛q i C q i e ˛p i e ˛

2

=2

. Then follow the outline of the proof

of Theorem C.8, using this inequality in place of inequality (C.45).)

C.5-7 ?

Show that choosing ˛ D ln.r=/ minimizes the right-hand side of inequal-

ity (C.47).

Problems

C-1 Balls and bins

In this problem, we investigate the effect of various assumptions on the number of

ways of placing n balls into b distinct bins.

a. Suppose that the n balls are distinct and that their order within a bin does not

matter. Argue that the number of ways of placing the balls in the bins is b n

.

b. Suppose that the balls are distinct and that the balls in each bin are ordered.

Prove that there are exactly .b C n 1/Š=.b 1/Š ways to place the balls in the

bins. (Hint: Consider the number of ways of arranging n distinct balls and b 1

indistinguishable sticks in a row.)

c. Suppose that the balls are identical, and hence their order within a bin does not

matter. Show that the number of ways of placing the balls in the bins is

bCn1

n

.

(Hint: Of the arrangements in part (b), how many are repeated if the balls are

made identical?)

d. Suppose that the balls are identical and that no bin may contain more than one

ball, so that n b. Show that the number of ways of placing the balls is

b

n

.

e. Suppose that the balls are identical and that no bin may be left empty. Assuming

that n b, show that the number of ways of placing the balls is

n1

b1

.

1216 Appendix C Counting and Probability

Appendix notes

The ﬁrst general methods for solving probability problems were discussed in a

famous correspondence between B. Pascal and P. de Fermat, which began in 1654,

and in a book by C. Huygens in 1657. Rigorous probability theory began with the

work of J. Bernoulli in 1713 and A. De Moivre in 1730. Further developments of

the theory were provided by P.-S. Laplace, S.-D. Poisson, and C. F. Gauss.

Sums of random variables were originally studied by P. L. Chebyshev and A. A.

Markov. A. N. Kolmogorov axiomatized probability theory in 1933. Chernoff [66]

and Hoeffding [173] provided bounds on the tails of distributions. Seminal work

in random combinatorial structures was done by P. Erd¨os.

Knuth [209] and Liu [237] are good references for elementary combinatorics

and counting. Standard textbooks such as Billingsley [46], Chung [67], Drake [95],

Feller [104], and Rozanov [300] offer comprehensive introductions to probability.

D Matrices

Matrices arise in numerous applications, including, but by no means limited to,

scientiﬁc computing. If you have seen matrices before, much of the material in this

appendix will be familiar to you, but some of it might be new. Section D.1 covers

basic matrix deﬁnitions and operations, and Section D.2 presents some basic matrix

properties.

D.1 Matrices and matrix operations

In this section, we review some basic concepts of matrix theory and some funda-

mental properties of matrices.

Matrices and vectors

A matrix is a rectangular array of numbers. For example,

A D

a 11 a 12 a 13

a 21 a 22 a 23

D

1 2 3

4 5 6

(D.1)

is a 2 3 matrix A D .a ij /, where for i D 1; 2 and j D 1; 2; 3, we denote the

element of the matrix in row i and column j by a ij . We use uppercase letters

to denote matrices and corresponding subscripted lowercase letters to denote their

elements. We denote the set of all m n matrices with real-valued entries by R mn

and, in general, the set of m n matrices with entries drawn from a set S by S mn

.

The transpose of a matrix A is the matrix A T obtained by exchanging the rows

and columns of A. For the matrix A of equation (D.1),

1218 Appendix D Matrices

A T D

1 4

2 5

3 6

:

A vector is a one-dimensional array of numbers. For example,

x D

2

3

5

is a vector of size 3. We sometimes call a vector of length n an n-vector. We

use lowercase letters to denote vectors, and we denote the ith element of a size-n

vector x by x i , for i D 1; 2; : : : ; n. We take the standard form of a vector to be

as a column vector equivalent to an n 1 matrix; the corresponding row vector is

obtained by taking the transpose:

x T D . 2 3 5 / :

The unit vector e i is the vector whose ith element is 1 and all of whose other

elements are 0. Usually, the size of a unit vector is clear from the context.

A zero matrix is a matrix all of whose entries are 0. Such a matrix is often

denoted 0, since the ambiguity between the number 0 and a matrix of 0s is usually

easily resolved from context. If a matrix of 0s is intended, then the size of the

matrix also needs to be derived from the context.

Square matrices

Square n n matrices arise frequently. Several special cases of square matrices

are of particular interest:

1. A diagonal matrix has a ij D 0 whenever i ¤ j . Because all of the off-diagonal

elements are zero, we can specify the matrix by listing the elements along the

diagonal:

diag.a 11 ; a 22 ; : : : ; a nn / D

˙

a 11 0 : : : 0

0 a 22 : : : 0

:

:

:

:

:

:

:

:

:

:

:

:

0 0 : : : a nn

:

2. The n n identity matrix I n is a diagonal matrix with 1s along the diagonal:

I n D diag.1; 1; : : : ; 1/

D

˙

1 0 : : : 0

0 1 : : : 0

:

:

:

:

:

:

:

:

:

:

:

:

0 0 : : : 1

:

D.1 Matrices and matrix operations 1219

When I appears without a subscript, we derive its size from the context. The ith

column of an identity matrix is the unit vector e i .

3. A tridiagonal matrix T is one for which t ij D 0 if ji j j > 1. Nonzero entries

appear only on the main diagonal, immediately above the main diagonal (t i;iC1

for i D 1; 2; : : : ; n 1), or immediately below the main diagonal (t iC1;i for

i D 1; 2; : : : ; n 1):

T D

t 11 t 12 0 0 : : : 0 0 0

t 21 t 22 t 23 0 : : : 0 0 0

0 t 32 t 33 t 34 : : : 0 0 0

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

0 0 0 0 : : : t n2;n2 t n2;n1 0

0 0 0 0 : : : t n1;n2 t n1;n1 t n1;n

0 0 0 0 : : : 0 t n;n1 t nn

˘

:

4. An upper-triangular matrix U is one for which u ij D 0 if i > j . All entries

below the diagonal are zero:

U D

˙

u 11 u 12 : : : u 1n

0 u 22 : : : u 2n

:

:

:

:

:

:

:

:

:

:

:

:

0 0 : : : u nn

:

An upper-triangular matrix is unit upper-triangular if it has all 1s along the

diagonal.

5. A lower-triangular matrix L is one for which l ij D 0 if i < j . All entries

above the diagonal are zero:

L D

˙

l 11 0 : : : 0

l 21 l 22 : : : 0

:

:

:

:

:

:

:

:

:

:

:

:

l n1 l n2 : : : l nn

:

A lower-triangular matrix is unit lower-triangular if it has all 1s along the

diagonal.

1220 Appendix D Matrices

6. A permutation matrix P has exactly one 1 in each row or column, and 0s

elsewhere. An example of a permutation matrix is

P D

ˇ

0 1 0 0 0

0 0 0 1 0

1 0 0 0 0

0 0 0 0 1

0 0 1 0 0

:

Such a matrix is called a permutation matrix because multiplying a vector x

by a permutation matrix has the effect of permuting (rearranging) the elements

of x. Exercise D.1-4 explores additional properties of permutation matrices.

7. A symmetric matrix A satisﬁes the condition A D A T . For example,

1 2 3

2 6 4

3 4 5

is a symmetric matrix.

Basic matrix operations

The elements of a matrix or vector are numbers from a number system, such as

the real numbers, the complex numbers, or integers modulo a prime. The number

system deﬁnes how to add and multiply numbers. We can extend these deﬁnitions

to encompass addition and multiplication of matrices.

We deﬁne matrix addition as follows. If A D .a ij / and B D .b ij / are m n

matrices, then their matrix sum C D .c ij / D A C B is the m n matrix deﬁned by

c ij D a ij C b ij

for i D 1; 2; : : : ; m and j D 1; 2; : : : ; n. That is, matrix addition is performed

componentwise. A zero matrix is the identity for matrix addition:

A C 0 D A D 0 C A :

If is a number and A D .a ij / is a matrix, then A D .a ij / is the scalar

multiple of A obtained by multiplying each of its elements by . As a special case,

we deﬁne the negative of a matrix A D .a ij / to be 1 A D A, so that the ij th

entry of A is a ij . Thus,

A C .A/ D 0 D .A/ C A :

D.1 Matrices and matrix operations 1221

We use the negative of a matrix to deﬁne matrix subtraction: A B D A C .B/.

We deﬁne matrix multiplication as follows. We start with two matrices A and B

that are compatible in the sense that the number of columns of A equals the number

of rows of B. (In general, an expression containing a matrix product AB is always

assumed to imply that matrices A and B are compatible.) If A D .a ik / is an m n

matrix and B D .b kj / is an n p matrix, then their matrix product C D AB is the

m p matrix C D .c ij /, where

c ij D

n X

kD1

a ik b kj (D.2)

for i D 1; 2; : : : ; m and j D 1; 2; : : : ; p. The procedure SQUARE-MATRIX-

MULTIPLY in Section 4.2 implements matrix multiplication in the straightfor-

ward manner based on equation (D.2), assuming that the matrices are square:

m D n D p. To multiply n n matrices, SQUARE-MATRIX-MULTIPLY per-

forms n 3

multiplications and n 2 .n 1/ additions, and so its running time is ‚.n 3 /.

Matrices have many (but not all) of the algebraic properties typical of numbers.

Identity matrices are identities for matrix multiplication:

I m A D AI n D A

for any m n matrix A. Multiplying by a zero matrix gives a zero matrix:

A 0 D 0 :

Matrix multiplication is associative:

A.BC / D .AB/C

for compatible matrices A, B, and C . Matrix multiplication distributes over addi-

tion:

A.B C C / D AB C AC ;

.B C C /D D BD C CD :

For n > 1, multiplication of n n matrices is not commutative. For example, if

A D

0 1

0 0

and B D

0 0

1 0

, then

AB D

1 0

0 0

and

BA D

0 0

0 1

:

1222 Appendix D Matrices

We deﬁne matrix-vector products or vector-vector products as if the vector were

the equivalent n 1 matrix (or a 1 n matrix, in the case of a row vector). Thus,

if A is an m n matrix and x is an n-vector, then Ax is an m-vector. If x and y

are n-vectors, then

x T y D

n X

iD1

x i y i

is a number (actually a 1 1 matrix) called the inner product of x and y. The ma-

trix xy T is an n n matrix Z called the outer product of x and y, with ´ ij D x i y j .

The (euclidean) norm kxk of an n-vector x is deﬁned by

kxk D .x

2

1

C x

2

2

C C x

2

n

/

1=2

D .x T x/

1=2

:

Thus, the norm of x is its length in n-dimensional euclidean space.

Exercises

D.1-1

Show that if A and B are symmetric n n matrices, then so are A C B and A B.

D.1-2

Prove that .AB/ T D B T A T and that A T A is always a symmetric matrix.

D.1-3

Prove that the product of two lower-triangular matrices is lower-triangular.

D.1-4

Prove that if P is an n n permutation matrix and A is an n n matrix, then the

matrix product PA is A with its rows permuted, and the matrix product AP is A

with its columns permuted. Prove that the product of two permutation matrices is

a permutation matrix.

D.2 Basic matrix properties

In this section, we deﬁne some basic properties pertaining to matrices: inverses,

linear dependence and independence, rank, and determinants. We also deﬁne the

class of positive-deﬁnite matrices.

D.2 Basic matrix properties 1223

Matrix inverses, ranks, and determinants

We deﬁne the inverse of an n n matrix A to be the n n matrix, denoted A 1

(if

it exists), such that AA 1 D I n D A 1 A. For example,

1 1

1 0

1

D

0 1

1 1

:

Many nonzero n n matrices do not have inverses. A matrix without an inverse is

called noninvertible, or singular. An example of a nonzero singular matrix is

1 0

1 0

:

If a matrix has an inverse, it is called invertible, or nonsingular. Matrix inverses,

when they exist, are unique. (See Exercise D.2-1.) If A and B are nonsingular

n n matrices, then

.BA/

1

D A

1

B

1

:

The inverse operation commutes with the transpose operation:

.A

1

/ T D .A T /

1

:

The vectors x 1 ; x 2 ; : : : ; x n are linearly dependent if there exist coefﬁcients

c 1 ; c 2 ; : : : ; c n , not all of which are zero, such that c 1 x 1 C c 2 x 2 C C c n x n D 0.

The row vectors x 1 D . 1 2 3 /, x 2 D . 2 6 4 /, and x 3 D . 4 11 9 / are

linearly dependent, for example, since 2x 1 C 3x 2 2x 3 D 0. If vectors are not

linearly dependent, they are linearly independent. For example, the columns of an

identity matrix are linearly independent.

The column rank of a nonzero m n matrix A is the size of the largest set

of linearly independent columns of A. Similarly, the row rank of A is the size

of the largest set of linearly independent rows of A. A fundamental property of

any matrix A is that its row rank always equals its column rank, so that we can

simply refer to the rank of A. The rank of an m n matrix is an integer between 0

and min.m; n/, inclusive. (The rank of a zero matrix is 0, and the rank of an n n

identity matrix is n.) An alternate, but equivalent and often more useful, deﬁnition

is that the rank of a nonzero m n matrix A is the smallest number r such that

there exist matrices B and C of respective sizes m r and r n such that

A D BC :

A square n n matrix has full rank if its rank is n. An m n matrix has full

column rank if its rank is n. The following theorem gives a fundamental property

of ranks.

1224 Appendix D Matrices

Theorem D.1

A square matrix has full rank if and only if it is nonsingular.

A null vector for a matrix A is a nonzero vector x such that Ax D 0. The

following theorem (whose proof is left as Exercise D.2-7) and its corollary relate

the notions of column rank and singularity to null vectors.

Theorem D.2

A matrix A has full column rank if and only if it does not have a null vector.

Corollary D.3

A square matrix A is singular if and only if it has a null vector.

The ij th minor of an n n matrix A, for n > 1, is the .n1/ .n1/ matrix A Œij

obtained by deleting the ith row and j th column of A. We deﬁne the determinant

of an n n matrix A recursively in terms of its minors by

det.A/ D

‚

a 11 if n D 1 ;

n X

j D1

.1/

1Cj

a 1j det.A Œ1j / if n > 1 :

The term .1/ iCj

det.A Œij / is known as the cofactor of the element a ij .

The following theorems, whose proofs are omitted here, express fundamental

properties of the determinant.

Theorem D.4 (Determinant properties)

The determinant of a square matrix A has the following properties:

If any row or any column of A is zero, then det.A/ D 0.

The determinant of A is multiplied by if the entries of any one row (or any

one column) of A are all multiplied by .

The determinant of A is unchanged if the entries in one row (respectively, col-

umn) are added to those in another row (respectively, column).

The determinant of A equals the determinant of A T .

The determinant of A is multiplied by 1 if any two rows (or any two columns)

are exchanged.

Also, for any square matrices A and B, we have det.AB/ D det.A/ det.B/.

D.2 Basic matrix properties 1225

Theorem D.5

An n n matrix A is singular if and only if det.A/ D 0.

Positive-deﬁnite matrices

Positive-deﬁnite matrices play an important role in many applications. An n n

matrix A is positive-deﬁnite if x T Ax > 0 for all n-vectors x ¤ 0. For

example, the identity matrix is positive-deﬁnite, since for any nonzero vector

x D . x 1 x 2 x n / T ,

x T I n x D x T x

D

n X

iD1

x

2

i

> 0 :

Matrices that arise in applications are often positive-deﬁnite due to the following

theorem.

Theorem D.6

For any matrix A with full column rank, the matrix A T A is positive-deﬁnite.

Proof We must show that x T .A T A/x > 0 for any nonzero vector x. For any

vector x,

x T .A T A/x D .Ax/ T .Ax/ (by Exercise D.1-2)

D kAxk

2

:

Note that kAxk

2

is just the sum of the squares of the elements of the vector Ax.

Therefore, kAxk

2

0. If kAxk

2

D 0, every element of Ax is 0, which is to say

Ax D 0. Since A has full column rank, Ax D 0 implies x D 0, by Theorem D.2.

Hence, A T A is positive-deﬁnite.

Section 28.3 explores other properties of positive-deﬁnite matrices.

Exercises

D.2-1

Prove that matrix inverses are unique, that is, if B and C are inverses of A, then

B D C .

D.2-2

Prove that the determinant of a lower-triangular or upper-triangular matrix is equal

to the product of its diagonal elements. Prove that the inverse of a lower-triangular

matrix, if it exists, is lower-triangular.

1226 Appendix D Matrices

D.2-3

Prove that if P is a permutation matrix, then P is invertible, its inverse is P T ,

and P T is a permutation matrix.

D.2-4

Let A and B be n n matrices such that AB D I. Prove that if A 0

is obtained

from A by adding row j into row i, then subtracting column i from column j of B

yields the inverse B 0

of A 0

.

D.2-5

Let A be a nonsingular n n matrix with complex entries. Show that every entry

of A 1

is real if and only if every entry of A is real.

D.2-6

Show that if A is a nonsingular, symmetric, n n matrix, then A 1

is symmetric.

Show that if B is an arbitrary m n matrix, then the m m matrix given by the

product BAB T is symmetric.

D.2-7

Prove Theorem D.2. That is, show that a matrix A has full column rank if and only

if Ax D 0 implies x D 0. (Hint: Express the linear dependence of one column on

the others as a matrix-vector equation.)

D.2-8

Prove that for any two compatible matrices A and B,

rank.AB/ min.rank.A/; rank.B// ;

where equality holds if either A or B is a nonsingular square matrix. (Hint: Use

the alternate deﬁnition of the rank of a matrix.)

Problems

D-1 Vandermonde matrix

Given numbers x 0 ; x 1 ; : : : ; x n1 , prove that the determinant of the Vandermonde

matrix

V.x 0 ; x 1 ; : : : ; x n1 / D

˙

1 x 0 x 2

0

x n1

0

1 x 1 x 2

1

x n1

1

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

1 x n1 x 2

n1

x n1

n1

Problems for Appendix D 1227

is

det.V.x 0 ; x 1 ; : : : ; x n1 // D

Y

0j <kn1

.x k x j / :

(Hint: Multiply column i by x 0 and add it to column i C 1 for i D n 1;

n 2; : : : ; 1, and then use induction.)

D-2 Permutations deﬁned by matrix-vector multiplication over GF.2/

One class of permutations of the integers in the set S n D f0; 1; 2; : : : ; 2 n 1g is

deﬁned by matrix multiplication over GF.2/. For each integer x in S n , we view its

binary representation as an n-bit vector x 0

x 1

x 2

:

:

:

x n1

;

where x D

P n1

iD0

x i 2 i

. If A is an n n matrix in which each entry is either 0

or 1, then we can deﬁne a permutation mapping each value x in S n to the number

whose binary representation is the matrix-vector product Ax. Here, we perform

all arithmetic over GF.2/: all values are either 0 or 1, and with one exception the

usual rules of addition and multiplication apply. The exception is that 1 C 1 D 0.

You can think of arithmetic over GF.2/ as being just like regular integer arithmetic,

except that you use only the least signiﬁcant bit.

As an example, for S 2 D f0; 1; 2; 3g, the matrix

A D

1 0

1 1

deﬁnes the following permutation A : A .0/ D 0, A .1/ D 3, A .2/ D 2,

A .3/ D 1. To see why A .3/ D 1, observe that, working in GF.2/,

A .3/ D

1 0

1 1

1

1

D

1 1 C 0 1

1 1 C 1 1

D

1

0

;

which is the binary representation of 1.

1228 Appendix D Matrices

For the remainder of this problem, we work over GF.2/, and all matrix and

vector entries are 0 or 1. We deﬁne the rank of a 0-1 matrix (a matrix for which

each entry is either 0 or 1) over GF.2/ the same as for a regular matrix, but with all

arithmetic that determines linear independence performed over GF.2/. We deﬁne

the range of an n n 0-1 matrix A by

R.A/ D fy W y D Ax for some x 2 S n g ;

so that R.A/ is the set of numbers in S n that we can produce by multiplying each

value x in S n by A.

a. If r is the rank of matrix A, prove that jR.A/j D 2 r

. Conclude that A deﬁnes a

permutation on S n only if A has full rank.

For a given n n matrix A and a given value y 2 R.A/, we deﬁne the preimage

of y by

P .A; y/ D fx W Ax D yg ;

so that P .A; y/ is the set of values in S n that map to y when multiplied by A.

b. If r is the rank of n n matrix A and y 2 R.A/, prove that jP .A; y/j D 2 nr

.

Let 0 m n, and suppose we partition the set S n into blocks of consec-

utive numbers, where the ith block consists of the 2 m

numbers i2 m ; i2 m C 1;

i2 m C 2; : : : ; .i C 1/2 m 1. For any subset S S n , deﬁne B.S; m/ to be the

set of size-2 m

blocks of S n containing some element of S. As an example, when

n D 3, m D 1, and S D f1; 4; 5g, then B.S; m/ consists of blocks 0 (since 1 is in

the 0th block) and 2 (since both 4 and 5 are in block 2).

c. Let r be the rank of the lower left .n m/ m submatrix of A, that is, the

matrix formed by taking the intersection of the bottom n m rows and the

leftmost m columns of A. Let S be any size-2 m

block of S n , and let S 0 D

fy W y D Ax for some x 2 Sg. Prove that jB.S 0 ; m/j D 2 r

and that for each

block in B.S 0 ; m/, exactly 2 mr

numbers in S map to that block.

Because multiplying the zero vector by any matrix yields a zero vector, the set

of permutations of S n deﬁned by multiplying by n n 0-1 matrices with full rank

over GF.2/ cannot include all permutations of S n . Let us extend the class of per-

mutations deﬁned by matrix-vector multiplication to include an additive term, so

that x 2 S n maps to Ax C c, where c is an n-bit vector and addition is performed

over GF.2/. For example, when

A D

1 0

1 1

Notes for Appendix D 1229

and

c D

0

1

;

we get the following permutation A;c : A;c .0/ D 2, A;c .1/ D 1, A;c .2/ D 0,

A;c .3/ D 3. We call any permutation that maps x 2 S n to Ax C c, for some n n

0-1 matrix A with full rank and some n-bit vector c, a linear permutation.

d. Use a counting argument to show that the number of linear permutations of S n

is much less than the number of permutations of S n .

e. Give an example of a value of n and a permutation of S n that cannot be achieved

by any linear permutation. (Hint: For a given permutation, think about how

multiplying a matrix by a unit vector relates to the columns of the matrix.)

Appendix notes

Linear-algebra textbooks provide plenty of background information on matrices.

The books by Strang [323, 324] are particularly good.

Bibliography

[1] Milton Abramowitz and Irene A. Stegun, editors. Handbook of Mathematical Functions.

Dover, 1965.

[2] G. M. Adel’son-Vel’ski˘ı and E. M. Landis. An algorithm for the organization of information.

Soviet Mathematics Doklady, 3(5):1259–1263, 1962.

[3] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related

problems. Communications of the ACM, 31(9):1116–1127, 1988.

[4] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. Annals of Mathe-

matics, 160(2):781–793, 2004.

[5] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. The Design and Analysis of

Computer Algorithms. Addison-Wesley, 1974.

[6] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. Data Structures and Algorithms.

Addison-Wesley, 1983.

[7] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. Network Flows: Theory,

Algorithms, and Applications. Prentice Hall, 1993.

[8] Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin, and Robert E. Tarjan. Faster algorithms

for the shortest path problem. Journal of the ACM, 37:213–223, 1990.

[9] Ravindra K. Ahuja and James B. Orlin. A fast and simple algorithm for the maximum ﬂow

problem. Operations Research, 37(5):748–759, 1989.

[10] Ravindra K. Ahuja, James B. Orlin, and Robert E. Tarjan. Improved time bounds for the

maximum ﬂow problem. SIAM Journal on Computing, 18(5):939–954, 1989.

[11] Mikl´os Ajtai, Nimrod Megiddo, and Orli Waarts. Improved algorithms and analysis for

secretary problems and generalizations. In Proceedings of the 36th Annual Symposium on

Foundations of Computer Science, pages 473–482, 1995.

[12] Selim G. Akl. The Design and Analysis of Parallel Algorithms. Prentice Hall, 1989.

[13] Mohamad Akra and Louay Bazzi. On the solution of linear recurrence equations. Compu-

tational Optimization and Applications, 10(2):195–210, 1998.

[14] Noga Alon. Generating pseudo-random permutations and maximum ﬂow algorithms. In-

formation Processing Letters, 35:201–204, 1990.

1232 Bibliography

[15] Arne Andersson. Balanced search trees made simple. In Proceedings of the Third Workshop

on Algorithms and Data Structures, volume 709 of Lecture Notes in Computer Science,

pages 60–71. Springer, 1993.

[16] Arne Andersson. Faster deterministic sorting and searching in linear space. In Proceedings

of the 37th Annual Symposium on Foundations of Computer Science, pages 135–141, 1996.

[17] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear

time? Journal of Computer and System Sciences, 57:74–93, 1998.

[18] Tom M. Apostol. Calculus, volume 1. Blaisdell Publishing Company, second edition, 1967.

[19] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multipro-

grammed multiprocessors. In Proceedings of the 10th Annual ACM Symposium on Parallel

Algorithms and Architectures, pages 119–129, 1998.

[20] Sanjeev Arora. Probabilistic checking of proofs and the hardness of approximation prob-

lems. PhD thesis, University of California, Berkeley, 1994.

[21] Sanjeev Arora. The approximability of NP-hard problems. In Proceedings of the 30th

Annual ACM Symposium on Theory of Computing, pages 337–348, 1998.

[22] Sanjeev Arora. Polynomial time approximation schemes for euclidean traveling salesman

and other geometric problems. Journal of the ACM, 45(5):753–782, 1998.

[23] Sanjeev Arora and Carsten Lund. Hardness of approximations. In Dorit S. Hochbaum,

editor, Approximation Algorithms for NP-Hard Problems, pages 399–446. PWS Publishing

Company, 1997.

[24] Javed A. Aslam. A simple bound on the expected height of a randomly built binary search

tree. Technical Report TR2001-387, Dartmouth College Department of Computer Science,

2001.

[25] Mikhail J. Atallah, editor. Algorithms and Theory of Computation Handbook. CRC Press,

1999.

[26] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi.

Complexity and Approximation: Combinatorial Optimization Problems and Their Approx-

imability Properties. Springer, 1999.

[27] Shai Avidan and Ariel Shamir. Seam carving for content-aware image resizing. ACM Trans-

actions on Graphics, 26(3), article 10, 2007.

[28] Sara Baase and Alan Van Gelder. Computer Algorithms: Introduction to Design and Anal-

ysis. Addison-Wesley, third edition, 2000.

[29] Eric Bach. Private communication, 1989.

[30] Eric Bach. Number-theoretic algorithms. In Annual Review of Computer Science, volume 4,

pages 119–172. Annual Reviews, Inc., 1990.

[31] Eric Bach and Jeffrey Shallit. Algorithmic Number Theory—Volume I: Efﬁcient Algorithms.

The MIT Press, 1996.

[32] David H. Bailey, King Lee, and Horst D. Simon. Using Strassen’s algorithm to accelerate

the solution of linear systems. The Journal of Supercomputing, 4(4):357–371, 1990.

Bibliography 1233

[33] Surender Baswana, Ramesh Hariharan, and Sandeep Sen. Improved decremental algo-

rithms for maintaining transitive closure and all-pairs shortest paths. Journal of Algorithms,

62(2):74–92, 2007.

[34] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. Acta

Informatica, 1(4):290–306, 1972.

[35] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes.

Acta Informatica, 1(3):173–189, 1972.

[36] Pierre Beauchemin, Gilles Brassard, Claude Cr´epeau, Claude Goutier, and Carl Pomerance.

The generation of random numbers that are probably prime. Journal of Cryptology, 1(1):53–

64, 1988.

[37] Richard Bellman. Dynamic Programming. Princeton University Press, 1957.

[38] Richard Bellman. On a routing problem. Quarterly of Applied Mathematics, 16(1):87–90,

1958.

[39] Michael Ben-Or. Lower bounds for algebraic computation trees. In Proceedings of the

Fifteenth Annual ACM Symposium on Theory of Computing, pages 80–86, 1983.

[40] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees.

In Proceedings of the 41st Annual Symposium on Foundations of Computer Science, pages

399–409, 2000.

[41] Samuel W. Bent and John W. John. Finding the median requires 2n comparisons. In Pro-

ceedings of the Seventeenth Annual ACM Symposium on Theory of Computing, pages 213–

216, 1985.

[42] Jon L. Bentley. Writing Efﬁcient Programs. Prentice Hall, 1982.

[43] Jon L. Bentley. Programming Pearls. Addison-Wesley, 1986.

[44] Jon L. Bentley, Dorothea Haken, and James B. Saxe. A general method for solving divide-

and-conquer recurrences. SIGACT News, 12(3):36–44, 1980.

[45] Daniel Bienstock and Benjamin McClosky. Tightening simplex mixed-integer sets with

guaranteed bounds. Optimization Online, July 2008.

[46] Patrick Billingsley. Probability and Measure. John Wiley & Sons, second edition, 1986.

[47] Guy E. Blelloch. Scan Primitives and Parallel Vector Models. PhD thesis, Department of

Electrical Engineering and Computer Science, MIT, 1989. Available as MIT Laboratory for

Computer Science Technical Report MIT/LCS/TR-463.

[48] Guy E. Blelloch. Programming parallel algorithms. Communications of the ACM,

39(3):85–97, 1996.

[49] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efﬁcient scheduling for

languages with ﬁne-grained parallelism. In Proceedings of the 7th Annual ACM Symposium

on Parallel Algorithms and Architectures, pages 1–12, 1995.

[50] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan.

Time bounds for selection. Journal of Computer and System Sciences, 7(4):448–461, 1973.

[51] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson,

Keith H. Randall, and Yuli Zhou. Cilk: An efﬁcient multithreaded runtime system. Journal

of Parallel and Distributed Computing, 37(1):55–69, 1996.

1234 Bibliography

[52] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by

work stealing. Journal of the ACM, 46(5):720–748, 1999.

[53] B´ela Bollob´as. Random Graphs. Academic Press, 1985.

[54] Gilles Brassard and Paul Bratley. Fundamentals of Algorithmics. Prentice Hall, 1996.

[55] Richard P. Brent. The parallel evaluation of general arithmetic expressions. Journal of the

ACM, 21(2):201–206, 1974.

[56] Richard P. Brent. An improved Monte Carlo factorization algorithm. BIT, 20(2):176–184,

1980.

[57] J. P. Buhler, H. W. Lenstra, Jr., and Carl Pomerance. Factoring integers with the number

ﬁeld sieve. In A. K. Lenstra and H. W. Lenstra, Jr., editors, The Development of the Number

Field Sieve, volume 1554 of Lecture Notes in Mathematics, pages 50–94. Springer, 1993.

[58] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. Journal of

Computer and System Sciences, 18(2):143–154, 1979.

[59] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. Using OpenMP: Portable Shared

Memory Parallel Programming. The MIT Press, 2007.

[60] Bernard Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type com-

plexity. Journal of the ACM, 47(6):1028–1047, 2000.

[61] Joseph Cheriyan and Torben Hagerup. A randomized maximum-ﬂow algorithm. SIAM

Journal on Computing, 24(2):203–226, 1995.

[62] Joseph Cheriyan and S. N. Maheshwari. Analysis of preﬂow push algorithms for maximum

network ﬂow. SIAM Journal on Computing, 18(6):1057–1086, 1989.

[63] Boris V. Cherkassky and Andrew V. Goldberg. On implementing the push-relabel method

for the maximum ﬂow problem. Algorithmica, 19(4):390–410, 1997.

[64] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest paths algorithms:

Theory and experimental evaluation. Mathematical Programming, 73(2):129–174, 1996.

[65] Boris V. Cherkassky, Andrew V. Goldberg, and Craig Silverstein. Buckets, heaps, lists and

monotone priority queues. SIAM Journal on Computing, 28(4):1326–1346, 1999.

[66] H. Chernoff. A measure of asymptotic efﬁciency for tests of a hypothesis based on the sum

of observations. Annals of Mathematical Statistics, 23(4):493–507, 1952.

[67] Kai Lai Chung. Elementary Probability Theory with Stochastic Processes. Springer, 1974.

[68] V. Chv´atal. A greedy heuristic for the set-covering problem. Mathematics of Operations

Research, 4(3):233–235, 1979.

[69] V. Chv´atal. Linear Programming. W. H. Freeman and Company, 1983.

[70] V. Chv´atal, D. A. Klarner, and D. E. Knuth. Selected combinatorial research problems.

Technical Report STAN-CS-72-292, Computer Science Department, Stanford University,

1972.

[71] Cilk Arts, Inc., Burlington, Massachusetts. Cilk++ Programmer’s Guide, 2008. Available

at http://www.cilk.com/archive/docs/cilk1guide.

Bibliography 1235

[72] Alan Cobham. The intrinsic computational difﬁculty of functions. In Proceedings of the

1964 Congress for Logic, Methodology, and the Philosophy of Science, pages 24–30. North-

Holland, 1964.

[73] H. Cohen and H. W. Lenstra, Jr. Primality testing and Jacobi sums. Mathematics of Com-

putation, 42(165):297–330, 1984.

[74] D. Comer. The ubiquitous B-tree. ACM Computing Surveys, 11(2):121–137, 1979.

[75] Stephen Cook. The complexity of theorem proving procedures. In Proceedings of the Third

Annual ACM Symposium on Theory of Computing, pages 151–158, 1971.

[76] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex

Fourier series. Mathematics of Computation, 19(90):297–301, 1965.

[77] Don Coppersmith. Modiﬁcations to the number ﬁeld sieve. Journal of Cryptology,

6(3):169–180, 1993.

[78] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progression.

Journal of Symbolic Computation, 9(3):251–280, 1990.

[79] Thomas H. Cormen, Thomas Sundquist, and Leonard F. Wisniewski. Asymptotically tight

bounds for performing BMMC permutations on parallel disk systems. SIAM Journal on

Computing, 28(1):105–136, 1998.

[80] Don Dailey and Charles E. Leiserson. Using Cilk to write multiprocessor chess programs.

In H. J. van den Herik and B. Monien, editors, Advances in Computer Games, volume 9,

pages 25–52. University of Maastricht, Netherlands, 2001.

[81] Paolo D’Alberto and Alexandru Nicolau. Adaptive Strassen’s matrix multiplication. In

Proceedings of the 21st Annual International Conference on Supercomputing, pages 284–

292, June 2007.

[82] Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. Algorithms. McGraw-Hill,

2008.

[83] Roman Dementiev, Lutz Kettner, Jens Mehnert, and Peter Sanders. Engineering a sorted list

data structure for 32 bit keys. In Proceedings of the Sixth Workshop on Algorithm Engineer-

ing and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics,

pages 142–151, January 2004.

[84] Camil Demetrescu and Giuseppe F. Italiano. Fully dynamic all pairs shortest paths with real

edge weights. Journal of Computer and System Sciences, 72(5):813–837, 2006.

[85] Eric V. Denardo and Bennett L. Fox. Shortest-route methods: 1. Reaching, pruning, and

buckets. Operations Research, 27(1):161–186, 1979.

[86] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans

Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. SIAM

Journal on Computing, 23(4):738–761, 1994.

[87] Whitﬁeld Difﬁe and Martin E. Hellman. New directions in cryptography. IEEE Transac-

tions on Information Theory, IT-22(6):644–654, 1976.

[88] E. W. Dijkstra. A note on two problems in connexion with graphs. Numerische Mathematik,

1(1):269–271, 1959.

1236 Bibliography

[89] E. A. Dinic. Algorithm for solution of a problem of maximum ﬂow in a network with power

estimation. Soviet Mathematics Doklady, 11(5):1277–1280, 1970.

[90] Brandon Dixon, Monika Rauch, and Robert E. Tarjan. Veriﬁcation and sensitivity analysis

of minimum spanning trees in linear time. SIAM Journal on Computing, 21(6):1184–1192,

1992.

[91] John D. Dixon. Factorization and primality tests. The American Mathematical Monthly,

91(6):333–352, 1984.

[92] Dorit Dor, Johan H˚astad, Staffan Ulfberg, and Uri Zwick. On lower bounds for selecting

the median. SIAM Journal on Discrete Mathematics, 14(3):299–311, 2001.

[93] Dorit Dor and Uri Zwick. Selecting the median. SIAM Journal on Computing, 28(5):1722–

1758, 1999.

[94] Dorit Dor and Uri Zwick. Median selection requires .2 C /n comparisons. SIAM Journal

on Discrete Mathematics, 14(3):312–325, 2001.

[95] Alvin W. Drake. Fundamentals of Applied Probability Theory. McGraw-Hill, 1967.

[96] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps:

An alternative to Fibonacci heaps with applications to parallel computation. Communica-

tions of the ACM, 31(11):1343–1354, 1988.

[97] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data

structures persistent. Journal of Computer and System Sciences, 38(1):86–124, 1989.

[98] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus efﬁciency in

parallel systems. IEEE Transactions on Computers, 38(3):408–423, 1989.

[99] Herbert Edelsbrunner. Algorithms in Combinatorial Geometry, volume 10 of EATCS Mono-

graphs on Theoretical Computer Science. Springer, 1987.

[100] Jack Edmonds. Paths, trees, and ﬂowers. Canadian Journal of Mathematics, 17:449–467,

1965.

[101] Jack Edmonds. Matroids and the greedy algorithm. Mathematical Programming, 1(1):127–

136, 1971.

[102] Jack Edmonds and Richard M. Karp. Theoretical improvements in the algorithmic efﬁ-

ciency for network ﬂow problems. Journal of the ACM, 19(2):248–264, 1972.

[103] Shimon Even. Graph Algorithms. Computer Science Press, 1979.

[104] William Feller. An Introduction to Probability Theory and Its Applications. John Wiley &

Sons, third edition, 1968.

[105] Robert W. Floyd. Algorithm 97 (SHORTEST PATH). Communications of the ACM,

5(6):345, 1962.

[106] Robert W. Floyd. Algorithm 245 (TREESORT). Communications of the ACM, 7(12):701,

1964.

[107] Robert W. Floyd. Permuting information in idealized two-level storage. In Raymond E.

Miller and James W. Thatcher, editors, Complexity of Computer Computations, pages 105–

109. Plenum Press, 1972.

Bibliography 1237

[108] Robert W. Floyd and Ronald L. Rivest. Expected time bounds for selection. Communica-

tions of the ACM, 18(3):165–172, 1975.

[109] Lestor R. Ford, Jr. and D. R. Fulkerson. Flows in Networks. Princeton University Press,

1962.

[110] Lestor R. Ford, Jr. and Selmer M. Johnson. A tournament problem. The American Mathe-

matical Monthly, 66(5):387–389, 1959.

[111] Michael L. Fredman. New bounds on the complexity of the shortest path problem. SIAM

Journal on Computing, 5(1):83–89, 1976.

[112] Michael L. Fredman, J´anos Koml´os, and Endre Szemer´edi. Storing a sparse table with O.1/

worst case access time. Journal of the ACM, 31(3):538–544, 1984.

[113] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data struc-

tures. In Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing,

pages 345–354, 1989.

[114] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved

network optimization algorithms. Journal of the ACM, 34(3):596–615, 1987.

[115] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with

fusion trees. Journal of Computer and System Sciences, 47(3):424–436, 1993.

[116] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum span-

ning trees and shortest paths. Journal of Computer and System Sciences, 48(3):533–551,

1994.

[117] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. Proceed-

ings of the IEEE, 93(2):216–231, 2005.

[118] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5

multithreaded language. In Proceedings of the 1998 ACM SIGPLAN Conference on Pro-

gramming Language Design and Implementation, pages 212–223, 1998.

[119] Harold N. Gabow. Path-based depth-ﬁrst search for strong and biconnected components.

Information Processing Letters, 74(3–4):107–114, 2000.

[120] Harold N. Gabow, Z. Galil, T. Spencer, and Robert E. Tarjan. Efﬁcient algorithms for ﬁnd-

ing minimum spanning trees in undirected and directed graphs. Combinatorica, 6(2):109–

122, 1986.

[121] Harold N. Gabow and Robert E. Tarjan. A linear-time algorithm for a special case of disjoint

set union. Journal of Computer and System Sciences, 30(2):209–221, 1985.

[122] Harold N. Gabow and Robert E. Tarjan. Faster scaling algorithms for network problems.

SIAM Journal on Computing, 18(5):1013–1036, 1989.

[123] Zvi Galil and Oded Margalit. All pairs shortest distances for graphs with small integer

length edges. Information and Computation, 134(2):103–139, 1997.

[124] Zvi Galil and Oded Margalit. All pairs shortest paths for graphs with small integer length

edges. Journal of Computer and System Sciences, 54(2):243–254, 1997.

[125] Zvi Galil and Kunsoo Park. Dynamic programming with convexity, concavity and sparsity.

Theoretical Computer Science, 92(1):49–76, 1992.

1238 Bibliography

[126] Zvi Galil and Joel Seiferas. Time-space-optimal string matching. Journal of Computer and

System Sciences, 26(3):280–294, 1983.

[127] Igal Galperin and Ronald L. Rivest. Scapegoat trees. In Proceedings of the 4th ACM-SIAM

Symposium on Discrete Algorithms, pages 165–174, 1993.

[128] Michael R. Garey, R. L. Graham, and J. D. Ullman. Worst-case analyis of memory al-

location algorithms. In Proceedings of the Fourth Annual ACM Symposium on Theory of

Computing, pages 143–150, 1972.

[129] Michael R. Garey and David S. Johnson. Computers and Intractability: A Guide to the

Theory of NP-Completeness. W. H. Freeman, 1979.

[130] Saul Gass. Linear Programming: Methods and Applications. International Thomson Pub-

lishing, fourth edition, 1975.

[131] F˘anic˘a Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by

cliques, and maximum independent set of a chordal graph. SIAM Journal on Computing,

1(2):180–187, 1972.

[132] Alan George and Joseph W-H Liu. Computer Solution of Large Sparse Positive Deﬁnite

Systems. Prentice Hall, 1981.

[133] E. N. Gilbert and E. F. Moore. Variable-length binary encodings. Bell System Technical

Journal, 38(4):933–967, 1959.

[134] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for

maximum cut and satisﬁability problems using semideﬁnite programming. Journal of the

ACM, 42(6):1115–1145, 1995.

[135] Michel X. Goemans and David P. Williamson. The primal-dual method for approximation

algorithms and its application to network design problems. In Dorit S. Hochbaum, editor,

Approximation Algorithms for NP-Hard Problems, pages 144–191. PWS Publishing Com-

pany, 1997.

[136] Andrew V. Goldberg. Efﬁcient Graph Algorithms for Sequential and Parallel Computers.

PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1987.

[137] Andrew V. Goldberg. Scaling algorithms for the shortest paths problem. SIAM Journal on

Computing, 24(3):494–504, 1995.

[138] Andrew V. Goldberg and Satish Rao. Beyond the ﬂow decomposition barrier. Journal of

the ACM, 45(5):783–797, 1998.

[139] Andrew V. Goldberg, ´Eva Tardos, and Robert E. Tarjan. Network ﬂow algorithms. In Bern-

hard Korte, L´aszl´o Lov´asz, Hans J¨urgen Pr¨omel, and Alexander Schrijver, editors, Paths,

Flows, and VLSI-Layout, pages 101–164. Springer, 1990.

[140] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum ﬂow problem.

Journal of the ACM, 35(4):921–940, 1988.

[141] D. Goldfarb and M. J. Todd. Linear programming. In G. L. Nemhauser, A. H. G. Rinnooy-

Kan, and M. J. Todd, editors, Handbook in Operations Research and Management Science,

Vol. 1, Optimization, pages 73–170. Elsevier Science Publishers, 1989.

[142] Shaﬁ Goldwasser and Silvio Micali. Probabilistic encryption. Journal of Computer and

System Sciences, 28(2):270–299, 1984.

Bibliography 1239

[143] Shaﬁ Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure

against adaptive chosen-message attacks. SIAM Journal on Computing, 17(2):281–308,

1988.

[144] Gene H. Golub and Charles F. Van Loan. Matrix Computations. The Johns Hopkins Uni-

versity Press, third edition, 1996.

[145] G. H. Gonnet. Handbook of Algorithms and Data Structures. Addison-Wesley, 1984.

[146] Rafael C. Gonzalez and Richard E. Woods. Digital Image Processing. Addison-Wesley,

1992.

[147] Michael T. Goodrich and Roberto Tamassia. Data Structures and Algorithms in Java. John

Wiley & Sons, 1998.

[148] Michael T. Goodrich and Roberto Tamassia. Algorithm Design: Foundations, Analysis, and

Internet Examples. John Wiley & Sons, 2001.

[149] Ronald L. Graham. Bounds for certain multiprocessor anomalies. Bell System Technical

Journal, 45(9):1563–1581, 1966.

[150] Ronald L. Graham. An efﬁcient algorithm for determining the convex hull of a ﬁnite planar

set. Information Processing Letters, 1(4):132–133, 1972.

[151] Ronald L. Graham and Pavol Hell. On the history of the minimum spanning tree problem.

Annals of the History of Computing, 7(1):43–57, 1985.

[152] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. Concrete Mathematics.

Addison-Wesley, second edition, 1994.

[153] David Gries. The Science of Programming. Springer, 1981.

[154] M. Gr¨otschel, L´aszl´o Lov´asz, and Alexander Schrijver. Geometric Algorithms and Combi-

natorial Optimization. Springer, 1988.

[155] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In

Proceedings of the 19th Annual Symposium on Foundations of Computer Science, pages

8–21, 1978.

[156] Dan Gusﬁeld. Algorithms on Strings, Trees, and Sequences: Computer Science and Com-

putational Biology. Cambridge University Press, 1997.

[157] H. Halberstam and R. E. Ingram, editors. The Mathematical Papers of Sir William Rowan

Hamilton, volume III (Algebra). Cambridge University Press, 1967.

[158] Yijie Han. Improved fast integer sorting in linear space. In Proceedings of the 12th ACM-

SIAM Symposium on Discrete Algorithms, pages 793–796, 2001.

[159] Yijie Han. An O.n

3

.log log n= log n/

5=4

/ time algorithm for all pairs shortest path. Algo-

rithmica, 51(4):428–434, 2008.

[160] Frank Harary. Graph Theory. Addison-Wesley, 1969.

[161] Gregory C. Harfst and Edward M. Reingold. A potential-based amortized analysis of the

union-ﬁnd data structure. SIGACT News, 31(3):86–95, 2000.

[162] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. Transac-

tions of the American Mathematical Society, 117:285–306, May 1965.

1240 Bibliography

[163] Michael T. Heideman, Don H. Johnson, and C. Sidney Burrus. Gauss and the history of the

Fast Fourier Transform. IEEE ASSP Magazine, 1(4):14–21, 1984.

[164] Monika R. Henzinger and Valerie King. Fully dynamic biconnectivity and transitive clo-

sure. In Proceedings of the 36th Annual Symposium on Foundations of Computer Science,

pages 664–672, 1995.

[165] Monika R. Henzinger and Valerie King. Randomized fully dynamic graph algorithms with

polylogarithmic time per operation. Journal of the ACM, 46(4):502–516, 1999.

[166] Monika R. Henzinger, Satish Rao, and Harold N. Gabow. Computing vertex connectivity:

New bounds from old techniques. Journal of Algorithms, 34(2):222–250, 2000.

[167] Nicholas J. Higham. Exploiting fast matrix multiplication within the level 3 BLAS. ACM

Transactions on Mathematical Software, 16(4):352–368, 1990.

[168] W. Daniel Hillis and Jr. Guy L. Steele. Data parallel algorithms. Communications of the

ACM, 29(12):1170–1183, 1986.

[169] C. A. R. Hoare. Algorithm 63 (PARTITION) and algorithm 65 (FIND). Communications

of the ACM, 4(7):321–322, 1961.

[170] C. A. R. Hoare. Quicksort. Computer Journal, 5(1):10–15, 1962.

[171] Dorit S. Hochbaum. Efﬁcient bounds for the stable set, vertex cover and set packing prob-

lems. Discrete Applied Mathematics, 6(3):243–254, 1983.

[172] Dorit S. Hochbaum, editor. Approximation Algorithms for NP-Hard Problems. PWS Pub-

lishing Company, 1997.

[173] W. Hoeffding. On the distribution of the number of successes in independent trials. Annals

of Mathematical Statistics, 27(3):713–721, 1956.

[174] Micha Hofri. Probabilistic Analysis of Algorithms. Springer, 1987.

[175] Micha Hofri. Analysis of Algorithms. Oxford University Press, 1995.

[176] John E. Hopcroft and Richard M. Karp. An n

5=2

algorithm for maximum matchings in

bipartite graphs. SIAM Journal on Computing, 2(4):225–231, 1973.

[177] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to Automata The-

ory, Languages, and Computation. Addison Wesley, third edition, 2006.

[178] John E. Hopcroft and Robert E. Tarjan. Efﬁcient algorithms for graph manipulation. Com-

munications of the ACM, 16(6):372–378, 1973.

[179] John E. Hopcroft and Jeffrey D. Ullman. Set merging algorithms. SIAM Journal on Com-

puting, 2(4):294–303, 1973.

[180] John E. Hopcroft and Jeffrey D. Ullman. Introduction to Automata Theory, Languages, and

Computation. Addison-Wesley, 1979.

[181] Ellis Horowitz, Sartaj Sahni, and Sanguthevar Rajasekaran. Computer Algorithms. Com-

puter Science Press, 1998.

[182] T. C. Hu and M. T. Shing. Computation of matrix chain products. Part I. SIAM Journal on

Computing, 11(2):362–373, 1982.

[183] T. C. Hu and M. T. Shing. Computation of matrix chain products. Part II. SIAM Journal on

Computing, 13(2):228–251, 1984.

Bibliography 1241

[184] T. C. Hu and A. C. Tucker. Optimal computer search trees and variable-length alphabetic

codes. SIAM Journal on Applied Mathematics, 21(4):514–532, 1971.

[185] David A. Huffman. A method for the construction of minimum-redundancy codes. Pro-

ceedings of the IRE, 40(9):1098–1101, 1952.

[186] Steven Huss-Lederman, Elaine M. Jacobson, Jeremy R. Johnson, Anna Tsao, and Thomas

Turnbull. Implementation of Strassen’s algorithm for matrix multiplication. In Proceedings

of the 1996 ACM/IEEE Conference on Supercomputing, article 32, 1996.

[187] Oscar H. Ibarra and Chul E. Kim. Fast approximation algorithms for the knapsack and sum

of subset problems. Journal of the ACM, 22(4):463–468, 1975.

[188] E. J. Isaac and R. C. Singleton. Sorting by address calculation. Journal of the ACM,

3(3):169–174, 1956.

[189] R. A. Jarvis. On the identiﬁcation of the convex hull of a ﬁnite set of points in the plane.

Information Processing Letters, 2(1):18–21, 1973.

[190] David S. Johnson. Approximation algorithms for combinatorial problems. Journal of Com-

puter and System Sciences, 9(3):256–278, 1974.

[191] David S. Johnson. The NP-completeness column: An ongoing guide—The tale of the sec-

ond prover. Journal of Algorithms, 13(3):502–524, 1992.

[192] Donald B. Johnson. Efﬁcient algorithms for shortest paths in sparse networks. Journal of

the ACM, 24(1):1–13, 1977.

[193] Richard Johnsonbaugh and Marcus Schaefer. Algorithms. Pearson Prentice Hall, 2004.

[194] A. Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. Soviet

Physics—Doklady, 7(7):595–596, 1963. Translation of an article in Doklady Akademii Nauk

SSSR, 145(2), 1962.

[195] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm

to ﬁnd minimum spanning trees. Journal of the ACM, 42(2):321–328, 1995.

[196] David R. Karger, Daphne Koller, and Steven J. Phillips. Finding the hidden path: Time

bounds for all-pairs shortest paths. SIAM Journal on Computing, 22(6):1199–1217, 1993.

[197] Howard Karloff. Linear Programming. Birkh¨auser, 1991.

[198] N. Karmarkar. A new polynomial-time algorithm for linear programming. Combinatorica,

4(4):373–395, 1984.

[199] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and

James W. Thatcher, editors, Complexity of Computer Computations, pages 85–103. Plenum

Press, 1972.

[200] Richard M. Karp. An introduction to randomized algorithms. Discrete Applied Mathemat-

ics, 34(1–3):165–201, 1991.

[201] Richard M. Karp and Michael O. Rabin. Efﬁcient randomized pattern-matching algorithms.

IBM Journal of Research and Development, 31(2):249–260, 1987.

[202] A. V. Karzanov. Determining the maximal ﬂow in a network by the method of preﬂows.

Soviet Mathematics Doklady, 15(2):434–437, 1974.

1242 Bibliography

[203] Valerie King. A simpler minimum spanning tree veriﬁcation algorithm. Algorithmica,

18(2):263–270, 1997.

[204] Valerie King, Satish Rao, and Robert E. Tarjan. A faster deterministic maximum ﬂow algo-

rithm. Journal of Algorithms, 17(3):447–474, 1994.

[205] Jeffrey H. Kingston. Algorithms and Data Structures: Design, Correctness, Analysis.

Addison-Wesley, second edition, 1997.

[206] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? SIAM Journal

on Computing, 15(2):287–299, 1986.

[207] Philip N. Klein and Neal E. Young. Approximation algorithms for NP-hard optimization

problems. In CRC Handbook on Algorithms, pages 34-1–34-19. CRC Press, 1999.

[208] Jon Kleinberg and ´Eva Tardos. Algorithm Design. Addison-Wesley, 2006.

[209] Donald E. Knuth. Fundamental Algorithms, volume 1 of The Art of Computer Program-

ming. Addison-Wesley, 1968. Third edition, 1997.

[210] Donald E. Knuth. Seminumerical Algorithms, volume 2 of The Art of Computer Program-

ming. Addison-Wesley, 1969. Third edition, 1997.

[211] Donald E. Knuth. Sorting and Searching, volume 3 of The Art of Computer Programming.

Addison-Wesley, 1973. Second edition, 1998.

[212] Donald E. Knuth. Optimum binary search trees. Acta Informatica, 1(1):14–25, 1971.

[213] Donald E. Knuth. Big omicron and big omega and big theta. SIGACT News, 8(2):18–23,

1976.

[214] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in

strings. SIAM Journal on Computing, 6(2):323–350, 1977.

[215] J. Koml´os. Linear veriﬁcation for spanning trees. Combinatorica, 5(1):57–65, 1985.

[216] Bernhard Korte and L´aszl´o Lov´asz. Mathematical structures underlying greedy algorithms.

In F. Gecseg, editor, Fundamentals of Computation Theory, volume 117 of Lecture Notes in

Computer Science, pages 205–209. Springer, 1981.

[217] Bernhard Korte and L´aszl´o Lov´asz. Structural properties of greedoids. Combinatorica,

3(3–4):359–374, 1983.

[218] Bernhard Korte and L´aszl´o Lov´asz. Greedoids—A structural framework for the greedy

algorithm. In W. Pulleybank, editor, Progress in Combinatorial Optimization, pages 221–

243. Academic Press, 1984.

[219] Bernhard Korte and L´aszl´o Lov´asz. Greedoids and linear objective functions. SIAM Journal

on Algebraic and Discrete Methods, 5(2):229–238, 1984.

[220] Dexter C. Kozen. The Design and Analysis of Algorithms. Springer, 1992.

[221] David W. Krumme, George Cybenko, and K. N. Venkataraman. Gossiping in minimal time.

SIAM Journal on Computing, 21(1):111–139, 1992.

[222] Joseph B. Kruskal, Jr. On the shortest spanning subtree of a graph and the traveling salesman

problem. Proceedings of the American Mathematical Society, 7(1):48–50, 1956.

[223] Leslie Lamport. How to make a multiprocessor computer that correctly executes multipro-

cess programs. IEEE Transactions on Computers, C-28(9):690–691, 1979.

Bibliography 1243

[224] Eugene L. Lawler. Combinatorial Optimization: Networks and Matroids. Holt, Rinehart,

and Winston, 1976.

[225] Eugene L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors. The

Traveling Salesman Problem. John Wiley & Sons, 1985.

[226] C. Y. Lee. An algorithm for path connection and its applications. IRE Transactions on

Electronic Computers, EC-10(3):346–365, 1961.

[227] Tom Leighton. Tight bounds on the complexity of parallel sorting. IEEE Transactions on

Computers, C-34(4):344–354, 1985.

[228] Tom Leighton. Notes on better master theorems for divide-and-conquer recurrences. Class

notes. Available at http://citeseer.ist.psu.edu/252350.html, October 1996.

[229] Tom Leighton and Satish Rao. Multicommodity max-ﬂow min-cut theorems and their use

in designing approximation algorithms. Journal of the ACM, 46(6):787–832, 1999.

[230] Daan Leijen and Judd Hall. Optimize managed code for multi-core machines. MSDN

Magazine, October 2007.

[231] Debra A. Lelewer and Daniel S. Hirschberg. Data compression. ACM Computing Surveys,

19(3):261–296, 1987.

[232] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, and J. M. Pollard. The number ﬁeld sieve.

In A. K. Lenstra and H. W. Lenstra, Jr., editors, The Development of the Number Field Sieve,

volume 1554 of Lecture Notes in Mathematics, pages 11–42. Springer, 1993.

[233] H. W. Lenstra, Jr. Factoring integers with elliptic curves. Annals of Mathematics,

126(3):649–673, 1987.

[234] L. A. Levin. Universal sorting problems. Problemy Peredachi Informatsii, 9(3):265–266,

1973. In Russian.

[235] Anany Levitin. Introduction to the Design & Analysis of Algorithms. Addison-Wesley,

2007.

[236] Harry R. Lewis and Christos H. Papadimitriou. Elements of the Theory of Computation.

Prentice Hall, second edition, 1998.

[237] C. L. Liu. Introduction to Combinatorial Mathematics. McGraw-Hill, 1968.

[238] L´aszl´o Lov´asz. On the ratio of optimal integral and fractional covers. Discrete Mathemat-

ics, 13(4):383–390, 1975.

[239] L´aszl´o Lov´asz and M. D. Plummer. Matching Theory, volume 121 of Annals of Discrete

Mathematics. North Holland, 1986.

[240] Bruce M. Maggs and Serge A. Plotkin. Minimum-cost spanning tree as a path-ﬁnding

problem. Information Processing Letters, 26(6):291–293, 1988.

[241] Michael Main. Data Structures and Other Objects Using Java. Addison-Wesley, 1999.

[242] Udi Manber. Introduction to Algorithms: A Creative Approach. Addison-Wesley, 1989.

[243] Conrado Mart´ınez and Salvador Roura. Randomized binary search trees. Journal of the

ACM, 45(2):288–323, 1998.

[244] William J. Masek and Michael S. Paterson. A faster algorithm computing string edit dis-

tances. Journal of Computer and System Sciences, 20(1):18–31, 1980.

1244 Bibliography

[245] H. A. Maurer, Th. Ottmann, and H.-W. Six. Implementing dictionaries using binary trees of

very small height. Information Processing Letters, 5(1):11–14, 1976.

[246] Ernst W. Mayr, Hans J¨urgen Pr¨omel, and Angelika Steger, editors. Lectures on Proof Veriﬁ-

cation and Approximation Algorithms, volume 1367 of Lecture Notes in Computer Science.

Springer, 1998.

[247] C. C. McGeoch. All pairs shortest paths and the essential subgraph. Algorithmica,

13(5):426–441, 1995.

[248] M. D. McIlroy. A killer adversary for quicksort. Software—Practice and Experience,

29(4):341–344, 1999.

[249] Kurt Mehlhorn. Sorting and Searching, volume 1 of Data Structures and Algorithms.

Springer, 1984.

[250] Kurt Mehlhorn. Graph Algorithms and NP-Completeness, volume 2 of Data Structures and

Algorithms. Springer, 1984.

[251] Kurt Mehlhorn. Multidimensional Searching and Computational Geometry, volume 3 of

Data Structures and Algorithms. Springer, 1984.

[252] Kurt Mehlhorn and Stefan N¨aher. Bounded ordered dictionaries in O.log log N / time and

O.n/ space. Information Processing Letters, 35(4):183–189, 1990.

[253] Kurt Mehlhorn and Stefan N¨aher. LEDA: A Platform for Combinatorial and Geometric

Computing. Cambridge University Press, 1999.

[254] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. Handbook of Applied

Cryptography. CRC Press, 1997.

[255] Gary L. Miller. Riemann’s hypothesis and tests for primality. Journal of Computer and

System Sciences, 13(3):300–317, 1976.

[256] John C. Mitchell. Foundations for Programming Languages. The MIT Press, 1996.

[257] Joseph S. B. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: A sim-

ple polynomial-time approximation scheme for geometric TSP, k-MST, and related prob-

lems. SIAM Journal on Computing, 28(4):1298–1309, 1999.

[258] Louis Monier. Algorithmes de Factorisation D’Entiers. PhD thesis, L’Universit´e Paris-Sud,

1980.

[259] Louis Monier. Evaluation and comparison of two efﬁcient probabilistic primality testing

algorithms. Theoretical Computer Science, 12(1):97–108, 1980.

[260] Edward F. Moore. The shortest path through a maze. In Proceedings of the International

Symposium on the Theory of Switching, pages 285–292. Harvard University Press, 1959.

[261] Rajeev Motwani, Joseph (Sefﬁ) Naor, and Prabakhar Raghavan. Randomized approxima-

tion algorithms in combinatorial optimization. In Dorit Hochbaum, editor, Approximation

Algorithms for NP-Hard Problems, chapter 11, pages 447–481. PWS Publishing Company,

1997.

[262] Rajeev Motwani and Prabhakar Raghavan. Randomized Algorithms. Cambridge University

Press, 1995.

[263] J. I. Munro and V. Raman. Fast stable in-place sorting with O.n/ data moves. Algorithmica,

16(2):151–160, 1996.

Bibliography 1245

[264] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. SIAM Journal

on Computing, 2(1):33–43, 1973.

[265] Ivan Niven and Herbert S. Zuckerman. An Introduction to the Theory of Numbers. John

Wiley & Sons, fourth edition, 1980.

[266] Alan V. Oppenheim and Ronald W. Schafer, with John R. Buck. Discrete-Time Signal

Processing. Prentice Hall, second edition, 1998.

[267] Alan V. Oppenheim and Alan S. Willsky, with S. Hamid Nawab. Signals and Systems.

Prentice Hall, second edition, 1997.

[268] James B. Orlin. A polynomial time primal network simplex algorithm for minimum cost

ﬂows. Mathematical Programming, 78(1):109–129, 1997.

[269] Joseph O’Rourke. Computational Geometry in C. Cambridge University Press, second

edition, 1998.

[270] Christos H. Papadimitriou. Computational Complexity. Addison-Wesley, 1994.

[271] Christos H. Papadimitriou and Kenneth Steiglitz. Combinatorial Optimization: Algorithms

and Complexity. Prentice Hall, 1982.

[272] Michael S. Paterson. Progress in selection. In Proceedings of the Fifth Scandinavian Work-

shop on Algorithm Theory, pages 368–379, 1996.

[273] Mihai Pˇatras¸cu and Mikkel Thorup. Time-space trade-offs for predecessor search. In Pro-

ceedings of the 38th Annual ACM Symposium on Theory of Computing, pages 232–240,

2006.

[274] Mihai Pˇatras¸cu and Mikkel Thorup. Randomization does not help searching predecessors.

In Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms, pages 555–564,

2007.

[275] Pavel A. Pevzner. Computational Molecular Biology: An Algorithmic Approach. The MIT

Press, 2000.

[276] Steven Phillips and Jeffery Westbrook. Online load balancing and network ﬂow. In Pro-

ceedings of the 25th Annual ACM Symposium on Theory of Computing, pages 402–411,

1993.

[277] J. M. Pollard. A Monte Carlo method for factorization. BIT, 15(3):331–334, 1975.

[278] J. M. Pollard. Factoring with cubic integers. In A. K. Lenstra and H. W. Lenstra, Jr., editors,

The Development of the Number Field Sieve, volume 1554 of Lecture Notes in Mathematics,

pages 4–10. Springer, 1993.

[279] Carl Pomerance. On the distribution of pseudoprimes. Mathematics of Computation,

37(156):587–593, 1981.

[280] Carl Pomerance, editor. Proceedings of the AMS Symposia in Applied Mathematics: Com-

putational Number Theory and Cryptography. American Mathematical Society, 1990.

[281] William K. Pratt. Digital Image Processing. John Wiley & Sons, fourth edition, 2007.

[282] Franco P. Preparata and Michael Ian Shamos. Computational Geometry: An Introduction.

Springer, 1985.

1246 Bibliography

[283] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Numer-

ical Recipes in C++: The Art of Scientiﬁc Computing. Cambridge University Press, second

edition, 2002.

[284] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Numer-

ical Recipes: The Art of Scientiﬁc Computing. Cambridge University Press, third edition,

2007.

[285] R. C. Prim. Shortest connection networks and some generalizations. Bell System Technical

Journal, 36(6):1389–1401, 1957.

[286] William Pugh. Skip lists: A probabilistic alternative to balanced trees. Communications of

the ACM, 33(6):668–676, 1990.

[287] Paul W. Purdom, Jr. and Cynthia A. Brown. The Analysis of Algorithms. Holt, Rinehart,

and Winston, 1985.

[288] Michael O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, Algorithms and Com-

plexity: New Directions and Recent Results, pages 21–39. Academic Press, 1976.

[289] Michael O. Rabin. Probabilistic algorithm for testing primality. Journal of Number Theory,

12(1):128–138, 1980.

[290] P. Raghavan and C. D. Thompson. Randomized rounding: A technique for provably good

algorithms and algorithmic proofs. Combinatorica, 7(4):365–374, 1987.

[291] Rajeev Raman. Recent results on the single-source shortest paths problem. SIGACT News,

28(2):81–87, 1997.

[292] James Reinders. Intel Threading Building Blocks: Outﬁtting C++ for Multi-core Processor

Parallelism. O’Reilly Media, Inc., 2007.

[293] Edward M. Reingold, J¨urg Nievergelt, and Narsingh Deo. Combinatorial Algorithms: The-

ory and Practice. Prentice Hall, 1977.

[294] Edward M. Reingold, Kenneth J. Urban, and David Gries. K-M-P string matching revisited.

Information Processing Letters, 64(5):217–223, 1997.

[295] Hans Riesel. Prime Numbers and Computer Methods for Factorization, volume 126 of

Progress in Mathematics. Birkh¨auser, second edition, 1994.

[296] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital

signatures and public-key cryptosystems. Communications of the ACM, 21(2):120–126,

1978. See also U.S. Patent 4,405,829.

[297] Herbert Robbins. A remark on Stirling’s formula. American Mathematical Monthly,

62(1):26–29, 1955.

[298] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis. An analysis of several heuristics for the

traveling salesman problem. SIAM Journal on Computing, 6(3):563–581, 1977.

[299] Salvador Roura. An improved master theorem for divide-and-conquer recurrences. In

Proceedings of Automata, Languages and Programming, 24th International Colloquium,

ICALP’97, volume 1256 of Lecture Notes in Computer Science, pages 449–459. Springer,

1997.

[300] Y. A. Rozanov. Probability Theory: A Concise Course. Dover, 1969.

Bibliography 1247

[301] S. Sahni and T. Gonzalez. P-complete approximation problems. Journal of the ACM,

23(3):555–565, 1976.

[302] A. Sch¨onhage, M. Paterson, and N. Pippenger. Finding the median. Journal of Computer

and System Sciences, 13(2):184–199, 1976.

[303] Alexander Schrijver. Theory of Linear and Integer Programming. John Wiley & Sons,

1986.

[304] Alexander Schrijver. Paths and ﬂows—A historical survey. CWI Quarterly, 6(3):169–183,

1993.

[305] Robert Sedgewick. Implementing quicksort programs. Communications of the ACM,

21(10):847–857, 1978.

[306] Robert Sedgewick. Algorithms. Addison-Wesley, second edition, 1988.

[307] Robert Sedgewick and Philippe Flajolet. An Introduction to the Analysis of Algorithms.

Addison-Wesley, 1996.

[308] Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs.

Journal of Computer and System Sciences, 51(3):400–403, 1995.

[309] Raimund Seidel and C. R. Aragon. Randomized search trees. Algorithmica, 16(4–5):464–

497, 1996.

[310] Jo˜ao Setubal and Jo˜ao Meidanis. Introduction to Computational Molecular Biology. PWS

Publishing Company, 1997.

[311] Clifford A. Shaffer. A Practical Introduction to Data Structures and Algorithm Analysis.

Prentice Hall, second edition, 2001.

[312] Jeffrey Shallit. Origins of the analysis of the Euclidean algorithm. Historia Mathematica,

21(4):401–419, 1994.

[313] Michael I. Shamos and Dan Hoey. Geometric intersection problems. In Proceedings of the

17th Annual Symposium on Foundations of Computer Science, pages 208–215, 1976.

[314] M. Sharir. A strong-connectivity algorithm and its applications in data ﬂow analysis. Com-

puters and Mathematics with Applications, 7(1):67–72, 1981.

[315] David B. Shmoys. Computing near-optimal solutions to combinatorial optimization prob-

lems. In William Cook, L´aszl´o Lov´asz, and Paul Seymour, editors, Combinatorial Opti-

mization, volume 20 of DIMACS Series in Discrete Mathematics and Theoretical Computer

Science. American Mathematical Society, 1995.

[316] Avi Shoshan and Uri Zwick. All pairs shortest paths in undirected graphs with integer

weights. In Proceedings of the 40th Annual Symposium on Foundations of Computer Sci-

ence, pages 605–614, 1999.

[317] Michael Sipser. Introduction to the Theory of Computation. Thomson Course Technology,

second edition, 2006.

[318] Steven S. Skiena. The Algorithm Design Manual. Springer, second edition, 1998.

[319] Daniel D. Sleator and Robert E. Tarjan. A data structure for dynamic trees. Journal of

Computer and System Sciences, 26(3):362–391, 1983.

1248 Bibliography

[320] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. Journal of the

ACM, 32(3):652–686, 1985.

[321] Joel Spencer. Ten Lectures on the Probabilistic Method, volume 64 of CBMS-NSF Regional

Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics,

1993.

[322] Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the sim-

plex algorithm usually takes polynomial time. Journal of the ACM, 51(3):385–463, 2004.

[323] Gilbert Strang. Introduction to Applied Mathematics. Wellesley-Cambridge Press, 1986.

[324] Gilbert Strang. Linear Algebra and Its Applications. Thomson Brooks/Cole, fourth edition,

2006.

[325] Volker Strassen. Gaussian elimination is not optimal. Numerische Mathematik, 14(3):354–

356, 1969.

[326] T. G. Szymanski. A special case of the maximal common subsequence problem. Technical

Report TR-170, Computer Science Laboratory, Princeton University, 1975.

[327] Robert E. Tarjan. Depth ﬁrst search and linear graph algorithms. SIAM Journal on Com-

puting, 1(2):146–160, 1972.

[328] Robert E. Tarjan. Efﬁciency of a good but not linear set union algorithm. Journal of the

ACM, 22(2):215–225, 1975.

[329] Robert E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint

sets. Journal of Computer and System Sciences, 18(2):110–127, 1979.

[330] Robert E. Tarjan. Data Structures and Network Algorithms. Society for Industrial and

Applied Mathematics, 1983.

[331] Robert E. Tarjan. Amortized computational complexity. SIAM Journal on Algebraic and

Discrete Methods, 6(2):306–318, 1985.

[332] Robert E. Tarjan. Class notes: Disjoint set union. COS 423, Princeton University, 1999.

[333] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. Jour-

nal of the ACM, 31(2):245–281, 1984.

[334] George B. Thomas, Jr., Maurice D. Weir, Joel Hass, and Frank R. Giordano. Thomas’

Calculus. Addison-Wesley, eleventh edition, 2005.

[335] Mikkel Thorup. Faster deterministic sorting and priority queues in linear space. In Pro-

ceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms, pages 550–555, 1998.

[336] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in

linear time. Journal of the ACM, 46(3):362–394, 1999.

[337] Mikkel Thorup. On RAM priority queues. SIAM Journal on Computing, 30(1):86–109,

2000.

[338] Richard Tolimieri, Myoung An, and Chao Lu. Mathematics of Multidimensional Fourier

Transform Algorithms. Springer, second edition, 1997.

[339] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In Proceedings

of the 16th Annual Symposium on Foundations of Computer Science, pages 75–84, 1975.

Bibliography 1249

[340] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear

space. Information Processing Letters, 6(3):80–82, 1977.

[341] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efﬁcient

priority queue. Mathematical Systems Theory, 10(1):99–127, 1976.

[342] Jan van Leeuwen, editor. Handbook of Theoretical Computer Science, Volume A: Algo-

rithms and Complexity. Elsevier Science Publishers and the MIT Press, 1990.

[343] Charles Van Loan. Computational Frameworks for the Fast Fourier Transform. Society for

Industrial and Applied Mathematics, 1992.

[344] Robert J. Vanderbei. Linear Programming: Foundations and Extensions. Kluwer Academic

Publishers, 1996.

[345] Vijay V. Vazirani. Approximation Algorithms. Springer, 2001.

[346] Rakesh M. Verma. General techniques for analyzing recursive algorithms with applications.

SIAM Journal on Computing, 26(2):568–581, 1997.

[347] Hao Wang and Bill Lin. Pipelined van Emde Boas tree: Algorithms, analysis, and ap-

plications. In 26th IEEE International Conference on Computer Communications, pages

2471–2475, 2007.

[348] Antony F. Ware. Fast approximate Fourier transforms for irregularly spaced data. SIAM

Review, 40(4):838–856, 1998.

[349] Stephen Warshall. A theorem on boolean matrices. Journal of the ACM, 9(1):11–12, 1962.

[350] Michael S. Waterman. Introduction to Computational Biology, Maps, Sequences and

Genomes. Chapman & Hall, 1995.

[351] Mark Allen Weiss. Data Structures and Problem Solving Using C++. Addison-Wesley,

second edition, 2000.

[352] Mark Allen Weiss. Data Structures and Problem Solving Using Java. Addison-Wesley,

third edition, 2006.

[353] Mark Allen Weiss. Data Structures and Algorithm Analysis in C++. Addison-Wesley, third

edition, 2007.

[354] Mark Allen Weiss. Data Structures and Algorithm Analysis in Java. Addison-Wesley,

second edition, 2007.

[355] Hassler Whitney. On the abstract properties of linear dependence. American Journal of

Mathematics, 57(3):509–533, 1935.

[356] Herbert S. Wilf. Algorithms and Complexity. A K Peters, second edition, 2002.

[357] J. W. J. Williams. Algorithm 232 (HEAPSORT). Communications of the ACM, 7(6):347–

348, 1964.

[358] Shmuel Winograd. On the algebraic complexity of functions. In Actes du Congr`es Interna-

tional des Math´ematiciens, volume 3, pages 283–288, 1970.

[359] Andrew C.-C. Yao. A lower bound to ﬁnding convex hulls. Journal of the ACM, 28(4):780–

787, 1981.

[360] Chee Yap. A real elementary approach to the master recurrence and generalizations. Un-

published manuscript. Available at http://cs.nyu.edu/yap/papers/, July 2008.

1250 Bibliography

[361] Yinyu Ye. Interior Point Algorithms: Theory and Analysis. John Wiley & Sons, 1997.

[362] Daniel Zwillinger, editor. CRC Standard Mathematical Tables and Formulae. Chapman &

Hall/CRC Press, 31st edition, 2003.

Index

This index uses the following conventions. Numbers are alphabetized as if spelled

out; for example, “2-3-4 tree” is indexed as if it were “two-three-four tree.” When

an entry refers to a place other than the main text, the page number is followed by

a tag: ex. for exercise, pr. for problem, ﬁg. for ﬁgure, and n. for footnote. A tagged

page number often indicates the ﬁrst page of an exercise or problem, which is not

necessarily the page on which the reference actually appears.

˛.n/, 574

(golden ratio), 59, 108 pr.

y (conjugate of the golden ratio), 59

.n/ (Euler’s phi function), 943

.n/-approximation algorithm, 1106, 1123

o-notation, 50–51, 64

O-notation, 45 ﬁg., 47–48, 64

O

0

-notation, 62 pr.

e O-notation, 62 pr.

!-notation, 51

-notation, 45 ﬁg., 48–49, 64

1

-notation, 62 pr.

e -notation, 62 pr.

‚-notation, 44–47, 45 ﬁg., 64

e ‚-notation, 62 pr.

f g (set), 1158

2 (set member), 1158

62 (not a set member), 1158

;

(empty language), 1058

(empty set), 1158

(subset), 1159

(proper subset), 1159

W (such that), 1159

\ (set intersection), 1159

[ (set union), 1159

(set difference), 1159

j j

(ﬂow value), 710

(length of a string), 986

(set cardinality), 1161

(Cartesian product), 1162

(cross product), 1016

h i

(sequence), 1166

(standard encoding), 1057   
 n

k

(choose), 1185

k k (euclidean norm), 1222

Š (factorial), 57

d e (ceiling), 54

b c (ﬂoor), 54

#

p

(lower square root), 546

"

p

(upper square root), 546 P

(sum), 1145 Q

(product), 1148

! (adjacency relation), 1169

(reachability relation), 1170

^ (AND), 697, 1071

: (NOT), 1071

\_ (OR), 697, 1071

˚ (group operator), 939

˝ (convolution operator), 901

1252 Index

(closure operator), 1058

j (divides relation), 927

− (does-not-divide relation), 927

(equivalent modulo n), 54, 1165 ex.

6 (not equivalent modulo n), 54

Œan (equivalence class modulo n), 928

Cn (addition modulo n), 940

n (multiplication modulo n), 940

.

a

p

/ (Legendre symbol), 982 pr.

" (empty string), 986, 1058

(preﬁx relation), 986

(sufﬁx relation), 986

< x (above relation), 1022

// (comment symbol), 21

(much-greater-than relation), 574

(much-less-than relation), 783

P (polynomial-time reducibility relation),

1067, 1077 ex.

AA-tree, 338

abelian group, 940

ABOVE, 1024

above relation ( < x), 1022

absent child, 1178

absolutely convergent series, 1146

absorption laws for sets, 1160

abstract problem, 1054

acceptable pair of integers, 972

acceptance

by an algorithm, 1058

by a ﬁnite automaton, 996

accepting state, 995

accounting method, 456–459

for binary counters, 458

for dynamic tables, 465–466

for stack operations, 457–458, 458 ex.

Ackermann’s function, 585

activity-selection problem, 415–422, 450

acyclic graph, 1170

relation to matroids, 448 pr.

add instruction, 23

addition

of binary integers, 22 ex.

of matrices, 1220

modulo n (Cn), 940

of polynomials, 898

additive group modulo n, 940

addressing, open, see open-address hash table

ADD-SUBARRAY, 805 pr.

adjacency-list representation, 590

replaced by a hash table, 593 ex.

adjacency-matrix representation, 591

adjacency relation (!), 1169

adjacent vertices, 1169

admissible edge, 749

admissible network, 749–750

adversary, 190

aggregate analysis, 452–456

for binary counters, 454–455

for breadth-ﬁrst search, 597

for depth-ﬁrst search, 606

for Dijkstra’s algorithm, 661

for disjoint-set data structures, 566–567,

568 ex.

for dynamic tables, 465

for Fibonacci heaps, 518, 522 ex.

for Graham’s scan, 1036

for the Knuth-Morris-Pratt algorithm, 1006

for Prim’s algorithm, 636

for rod-cutting, 367

for shortest paths in a dag, 655

for stack operations, 452–454

aggregate ﬂow, 863

Akra-Bazzi method for solving a recurrence,

112–113

algorithm, 5

correctness of, 6

origin of word, 42

running time of, 25

as a technology, 13

Alice, 959

ALLOCATE-NODE, 492

ALLOCATE-OBJECT, 244

allocation of objects, 243–244

all-pairs shortest paths, 644, 684–707

in dynamic graphs, 707

in -dense graphs, 706 pr.

Floyd-Warshall algorithm for, 693–697, 706

Johnson’s algorithm for, 700–706

by matrix multiplication, 686–693, 706–707

by repeated squaring, 689–691

alphabet, 995, 1057

˛.n/, 574

amortized analysis, 451–478

accounting method of, 456–459

aggregate analysis, 367, 452–456

Index 1253

for bit-reversal permutation, 472 pr.

for breadth-ﬁrst search, 597

for depth-ﬁrst search, 606

for Dijkstra’s algorithm, 661

for disjoint-set data structures, 566–567,

568 ex., 572 ex., 575–581, 581–582 ex.

for dynamic tables, 463–471

for Fibonacci heaps, 509–512, 517–518,

520–522, 522 ex.

for the generic push-relabel algorithm, 746

for Graham’s scan, 1036

for the Knuth-Morris-Pratt algorithm, 1006

for making binary search dynamic, 473 pr.

potential method of, 459–463

for restructuring red-black trees, 474 pr.

for self-organizing lists with move-to-front,

476 pr.

for shortest paths in a dag, 655

for stacks on secondary storage, 502 pr.

for weight-balanced trees, 473 pr.

amortized cost

in the accounting method, 456

in aggregate analysis, 452

in the potential method, 459

ancestor, 1176

least common, 584 pr.

AND function (^), 697, 1071

AND gate, 1070

and, in pseudocode, 22

antiparallel edges, 711–712

antisymmetric relation, 1164

ANY-SEGMENTS-INTERSECT, 1025

approximation

by least squares, 835–839

of summation by integrals, 1154–1156

approximation algorithm, 10, 1105–1140

for bin packing, 1134 pr.

for MAX-CNF satisﬁability, 1127 ex.

for maximum clique, 1111 ex., 1134 pr.

for maximum matching, 1135 pr.

for maximum spanning tree, 1137 pr.

for maximum-weight cut, 1127 ex.

for MAX-3-CNF satisﬁability, 1123–1124,

1139

for minimum-weight vertex cover,

1124–1127, 1139

for parallel machine scheduling, 1136 pr.

randomized, 1123

for set cover, 1117–1122, 1139

for subset sum, 1128–1134, 1139

for traveling-salesman problem, 1111–1117,

1139

for vertex cover, 1108–1111, 1139

for weighted set cover, 1135 pr.

for 0-1 knapsack problem, 1137 pr., 1139

approximation error, 836

approximation ratio, 1106, 1123

approximation scheme, 1107

APPROX-MIN-WEIGHT-VC, 1126

APPROX-SUBSET-SUM, 1131

APPROX-TSP-TOUR, 1112

APPROX-VERTEX-COVER, 1109

arbitrage, 679 pr.

arc, see edge

argument of a function, 1166–1167

arithmetic instructions, 23

arithmetic, modular, 54, 939–946

arithmetic series, 1146

arithmetic with inﬁnities, 650

arm, 485

array, 21

Monge, 110 pr.

passing as a parameter, 21

articulation point, 621 pr.

assignment

multiple, 21

satisfying, 1072, 1079

truth, 1072, 1079

associative laws for sets, 1160

associative operation, 939

asymptotically larger, 52

asymptotically nonnegative, 45

asymptotically positive, 45

asymptotically smaller, 52

asymptotically tight bound, 45

asymptotic efﬁciency, 43

asymptotic lower bound, 48

asymptotic notation, 43–53, 62 pr.

and graph algorithms, 588

and linearity of summations, 1146

asymptotic upper bound, 47

attribute of an object, 21

augmentation of a ﬂow, 716

augmenting data structures, 339–355

augmenting path, 719–720, 763 pr.

authentication, 284 pr., 960–961, 964

1254 Index

automaton

ﬁnite, 995

string-matching, 996–1002

auxiliary hash function, 272

auxiliary linear program, 886

average-case running time, 28, 116

AVL-INSERT, 333 pr.

AVL tree, 333 pr., 337

axioms, for probability, 1190

babyface, 602 ex.

back edge, 609, 613

back substitution, 817

BAD-SET-COVER-INSTANCE, 1122 ex.

BALANCE, 333 pr.

balanced search tree

AA-trees, 338

AVL trees, 333 pr., 337

B-trees, 484–504

k-neighbor trees, 338

red-black trees, 308–338

scapegoat trees, 338

splay trees, 338, 482

treaps, 333 pr., 338

2-3-4 trees, 489, 503 pr.

2-3 trees, 337, 504

weight-balanced trees, 338, 473 pr.

balls and bins, 133–134, 1215 pr.

base-a pseudoprime, 967

base case, 65, 84

base, in DNA, 391

basic feasible solution, 866

basic solution, 866

basic variable, 855

basis function, 835

Bayes’s theorem, 1194

BELLMAN-FORD, 651

Bellman-Ford algorithm, 651–655, 682

for all-pairs shortest paths, 684

in Johnson’s algorithm, 702–704

and objective functions, 670 ex.

to solve systems of difference constraints,

668

Yen’s improvement to, 678 pr.

BELOW, 1024

Bernoulli trial, 1201

and balls and bins, 133–134

and streaks, 135–139

best-case running time, 29 ex., 49

BFS, 595

BIASED-RANDOM, 117 ex.

biconnected component, 621 pr.

big-oh notation, 45 ﬁg., 47–48, 64

big-omega notation, 45 ﬁg., 48–49, 64

bijective function, 1167

binary character code, 428

binary counter

analyzed by accounting method, 458

analyzed by aggregate analysis, 454–455

analyzed by potential method, 461–462

bit-reversed, 472 pr.

binary entropy function, 1187

binary gcd algorithm, 981 pr.

binary heap, see heap

binary relation, 1163

binary search, 39 ex.

with fast insertion, 473 pr.

in insertion sort, 39 ex.

in multithreaded merging, 799–800

in searching B-trees, 499 ex.

BINARY-SEARCH, 799

binary search tree, 286–307

AA-trees, 338

AVL trees, 333 pr., 337

deletion from, 295–298, 299 ex.

with equal keys, 303 pr.

insertion into, 294–295

k-neighbor trees, 338

maximum key of, 291

minimum key of, 291

optimal, 397–404, 413

predecessor in, 291–292

querying, 289–294

randomly built, 299–303, 304 pr.

right-converting of, 314 ex.

scapegoat trees, 338

searching, 289–291

for sorting, 299 ex.

splay trees, 338

successor in, 291–292

and treaps, 333 pr.

weight-balanced trees, 338

see also red-black tree

binary-search-tree property, 287

in treaps, 333 pr.

vs. min-heap property, 289 ex.

Index 1255

binary tree, 1177

full, 1178

number of different ones, 306 pr.

representation of, 246

superimposed upon a bit vector, 533–534

see also binary search tree

binomial coefﬁcient, 1186–1187

binomial distribution, 1203–1206

and balls and bins, 133

maximum value of, 1207 ex.

tails of, 1208–1215

binomial expansion, 1186

binomial heap, 527 pr.

binomial tree, 527 pr.

bin packing, 1134 pr.

bipartite graph, 1172

corresponding ﬂow network of, 732

d-regular, 736 ex.

and hypergraphs, 1173 ex.

bipartite matching, 530, 732–736, 747 ex., 766

Hopcroft-Karp algorithm for, 763 pr.

birthday paradox, 130–133, 142 ex.

bisection of a tree, 1181 pr.

bitonic euclidean traveling-salesman problem,

405 pr.

bitonic sequence, 682 pr.

bitonic tour, 405 pr.

bit operation, 927

in Euclid’s algorithm, 981 pr.

bit-reversal permutation, 472 pr., 918

BIT-REVERSE-COPY, 918

bit-reversed binary counter, 472 pr.

BIT-REVERSED-INCREMENT, 472 pr.

bit vector, 255 ex., 532–536

black-height, 309

black vertex, 594, 603

blocking ﬂow, 765

block structure in pseudocode, 20

Bob, 959

Boole’s inequality, 1195 ex.

boolean combinational circuit, 1071

boolean combinational element, 1070

boolean connective, 1079

boolean formula, 1049, 1066 ex., 1079,

1086 ex.

boolean function, 1187 ex.

boolean matrix multiplication, 832 ex.

Bor˙uvka’s algorithm, 641

bottleneck spanning tree, 640 pr.

bottleneck traveling-salesman problem,

1117 ex.

bottom of a stack, 233

BOTTOM-UP-CUT-ROD, 366

bottom-up method, for dynamic programming,

365

bound

asymptotically tight, 45

asymptotic lower, 48

asymptotic upper, 47

on binomial coefﬁcients, 1186–1187

on binomial distributions, 1206

polylogarithmic, 57

on the tails of a binomial distribution,

1208–1215

see also lower bounds

boundary condition, in a recurrence, 67, 84

boundary of a polygon, 1020 ex.

bounding a summation, 1149–1156

box, nesting, 678 pr.

B

C

-tree, 488

branching factor, in B-trees, 487

branch instructions, 23

breadth-ﬁrst search, 594–602, 623

in maximum ﬂow, 727–730, 766

and shortest paths, 597–600, 644

similarity to Dijkstra’s algorithm, 662,

663 ex.

breadth-ﬁrst tree, 594, 600

bridge, 621 pr.

B

-tree, 489 n.

B-tree, 484–504

compared with red-black trees, 484, 490

creating, 492

deletion from, 499–502

full node in, 489

height of, 489–490

insertion into, 493–497

minimum degree of, 489

minimum key of, 497 ex.

properties of, 488–491

searching, 491–492

splitting a node in, 493–495

2-3-4 trees, 489

B-TREE-CREATE, 492

B-TREE-DELETE, 499

B-TREE-INSERT, 495

1256 Index

B-TREE-INSERT-NONFULL, 496

B-TREE-SEARCH, 492, 499 ex.

B-TREE-SPLIT-CHILD, 494

BUBBLESORT, 40 pr.

bucket, 200

bucket sort, 200–204

BUCKET-SORT, 201

BUILD-MAX-HEAP, 157

BUILD-MAX-HEAP

0

, 167 pr.

BUILD-MIN-HEAP, 159

butterﬂy operation, 915

by, in pseudocode, 21

cache, 24, 449 pr.

cache hit, 449 pr.

cache miss, 449 pr.

cache obliviousness, 504

caching, off-line, 449 pr.

call

in a multithreaded computation, 776

of a subroutine, 23, 25 n.

by value, 21

call edge, 778

cancellation lemma, 907

cancellation of ﬂow, 717

canonical form for task scheduling, 444

capacity

of a cut, 721

of an edge, 709

residual, 716, 719

of a vertex, 714 ex.

capacity constraint, 709–710

cardinality of a set (j j), 1161

Carmichael number, 968, 975 ex.

Cartesian product ( ), 1162

Cartesian sum, 906 ex.

cascading cut, 520

CASCADING-CUT, 519

Catalan numbers, 306 pr., 372

ceiling function (d e), 54

in master theorem, 103–106

ceiling instruction, 23

certain event, 1190

certiﬁcate

in a cryptosystem, 964

for veriﬁcation algorithms, 1063

CHAINED-HASH-DELETE, 258

CHAINED-HASH-INSERT, 258

CHAINED-HASH-SEARCH, 258

chaining, 257–260, 283 pr.

chain of a convex hull, 1038

changing a key, in a Fibonacci heap, 529 pr.

changing variables, in the substitution method,

86–87

character code, 428

chess-playing program, 790–791

child

in a binary tree, 1178

in a multithreaded computation, 776

in a rooted tree, 1176

child list in a Fibonacci heap, 507

Chinese remainder theorem, 950–954, 983

chip multiprocessor, 772

chirp transform, 914 ex.

choose

n

k

, 1185

chord, 345 ex.

Cilk, 774, 812

Cilk++, 774, 812

ciphertext, 960

circuit

boolean combinational, 1071

depth of, 919

for fast Fourier transform, 919–920

CIRCUIT-SAT, 1072

circuit satisﬁability, 1070–1077

circular, doubly linked list with a sentinel, 239

circular linked list, 236

see also linked list

class

complexity, 1059

equivalence, 1164

classiﬁcation of edges

in breadth-ﬁrst search, 621 pr.

in depth-ﬁrst search, 609–610, 611 ex.

in a multithreaded dag, 778–779

clause, 1081–1082

clean area, 208 pr.

clique, 1086–1089, 1105

approximation algorithm for, 1111 ex.,

1134 pr.

CLIQUE, 1087

closed interval, 348

closed semiring, 707

closest pair, ﬁnding, 1039–1044, 1047

closest-point heuristic, 1117 ex.

Index 1257

closure

group property, 939

of a language, 1058

operator (

), 1058

transitive, see transitive closure

cluster

in a bit vector with a superimposed tree of

constant height, 534

for parallel computing, 772

in proto van Emde Boas structures, 538

in van Emde Boas trees, 546

clustering, 272

CNF (conjunctive normal form), 1049, 1082

CNF satisﬁability, 1127 ex.

coarsening leaves of recursion

in merge sort, 39 pr.

when recursively spawning, 787

code, 428–429

Huffman, 428–437, 450

codeword, 429

codomain, 1166

coefﬁcient

binomial, 1186

of a polynomial, 55, 898

in slack form, 856

coefﬁcient representation, 900

and fast multiplication, 903–905

cofactor, 1224

coin changing, 446 pr.

colinearity, 1016

collision, 257

resolution by chaining, 257–260

resolution by open addressing, 269–277

collision-resistant hash function, 964

coloring, 1103 pr., 1180 pr.

color, of a red-black-tree node, 308

column-major order, 208 pr.

column rank, 1223

columnsort, 208 pr.

column vector, 1218

combination, 1185

combinational circuit, 1071

combinational element, 1070

combine step, in divide-and-conquer, 30, 65

comment, in pseudocode (//), 21

commodity, 862

common divisor, 929

greatest, see greatest common divisor

common multiple, 939 ex.

common subexpression, 915

common subsequence, 7, 391

longest, 7, 390–397, 413

commutative laws for sets, 1159

commutative operation, 940

COMPACTIFY-LIST, 245 ex.

compact list, 250 pr.

COMPACT-LIST-SEARCH, 250 pr.

COMPACT-LIST-SEARCH

0

, 251 pr.

comparable line segments, 1022

COMPARE-EXCHANGE, 208 pr.

compare-exchange operation, 208 pr.

comparison sort, 191

and binary search trees, 289 ex.

randomized, 205 pr.

and selection, 222

compatible activities, 415

compatible matrices, 371, 1221

competitive analysis, 476 pr.

complement

of an event, 1190

of a graph, 1090

of a language, 1058

Schur, 820, 834

of a set, 1160

complementary slackness, 894 pr.

complete graph, 1172

complete k-ary tree, 1179

see also heap

completeness of a language, 1077 ex.

complete step, 782

completion time, 447 pr., 1136 pr.

complexity class, 1059

co-NP, 1064

NP, 1049, 1064

NPC, 1050, 1069

P, 1049, 1055

complexity measure, 1059

complex numbers

inverting matrices of, 832 ex.

multiplication of, 83 ex.

complex root of unity, 906

interpolation at, 912–913

component

biconnected, 621 pr.

connected, 1170

strongly connected, 1170

1258 Index

component graph, 617

composite number, 928

witness to, 968

composition, of multithreaded computations,

784 ﬁg.

computational depth, 812

computational geometry, 1014–1047

computational problem, 5–6

computation dag, 777

computation, multithreaded, 777

COMPUTE-PREFIX-FUNCTION, 1006

COMPUTE-TRANSITION-FUNCTION, 1001

concatenation

of languages, 1058

of strings, 986

concrete problem, 1055

concurrency keywords, 774, 776, 785

concurrency platform, 773

conditional branch instruction, 23

conditional independence, 1195 ex.

conditional probability, 1192, 1194

conﬁguration, 1074

conjugate of the golden ratio (y), 59

conjugate transpose, 832 ex.

conjunctive normal form, 1049, 1082

connected component, 1170

identiﬁed using depth-ﬁrst search, 612 ex.

identiﬁed using disjoint-set data structures,

562–564

CONNECTED-COMPONENTS, 563

connected graph, 1170

connective, 1079

co-NP (complexity class), 1064

conquer step, in divide-and-conquer, 30, 65

conservation of ﬂow, 709–710

consistency

of literals, 1088

sequential, 779, 812

CONSOLIDATE, 516

consolidating a Fibonacci-heap root list,

513–517

constraint, 851

difference, 665

equality, 670 ex., 852–853

inequality, 852–853

linear, 846

nonnegativity, 851, 853

tight, 865

violation of, 865

constraint graph, 666–668

contain, in a path, 1170

continuation edge, 778

continuous uniform probability distribution,

1192

contraction

of a dynamic table, 467–471

of a matroid, 442

of an undirected graph by an edge, 1172

control instructions, 23

convergence property, 650, 672–673

convergent series, 1146

converting binary to decimal, 933 ex.

convex combination of points, 1015

convex function, 1199

convex hull, 8, 1029–1039, 1046 pr.

convex layers, 1044 pr.

convex polygon, 1020 ex.

convex set, 714 ex.

convolution (˝), 901

convolution theorem, 913

copy instruction, 23

correctness of an algorithm, 6

corresponding ﬂow network for bipartite

matching, 732

countably inﬁnite set, 1161

counter, see binary counter

counting, 1183–1189

probabilistic, 143 pr.

counting sort, 194–197

in radix sort, 198

COUNTING-SORT, 195

coupon collector’s problem, 134

cover

path, 761 pr.

by a subset, 1118

vertex, 1089, 1108, 1124–1127, 1139

covertical, 1024

CREATE-NEW-RS-VEB-TREE, 557 pr.

credit, 456

critical edge, 729

critical path

of a dag, 657

of a multithreaded computation, 779

cross a cut, 626

cross edge, 609

cross product ( ), 1016

Index 1259

cryptosystem, 958–965, 983

cubic spline, 840 pr.

currency exchange, 390 ex., 679 pr.

curve ﬁtting, 835–839

cut

capacity of, 721

cascading, 520

of a ﬂow network, 720–724

minimum, 721, 731 ex.

net ﬂow across, 720

of an undirected graph, 626

weight of, 1127 ex.

CUT, 519

CUT-ROD, 363

cutting, in a Fibonacci heap, 519

cycle of a graph, 1170

hamiltonian, 1049, 1061

minimum mean-weight, 680 pr.

negative-weight, see negative-weight cycle

and shortest paths, 646–647

cyclic group, 955

cyclic rotation, 1012 ex.

cycling, of simplex algorithm, 875

dag, see directed acyclic graph

DAG-SHORTEST-PATHS, 655

d-ary heap, 167 pr.

in shortest-paths algorithms, 706 pr.

data-movement instructions, 23

data-parallel model, 811

data structure, 9, 229–355, 481–585

AA-trees, 338

augmentation of, 339–355

AVL trees, 333 pr., 337

binary search trees, 286–307

binomial heaps, 527 pr.

bit vectors, 255 ex., 532–536

B-trees, 484–504

deques, 236 ex.

dictionaries, 229

direct-address tables, 254–255

for disjoint sets, 561–585

for dynamic graphs, 483

dynamic sets, 229–231

dynamic trees, 482

exponential search trees, 212, 483

Fibonacci heaps, 505–530

fusion trees, 212, 483

hash tables, 256–261

heaps, 151–169

interval trees, 348–354

k-neighbor trees, 338

linked lists, 236–241

mergeable heap, 505

order-statistic trees, 339–345

persistent, 331 pr., 482

potential of, 459

priority queues, 162–166

proto van Emde Boas structures, 538–545

queues, 232, 234–235

radix trees, 304 pr.

red-black trees, 308–338

relaxed heaps, 530

rooted trees, 246–249

scapegoat trees, 338

on secondary storage, 484–487

skip lists, 338

splay trees, 338, 482

stacks, 232–233

treaps, 333 pr., 338

2-3-4 heaps, 529 pr.

2-3-4 trees, 489, 503 pr.

2-3 trees, 337, 504

van Emde Boas trees, 531–560

weight-balanced trees, 338

data type, 23

deadline, 444

deallocation of objects, 243–244

decision by an algorithm, 1058–1059

decision problem, 1051, 1054

and optimization problems, 1051

decision tree, 192–193

DECREASE-KEY, 162, 505

decreasing a key

in Fibonacci heaps, 519–522

in 2-3-4 heaps, 529 pr.

DECREMENT, 456 ex.

degeneracy, 874

degree

of a binomial-tree root, 527 pr.

maximum, of a Fibonacci heap, 509,

523–526

minimum, of a B-tree, 489

of a node, 1177

of a polynomial, 55, 898

of a vertex, 1169

1260 Index

degree-bound, 898

DELETE, 230, 505

DELETE-LARGER-HALF, 463 ex.

deletion

from binary search trees, 295–298, 299 ex.

from a bit vector with a superimposed binary

tree, 534

from a bit vector with a superimposed tree of

constant height, 535

from B-trees, 499–502

from chained hash tables, 258

from direct-address tables, 254

from dynamic tables, 467–471

from Fibonacci heaps, 522, 526 pr.

from heaps, 166 ex.

from interval trees, 349

from linked lists, 238

from open-address hash tables, 271

from order-statistic trees, 343–344

from proto van Emde Boas structures, 544

from queues, 234

from red-black trees, 323–330

from stacks, 232

from sweep-line statuses, 1024

from 2-3-4 heaps, 529 pr.

from van Emde Boas trees, 554–556

DeMorgan’s laws

for propositional logic, 1083

for sets, 1160, 1162 ex.

dense graph, 589

-dense, 706 pr.

density

of prime numbers, 965–966

of a rod, 370 ex.

dependence

and indicator random variables, 119

linear, 1223

see also independence

depth

average, of a node in a randomly built binary

search tree, 304 pr.

of a circuit, 919

of a node in a rooted tree, 1177

of quicksort recursion tree, 178 ex.

of a stack, 188 pr.

depth-determination problem, 583 pr.

depth-ﬁrst forest, 603

depth-ﬁrst search, 603–612, 623

in ﬁnding articulation points, bridges, and

biconnected components, 621 pr.

in ﬁnding strongly connected components,

615–621, 623

in topological sorting, 612–615

depth-ﬁrst tree, 603

deque, 236 ex.

DEQUEUE, 235

derivative of a series, 1147

descendant, 1176

destination vertex, 644

det, see determinant

determinacy race, 788

determinant, 1224–1225

and matrix multiplication, 832 ex.

deterministic algorithm, 123

multithreaded, 787

DETERMINISTIC-SEARCH, 143 pr.

DFS, 604

DFS-VISIT, 604

DFT (discrete Fourier transform), 9, 909

diagonal matrix, 1218

LUP decomposition of, 827 ex.

diameter of a tree, 602 ex.

dictionary, 229

difference constraints, 664–670

difference equation, see recurrence

difference of sets (), 1159

symmetric, 763 pr.

differentiation of a series, 1147

digital signature, 960

digraph, see directed graph

DIJKSTRA, 658

Dijkstra’s algorithm, 658–664, 682

for all-pairs shortest paths, 684, 704

implemented with a Fibonacci heap, 662

implemented with a min-heap, 662

with integer edge weights, 664 ex.

in Johnson’s algorithm, 702

similarity to breadth-ﬁrst search, 662,

663 ex.

similarity to Prim’s algorithm, 634, 662

DIRECT-ADDRESS-DELETE, 254

direct addressing, 254–255, 532–536

DIRECT-ADDRESS-INSERT, 254

DIRECT-ADDRESS-SEARCH, 254

direct-address table, 254–255

directed acyclic graph (dag), 1172

Index 1261

and back edges, 613

and component graphs, 617

and hamiltonian paths, 1066 ex.

longest simple path in, 404 pr.

for representing a multithreaded

computation, 777

single-source shortest-paths algorithm for,

655–658

topological sort of, 612–615, 623

directed graph, 1168

all-pairs shortest paths in, 684–707

constraint graph, 666

Euler tour of, 623 pr., 1048

hamiltonian cycle of, 1049

and longest paths, 1048

path cover of, 761 pr.

PERT chart, 657, 657 ex.

semiconnected, 621 ex.

shortest path in, 643

single-source shortest paths in, 643–683

singly connected, 612 ex.

square of, 593 ex.

transitive closure of, 697

transpose of, 592 ex.

universal sink in, 593 ex.

see also directed acyclic graph, graph,

network

directed segment, 1015–1017

directed version of an undirected graph, 1172

DIRECTION, 1018

dirty area, 208 pr.

DISCHARGE, 751

discharge of an overﬂowing vertex, 751

discovered vertex, 594, 603

discovery time, in depth-ﬁrst search, 605

discrete Fourier transform, 9, 909

discrete logarithm, 955

discrete logarithm theorem, 955

discrete probability distribution, 1191

discrete random variable, 1196–1201

disjoint-set data structure, 561–585

analysis of, 575–581, 581 ex.

in connected components, 562–564

in depth determination, 583 pr.

disjoint-set-forest implementation of,

568–572

in Kruskal’s algorithm, 631

linear-time special case of, 585

linked-list implementation of, 564–568

in off-line least common ancestors, 584 pr.

in off-line minimum, 582 pr.

in task scheduling, 448 pr.

disjoint-set forest, 568–572

analysis of, 575–581, 581 ex.

rank properties of, 575, 581 ex.

see also disjoint-set data structure

disjoint sets, 1161

disjunctive normal form, 1083

disk, 1028 ex.

disk drive, 485–487

see also secondary storage

DISK-READ, 487

DISK-WRITE, 487

distance

edit, 406 pr.

euclidean, 1039

Lm, 1044 ex.

Manhattan, 225 pr., 1044 ex.

of a shortest path, 597

distributed memory, 772

distribution

binomial, 1203–1206

continuous uniform, 1192

discrete, 1191

geometric, 1202–1203

of inputs, 116, 122

of prime numbers, 965

probability, 1190

sparse-hulled, 1046 pr.

uniform, 1191

distributive laws for sets, 1160

divergent series, 1146

divide-and-conquer method, 30–35, 65

analysis of, 34–35

for binary search, 39 ex.

for conversion of binary to decimal, 933 ex.

for fast Fourier transform, 909–912

for ﬁnding the closest pair of points,

1040–1043

for ﬁnding the convex hull, 1030

for matrix inversion, 829–831

for matrix multiplication, 76–83, 792–797

for maximum-subarray problem, 68–75

for merge sort, 30–37, 797–805

for multiplication, 920 pr.

1262 Index

for multithreaded matrix multiplication,

792–797

for multithreaded merge sort, 797–805

for quicksort, 170–190

relation to dynamic programming, 359

for selection, 215–224

solving recurrences for, 83–106, 112–113

for Strassen’s algorithm, 79–83

divide instruction, 23

divides relation (j), 927

divide step, in divide-and-conquer, 30, 65

division method, 263, 268–269 ex.

division theorem, 928

divisor, 927–928

common, 929

see also greatest common divisor

DNA, 6–7, 390–391, 406 pr.

DNF (disjunctive normal form), 1083

does-not-divide relation ( − ), 927

domain, 1166

dominates relation, 1045 pr.

double hashing, 272–274, 277 ex.

doubly linked list, 236

see also linked list

downto, in pseudocode, 21

d-regular graph, 736 ex.

duality, 879–886, 895 pr.

weak, 880–881, 886 ex.

dual linear program, 879

dummy key, 397

dynamic graph, 562 n.

all-pairs shortest paths algorithms for, 707

data structures for, 483

minimum-spanning-tree algorithm for,

637 ex.

transitive closure of, 705 pr., 707

dynamic multithreaded algorithm, see

multithreaded algorithm

dynamic multithreading, 773

dynamic order statistics, 339–345

dynamic-programming method, 359–413

for activity selection, 421 ex.

for all-pairs shortest paths, 686–697

for bitonic euclidean traveling-salesman

problem, 405 pr.

bottom-up, 365

for breaking a string, 410 pr.

compared with greedy algorithms, 381,

390 ex., 418, 423–427

for edit distance, 406 pr.

elements of, 378–390

for Floyd-Warshall algorithm, 693–697

for inventory planning, 411 pr.

for longest common subsequence, 390–397

for longest palindrome subsequence, 405 pr.

for longest simple path in a weighted

directed acyclic graph, 404 pr.

for matrix-chain multiplication, 370–378

and memoization, 387–389

for optimal binary search trees, 397–404

optimal substructure in, 379–384

overlapping subproblems in, 384–386

for printing neatly, 405 pr.

reconstructing an optimal solution in, 387

relation to divide-and-conquer, 359

for rod-cutting, 360–370

for seam carving, 409 pr.

for signing free agents, 411 pr.

top-down with memoization, 365

for transitive closure, 697–699

for Viterbi algorithm, 408 pr.

for 0-1 knapsack problem, 427 ex.

dynamic set, 229–231

see also data structure

dynamic table, 463–471

analyzed by accounting method, 465–466

analyzed by aggregate analysis, 465

analyzed by potential method, 466–471

load factor of, 463

dynamic tree, 482

e, 55

E Œ (expected value), 1197

early-ﬁrst form, 444

early task, 444

edge, 1168

admissible, 749

antiparallel, 711–712

attributes of, 592

back, 609

bridge, 621 pr.

call, 778

capacity of, 709

classiﬁcation in breadth-ﬁrst search, 621 pr.

classiﬁcation in depth-ﬁrst search, 609–610

Index 1263

continuation, 778

critical, 729

cross, 609

forward, 609

inadmissible, 749

light, 626

negative-weight, 645–646

residual, 716

return, 779

safe, 626

saturated, 739

spawn, 778

tree, 601, 603, 609

weight of, 591

edge connectivity, 731 ex.

edge set, 1168

edit distance, 406 pr.

Edmonds-Karp algorithm, 727–730

elementary event, 1189

elementary insertion, 465

element of a set (2), 1158

ellipsoid algorithm, 850, 897

elliptic-curve factorization method, 984

elseif, in pseudocode, 20 n.

else, in pseudocode, 20

empty language (;), 1058

empty set (;), 1158

empty set laws, 1159

empty stack, 233

empty string ("), 986, 1058

empty tree, 1178

encoding of problem instances, 1055–1057

endpoint

of an interval, 348

of a line segment, 1015

ENQUEUE, 235

entering a vertex, 1169

entering variable, 867

entropy function, 1187

-dense graph, 706 pr.

-universal hash function, 269 ex.

equality

of functions, 1166

linear, 845

of sets, 1158

equality constraint, 670 ex., 852

and inequality constraints, 853

tight, 865

violation of, 865

equation

and asymptotic notation, 49–50

normal, 837

recurrence, see recurrence

equivalence class, 1164

modulo n (Œan), 928

equivalence, modular (), 54, 1165 ex.

equivalence relation, 1164

and modular equivalence, 1165 ex.

equivalent linear programs, 852

error, in pseudocode, 22

escape problem, 760 pr.

EUCLID, 935

Euclid’s algorithm, 933–939, 981 pr., 983

euclidean distance, 1039

euclidean norm (k k), 1222

Euler’s constant, 943

Euler’s phi function, 943

Euler’s theorem, 954, 975 ex.

Euler tour, 623 pr., 1048

and hamiltonian cycles, 1048

evaluation of a polynomial, 41 pr., 900, 905 ex.

derivatives of, 922 pr.

at multiple points, 923 pr.

event, 1190

event point, 1023

event-point schedule, 1023

EXACT-SUBSET-SUM, 1129

excess ﬂow, 736

exchange property, 437

exclusion and inclusion, 1163 ex.

execute a subroutine, 25 n.

expansion of a dynamic table, 464–467

expectation, see expected value

expected running time, 28, 117

expected value, 1197–1199

of a binomial distribution, 1204

of a geometric distribution, 1202

of an indicator random variable, 118

explored vertex, 605

exponential function, 55–56

exponential height, 300

exponential search tree, 212, 483

exponential series, 1147

exponentiation instruction, 24

exponentiation, modular, 956

EXTENDED-BOTTOM-UP-CUT-ROD, 369

1264 Index

EXTENDED-EUCLID, 937

EXTEND-SHORTEST-PATHS, 688

extension of a set, 438

exterior of a polygon, 1020 ex.

external node, 1176

external path length, 1180 ex.

extracting the maximum key

from d-ary heaps, 167 pr.

from max-heaps, 163

extracting the minimum key

from Fibonacci heaps, 512–518

from 2-3-4 heaps, 529 pr.

from Young tableaus, 167 pr.

EXTRACT-MAX, 162–163

EXTRACT-MIN, 162, 505

factor, 928

twiddle, 912

factorial function (Š), 57–58

factorization, 975–980, 984

unique, 931

failure, in a Bernoulli trial, 1201

fair coin, 1191

fan-out, 1071

Farkas’s lemma, 895 pr.

farthest-pair problem, 1030

FASTER-ALL-PAIRS-SHORTEST-PATHS, 691,

692 ex.

fast Fourier transform (FFT), 898–925

circuit for, 919–920

iterative implementation of, 915–918

multidimensional, 921 pr.

multithreaded algorithm for, 804 ex.

recursive implementation of, 909–912

using modular arithmetic, 923 pr.

feasibility problem, 665, 894 pr.

feasible linear program, 851

feasible region, 847

feasible solution, 665, 846, 851

Fermat’s theorem, 954

FFT, see fast Fourier transform

FFTW, 924

FIB, 775

FIB-HEAP-CHANGE-KEY, 529 pr.

FIB-HEAP-DECREASE-KEY, 519

FIB-HEAP-DELETE, 522

FIB-HEAP-EXTRACT-MIN, 513

FIB-HEAP-INSERT, 510

FIB-HEAP-LINK, 516

FIB-HEAP-PRUNE, 529 pr.

FIB-HEAP-UNION, 512

Fibonacci heap, 505–530

changing a key in, 529 pr.

compared with binary heaps, 506–507

creating, 510

decreasing a key in, 519–522

deletion from, 522, 526 pr.

in Dijkstra’s algorithm, 662

extracting the minimum key from, 512–518

insertion into, 510–511

in Johnson’s algorithm, 704

maximum degree of, 509, 523–526

minimum key of, 511

potential function for, 509

in Prim’s algorithm, 636

pruning, 529 pr.

running times of operations on, 506 ﬁg.

uniting, 511–512

Fibonacci numbers, 59–60, 108 pr., 523

computation of, 774–780, 981 pr.

FIFO (ﬁrst-in, ﬁrst-out), 232

see also queue

ﬁnal-state function, 996

ﬁnal strand, 779

FIND-DEPTH, 583 pr.

FIND-MAX-CROSSING-SUBARRAY, 71

FIND-MAXIMUM-SUBARRAY, 72

ﬁnd path, 569

FIND-SET, 562

disjoint-set-forest implementation of, 571,

585

linked-list implementation of, 564

ﬁnished vertex, 603

ﬁnishing time, in depth-ﬁrst search, 605

and strongly connected components, 618

ﬁnish time, in activity selection, 415

ﬁnite automaton, 995

for string matching, 996–1002

FINITE-AUTOMATON-MATCHER, 999

ﬁnite group, 940

ﬁnite sequence, 1166

ﬁnite set, 1161

ﬁrst-ﬁt heuristic, 1134 pr.

ﬁrst-in, ﬁrst-out, 232

see also queue

ﬁxed-length code, 429

Index 1265

ﬂoating-point data type, 23

ﬂoor function (b c), 54

in master theorem, 103–106

ﬂoor instruction, 23

ﬂow, 709–714

aggregate, 863

augmentation of, 716

blocking, 765

cancellation of, 717

excess, 736

integer-valued, 733

net, across a cut, 720

value of, 710

ﬂow conservation, 709–710

ﬂow network, 709–714

corresponding to a bipartite graph, 732

cut of, 720–724

with multiple sources and sinks, 712

FLOYD-WARSHALL, 695

FLOYD-WARSHALL

0

, 699 ex.

Floyd-Warshall algorithm, 693–697,

699–700 ex., 706

multithreaded, 797 ex.

FORD-FULKERSON, 724

Ford-Fulkerson method, 714–731, 765

FORD-FULKERSON-METHOD, 715

forest, 1172–1173

depth-ﬁrst, 603

disjoint-set, 568–572

for, in pseudocode, 20–21

and loop invariants, 19 n.

formal power series, 108 pr.

formula satisﬁability, 1079–1081, 1105

forward edge, 609

forward substitution, 816–817

Fourier transform, see discrete Fourier

transform, fast Fourier transform

fractional knapsack problem, 426, 428 ex.

free agent, 411 pr.

freeing of objects, 243–244

free list, 243

FREE-OBJECT, 244

free tree, 1172–1176

frequency domain, 898

full binary tree, 1178, 1180 ex.

relation to optimal code, 430

full node, 489

full rank, 1223

full walk of a tree, 1114

fully parenthesized matrix-chain product, 370

fully polynomial-time approximation scheme,

1107

for subset sum, 1128–1134, 1139

function, 1166–1168

Ackermann’s, 585

basis, 835

convex, 1199

ﬁnal-state, 996

hash, see hash function

linear, 26, 845

objective, 664, 847, 851

potential, 459

preﬁx, 1003–1004

quadratic, 27

reduction, 1067

sufﬁx, 996

transition, 995, 1001–1002, 1012 ex.

functional iteration, 58

fundamental theorem of linear programming,

892

furthest-in-future strategy, 449 pr.

fusion tree, 212, 483

fuzzy sorting, 189 pr.

Gabow’s scaling algorithm for single-source

shortest paths, 679 pr.

gap character, 989 ex., 1002 ex.

gap heuristic, 760 ex., 766

garbage collection, 151, 243

gate, 1070

Gaussian elimination, 819, 842

gcd, see greatest common divisor

general number-ﬁeld sieve, 984

generating function, 108 pr.

generator

of a subgroup, 944

of Z

n

, 955

GENERIC-MST, 626

GENERIC-PUSH-RELABEL, 741

generic push-relabel algorithm, 740–748

geometric distribution, 1202–1203

and balls and bins, 134

geometric series, 1147

geometry, computational, 1014–1047

GF.2/, 1227 pr.

gift wrapping, 1037, 1047

1266 Index

global variable, 21

Goldberg’s algorithm, see push-relabel

algorithm

golden ratio (), 59, 108 pr.

gossiping, 478

GRAFT, 583 pr.

Graham’s scan, 1030–1036, 1047

GRAHAM-SCAN, 1031

graph, 1168–1173

adjacency-list representation of, 590

adjacency-matrix representation of, 591

algorithms for, 587–766

and asymptotic notation, 588

attributes of, 588, 592

breadth-ﬁrst search of, 594–602, 623

coloring of, 1103 pr.

complement of, 1090

component, 617

constraint, 666–668

dense, 589

depth-ﬁrst search of, 603–612, 623

dynamic, 562 n.

-dense, 706 pr.

hamiltonian, 1061

incidence matrix of, 448 pr., 593 ex.

interval, 422 ex.

nonhamiltonian, 1061

shortest path in, 597

singly connected, 612 ex.

sparse, 589

static, 562 n.

subproblem, 367–368

tour of, 1096

weighted, 591

see also directed acyclic graph, directed

graph, ﬂow network, undirected graph,

tree

graphic matroid, 437–438, 642

GRAPH-ISOMORPHISM, 1065 ex.

gray vertex, 594, 603

greatest common divisor (gcd), 929–930,

933 ex.

binary gcd algorithm for, 981 pr.

Euclid’s algorithm for, 933–939, 981 pr., 983

with more than two arguments, 939 ex.

recursion theorem for, 934

greedoid, 450

GREEDY, 440

GREEDY-ACTIVITY-SELECTOR, 421

greedy algorithm, 414–450

for activity selection, 415–422

for coin changing, 446 pr.

compared with dynamic programming, 381,

390 ex., 418, 423–427

Dijkstra’s algorithm, 658–664

elements of, 423–428

for fractional knapsack problem, 426

greedy-choice property in, 424–425

for Huffman code, 428–437

Kruskal’s algorithm, 631–633

and matroids, 437–443

for minimum spanning tree, 631–638

for multithreaded scheduling, 781–783

for off-line caching, 449 pr.

optimal substructure in, 425

Prim’s algorithm, 634–636

for set cover, 1117–1122, 1139

for task scheduling, 443–446, 447–448 pr.

on a weighted matroid, 439–442

for weighted set cover, 1135 pr.

greedy-choice property, 424–425

of activity selection, 417–418

of Huffman codes, 433–434

of a weighted matroid, 441

greedy scheduler, 782

GREEDY-SET-COVER, 1119

grid, 760 pr.

group, 939–946

cyclic, 955

operator (˚), 939

guessing the solution, in the substitution

method, 84–85

half 3-CNF satisﬁability, 1101 ex.

half-open interval, 348

Hall’s theorem, 735 ex.

halting problem, 1048

halving lemma, 908

HAM-CYCLE, 1062

hamiltonian cycle, 1049, 1061, 1091–1096,

1105

hamiltonian graph, 1061

hamiltonian path, 1066 ex., 1101 ex.

HAM-PATH, 1066 ex.

handle, 163, 507

handshaking lemma, 1172 ex.

Index 1267

harmonic number, 1147, 1153–1154

harmonic series, 1147, 1153–1154

HASH-DELETE, 277 ex.

hash function, 256, 262–269

auxiliary, 272

collision-resistant, 964

division method for, 263, 268–269 ex.

-universal, 269 ex.

multiplication method for, 263–264

universal, 265–268

hashing, 253–285

with chaining, 257–260, 283 pr.

double, 272–274, 277 ex.

k-universal, 284 pr.

in memoization, 365, 387

with open addressing, 269–277

perfect, 277–282, 285

to replace adjacency lists, 593 ex.

universal, 265–268

HASH-INSERT, 270, 277 ex.

HASH-SEARCH, 271, 277 ex.

hash table, 256–261

dynamic, 471 ex.

secondary, 278

see also hashing

hash value, 256

hat-check problem, 122 ex.

head

in a disk drive, 485

of a linked list, 236

of a queue, 234

heap, 151–169

analyzed by potential method, 462 ex.

binomial, 527 pr.

building, 156–159, 166 pr.

compared with Fibonacci heaps, 506–507

d-ary, 167 pr., 706 pr.

deletion from, 166 ex.

in Dijkstra’s algorithm, 662

extracting the maximum key from, 163

Fibonacci, see Fibonacci heap

as garbage-collected storage, 151

height of, 153

in Huffman’s algorithm, 433

to implement a mergeable heap, 506

increasing a key in, 163–164

insertion into, 164

in Johnson’s algorithm, 704

max-heap, 152

maximum key of, 163

mergeable, see mergeable heap

min-heap, 153

in Prim’s algorithm, 636

as a priority queue, 162–166

relaxed, 530

running times of operations on, 506 ﬁg.

and treaps, 333 pr.

2-3-4, 529 pr.

HEAP-DECREASE-KEY, 165 ex.

HEAP-DELETE, 166 ex.

HEAP-EXTRACT-MAX, 163

HEAP-EXTRACT-MIN, 165 ex.

HEAP-INCREASE-KEY, 164

HEAP-MAXIMUM, 163

HEAP-MINIMUM, 165 ex.

heap property, 152

maintenance of, 154–156

vs. binary-search-tree property, 289 ex.

heapsort, 151–169

HEAPSORT, 160

heel, 602 ex.

height

of a binomial tree, 527 pr.

black-, 309

of a B-tree, 489–490

of a d-ary heap, 167 pr.

of a decision tree, 193

exponential, 300

of a heap, 153

of a node in a heap, 153, 159 ex.

of a node in a tree, 1177

of a red-black tree, 309

of a tree, 1177

height-balanced tree, 333 pr.

height function, in push-relabel algorithms, 738

hereditary family of subsets, 437

Hermitian matrix, 832 ex.

high endpoint of an interval, 348

high function, 537, 546

HIRE-ASSISTANT, 115

hiring problem, 114–115, 123–124, 145

on-line, 139–141

probabilistic analysis of, 120–121

hit

cache, 449 pr.

spurious, 991

1268 Index

HOARE-PARTITION, 185 pr.

HOPCROFT-KARP, 764 pr.

Hopcroft-Karp bipartite matching algorithm,

763 pr.

horizontal ray, 1021 ex.

Horner’s rule, 41 pr., 900

in the Rabin-Karp algorithm, 990

HUFFMAN, 431

Huffman code, 428–437, 450

hull, convex, 8, 1029–1039, 1046 pr.

Human Genome Project, 6

hyperedge, 1172

hypergraph, 1172

and bipartite graphs, 1173 ex.

ideal parallel computer, 779

idempotency laws for sets, 1159

identity, 939

identity matrix, 1218

if, in pseudocode, 20

image, 1167

image compression, 409 pr., 413

inadmissible edge, 749

incidence, 1169

incidence matrix

and difference constraints, 666

of a directed graph, 448 pr., 593 ex.

of an undirected graph, 448 pr.

inclusion and exclusion, 1163 ex.

incomplete step, 782

INCREASE-KEY, 162

increasing a key, in a max-heap, 163–164

INCREMENT, 454

incremental design method, 29

for ﬁnding the convex hull, 1030

in-degree, 1169

indentation in pseudocode, 20

independence

of events, 1192–1193, 1195 ex.

of random variables, 1197

of subproblems in dynamic programming,

383–384

independent family of subsets, 437

independent set, 1101 pr.

of tasks, 444

independent strands, 789

index function, 537, 546

index of an element of Z

n

, 955

indicator random variable, 118–121

in analysis of expected height of a randomly

built binary search tree, 300–303

in analysis of inserting into a treap, 333 pr.

in analysis of streaks, 138–139

in analysis of the birthday paradox, 132–133

in approximation algorithm for

MAX-3-CNF satisﬁability, 1124

in bounding the right tail of the binomial

distribution, 1212–1213

in bucket sort analysis, 202–204

expected value of, 118

in hashing analysis, 259–260

in hiring-problem analysis, 120–121

and linearity of expectation, 119

in quicksort analysis, 182–184, 187 pr.

in randomized-selection analysis, 217–219,

226 pr.

in universal-hashing analysis, 265–266

induced subgraph, 1171

inequality constraint, 852

and equality constraints, 853

inequality, linear, 846

infeasible linear program, 851

infeasible solution, 851

inﬁnite sequence, 1166

inﬁnite set, 1161

inﬁnite sum, 1145

inﬁnity, arithmetic with, 650

INITIALIZE-PREFLOW, 740

INITIALIZE-SIMPLEX, 871, 887

INITIALIZE-SINGLE-SOURCE, 648

initial strand, 779

injective function, 1167

inner product, 1222

inorder tree walk, 287, 293 ex., 342

INORDER-TREE-WALK, 288

in-place sorting, 17, 148, 206 pr.

input

to an algorithm, 5

to a combinational circuit, 1071

distribution of, 116, 122

to a logic gate, 1070

size of, 25

input alphabet, 995

INSERT, 162, 230, 463 ex., 505

insertion

into binary search trees, 294–295

Index 1269

into a bit vector with a superimposed binary

tree, 534

into a bit vector with a superimposed tree of

constant height, 534

into B-trees, 493–497

into chained hash tables, 258

into d-ary heaps, 167 pr.

into direct-address tables, 254

into dynamic tables, 464–467

elementary, 465

into Fibonacci heaps, 510–511

into heaps, 164

into interval trees, 349

into linked lists, 237–238

into open-address hash tables, 270

into order-statistic trees, 343

into proto van Emde Boas structures, 544

into queues, 234

into red-black trees, 315–323

into stacks, 232

into sweep-line statuses, 1024

into treaps, 333 pr.

into 2-3-4 heaps, 529 pr.

into van Emde Boas trees, 552–554

into Young tableaus, 167 pr.

insertion sort, 12, 16–20, 25–27

in bucket sort, 201–204

compared with merge sort, 14 ex.

compared with quicksort, 178 ex.

decision tree for, 192 ﬁg.

in merge sort, 39 pr.

in quicksort, 185 ex.

using binary search, 39 ex.

INSERTION-SORT, 18, 26, 208 pr.

instance

of an abstract problem, 1051, 1054

of a problem, 5

instructions of the RAM model, 23

integer data type, 23

integer linear programming, 850, 895 pr.,

1101 ex.

integers ( Z ), 1158

integer-valued ﬂow, 733

integrality theorem, 734

integral, to approximate summations,

1154–1156

integration of a series, 1147

interior of a polygon, 1020 ex.

interior-point method, 850, 897

intermediate vertex, 693

internal node, 1176

internal path length, 1180 ex.

interpolation by a cubic spline, 840 pr.

interpolation by a polynomial, 901, 906 ex.

at complex roots of unity, 912–913

intersection

of chords, 345 ex.

determining, for a set of line segments,

1021–1029, 1047

determining, for two line segments,

1017–1019

of languages, 1058

of sets (\), 1159

interval, 348

fuzzy sorting of, 189 pr.

INTERVAL-DELETE, 349

interval graph, 422 ex.

INTERVAL-INSERT, 349

INTERVAL-SEARCH, 349, 351

INTERVAL-SEARCH-EXACTLY, 354 ex.

interval tree, 348–354

interval trichotomy, 348

intractability, 1048

invalid shift, 985

inventory planning, 411 pr.

inverse

of a bijective function, 1167

in a group, 940

of a matrix, 827–831, 842, 1223, 1225 ex.

multiplicative, modulo n, 949

inversion

in a self-organizing list, 476 pr.

in a sequence, 41 pr., 122 ex., 345 ex.

inverter, 1070

invertible matrix, 1223

isolated vertex, 1169

isomorphic graphs, 1171

iterated function, 63 pr.

iterated logarithm function, 58–59

ITERATIVE-FFT, 917

ITERATIVE-TREE-SEARCH, 291

iter function, 577

Jarvis’s march, 1037–1038, 1047

Jensen’s inequality, 1199

JOHNSON, 704

1270 Index

Johnson’s algorithm, 700–706

joining

of red-black trees, 332 pr.

of 2-3-4 trees, 503 pr.

joint probability density function, 1197

Josephus permutation, 355 pr.

Karmarkar’s algorithm, 897

Karp’s minimum mean-weight cycle algorithm,

680 pr.

k-ary tree, 1179

k-CNF, 1049

k-coloring, 1103 pr., 1180 pr.

k-combination, 1185

k-conjunctive normal form, 1049

kernel of a polygon, 1038 ex.

key, 16, 147, 162, 229

dummy, 397

interpreted as a natural number, 263

median, of a B-tree node, 493

public, 959, 962

secret, 959, 962

static, 277

keywords, in pseudocode, 20–22

multithreaded, 774, 776–777, 785–786

“killer adversary” for quicksort, 190

Kirchhoff’s current law, 708

Kleene star (

), 1058

KMP algorithm, 1002–1013

KMP-MATCHER, 1005

knapsack problem

fractional, 426, 428 ex.

0-1, 425, 427 ex., 1137 pr., 1139

k-neighbor tree, 338

knot, of a spline, 840 pr.

Knuth-Morris-Pratt algorithm, 1002–1013

k-permutation, 126, 1184

Kraft inequality, 1180 ex.

Kruskal’s algorithm, 631–633, 642

with integer edge weights, 637 ex.

k-sorted, 207 pr.

k-string, 1184

k-subset, 1161

k-substring, 1184

kth power, 933 ex.

k-universal hashing, 284 pr.

Lagrange’s formula, 902

Lagrange’s theorem, 944

Lam´e’s theorem, 936

language, 1057

completeness of, 1077 ex.

proving NP-completeness of, 1078–1079

veriﬁcation of, 1063

last-in, ﬁrst-out, 232

see also stack

late task, 444

layers

convex, 1044 pr.

maximal, 1045 pr.

LCA, 584 pr.

lcm (least common multiple), 939 ex.

LCS, 7, 390–397, 413

LCS-LENGTH, 394

leading submatrix, 833, 839 ex.

leaf, 1176

least common ancestor, 584 pr.

least common multiple, 939 ex.

least-squares approximation, 835–839

leaving a vertex, 1169

leaving variable, 867

LEFT, 152

left child, 1178

left-child, right-sibling representation, 246,

249 ex.

LEFT-ROTATE, 313, 353 ex.

left rotation, 312

left spine, 333 pr.

left subtree, 1178

Legendre symbol .

a

p

/, 982 pr.

length

of a path, 1170

of a sequence, 1166

of a spine, 333 pr.

of a string, 986, 1184

level

of a function, 573

of a tree, 1177

level function, 576

lexicographically less than, 304 pr.

lexicographic sorting, 304 pr.

lg (binary logarithm), 56

lg

(iterated logarithm function), 58–59

lg

k

(exponentiation of logarithms), 56

lg lg (composition of logarithms), 56

LIFO (last-in, ﬁrst-out), 232

Index 1271

see also stack

light edge, 626

linear constraint, 846

linear dependence, 1223

linear equality, 845

linear equations

solving modular, 946–950

solving systems of, 813–827

solving tridiagonal systems of, 840 pr.

linear function, 26, 845

linear independence, 1223

linear inequality, 846

linear-inequality feasibility problem, 894 pr.

linearity of expectation, 1198

and indicator random variables, 119

linearity of summations, 1146

linear order, 1165

linear permutation, 1229 pr.

linear probing, 272

linear programming, 7, 843–897

algorithms for, 850

applications of, 849

duality in, 879–886

ellipsoid algorithm for, 850, 897

ﬁnding an initial solution in, 886–891

fundamental theorem of, 892

interior-point methods for, 850, 897

Karmarkar’s algorithm for, 897

and maximum ﬂow, 860–861

and minimum-cost circulation, 896 pr.

and minimum-cost ﬂow, 861–862

and minimum-cost multicommodity ﬂow,

864 ex.

and multicommodity ﬂow, 862–863

simplex algorithm for, 864–879, 896

and single-pair shortest path, 859–860

and single-source shortest paths, 664–670,

863 ex.

slack form for, 854–857

standard form for, 850–854

see also integer linear programming, 0-1

integer programming

linear-programming relaxation, 1125

linear search, 22 ex.

linear speedup, 780

line segment, 1015

comparable, 1022

determining turn of, 1017

determining whether any intersect,

1021–1029, 1047

determining whether two intersect,

1017–1019

link

of binomial trees, 527 pr.

of Fibonacci-heap roots, 513

of trees in a disjoint-set forest, 570–571

LINK, 571

linked list, 236–241

compact, 245 ex., 250 pr.

deletion from, 238

to implement disjoint sets, 564–568

insertion into, 237–238

neighbor list, 750

searching, 237, 268 ex.

self-organizing, 476 pr.

list, see linked list

LIST-DELETE, 238

LIST-DELETE

0

, 238

LIST-INSERT, 238

LIST-INSERT

0

, 240

LIST-SEARCH, 237

LIST-SEARCH

0

, 239

literal, 1082

little-oh notation, 50–51, 64

little-omega notation, 51

Lm-distance, 1044 ex.

ln (natural logarithm), 56

load factor

of a dynamic table, 463

of a hash table, 258

load instruction, 23

local variable, 21

logarithm function (log), 56–57

discrete, 955

iterated (lg

), 58–59

logical parallelism, 777

logic gate, 1070

longest common subsequence, 7, 390–397, 413

longest palindrome subsequence, 405 pr.

LONGEST-PATH, 1060 ex.

LONGEST-PATH-LENGTH, 1060 ex.

longest simple cycle, 1101 ex.

longest simple path, 1048

in an unweighted graph, 382

in a weighted directed acyclic graph, 404 pr.

LOOKUP-CHAIN, 388

1272 Index

loop, in pseudocode, 20

parallel, 785–787

loop invariant, 18–19

for breadth-ﬁrst search, 595

for building a heap, 157

for consolidating the root list of a Fibonacci

heap, 517

for determining the rank of an element in an

order-statistic tree, 342

for Dijkstra’s algorithm, 660

and for loops, 19 n.

for the generic minimum-spanning-tree

method, 625

for the generic push-relabel algorithm, 743

for Graham’s scan, 1034

for heapsort, 160 ex.

for Horner’s rule, 41 pr.

for increasing a key in a heap, 166 ex.

initialization of, 19

for insertion sort, 18

maintenance of, 19

for merging, 32

for modular exponentiation, 957

origin of, 42

for partitioning, 171

for Prim’s algorithm, 636

for the Rabin-Karp algorithm, 993

for randomly permuting an array, 127,

128 ex.

for red-black tree insertion, 318

for the relabel-to-front algorithm, 755

for searching an interval tree, 352

for the simplex algorithm, 872

for string-matching automata, 998, 1000

and termination, 19

low endpoint of an interval, 348

lower bounds

on approximations, 1140

asymptotic, 48

for average sorting, 207 pr.

on binomial coefﬁcients, 1186

for comparting water jugs, 206 pr.

for convex hull, 1038 ex., 1047

for disjoint-set data structures, 585

for ﬁnding the minimum, 214

for ﬁnding the predecessor, 560

for length of an optimal traveling-salesman

tour, 1112–1115

for median ﬁnding, 227

for merging, 208 pr.

for minimum-weight vertex cover,

1124–1126

for multithreaded computations, 780

and potential functions, 478

for priority-queue operations, 531

and recurrences, 67

for simultaneous minimum and maximum,

215 ex.

for size of an optimal vertex cover, 1110,

1135 pr.

for sorting, 191–194, 205 pr., 211, 531

for streaks, 136–138, 142 ex.

on summations, 1152, 1154

lower median, 213

lower square root

#

p

, 546

lower-triangular matrix, 1219, 1222 ex.,

1225 ex.

low function, 537, 546

LU decomposition, 806 pr., 819–822

LU-DECOMPOSITION, 821

LUP decomposition, 806 pr., 815

computation of, 822–825

of a diagonal matrix, 827 ex.

in matrix inversion, 828

and matrix multiplication, 832 ex.

of a permutation matrix, 827 ex.

use of, 815–819

LUP-DECOMPOSITION, 824

LUP-SOLVE, 817

main memory, 484

MAKE-HEAP, 505

MAKE-SET, 561

disjoint-set-forest implementation of, 571

linked-list implementation of, 564

makespan, 1136 pr.

MAKE-TREE, 583 pr.

Manhattan distance, 225 pr., 1044 ex.

marked node, 508, 519–520

Markov’s inequality, 1201 ex.

master method for solving a recurrence, 93–97

master theorem, 94

proof of, 97–106

matched vertex, 732

matching

bipartite, 732, 763 pr.

Index 1273

maximal, 1110, 1135 pr.

maximum, 1135 pr.

and maximum ﬂow, 732–736, 747 ex.

perfect, 735 ex.

of strings, 985–1013

weighted bipartite, 530

matric matroid, 437

matrix, 1217–1229

addition of, 1220

adjacency, 591

conjugate transpose of, 832 ex.

determinant of, 1224–1225

diagonal, 1218

Hermitian, 832 ex.

identity, 1218

incidence, 448 pr., 593 ex.

inversion of, 806 pr., 827–831, 842

lower-triangular, 1219, 1222 ex., 1225 ex.

multiplication of, see matrix multiplication

negative of, 1220

permutation, 1220, 1222 ex.

predecessor, 685

product of, with a vector, 785–787, 789–790,

792 ex.

pseudoinverse of, 837

scalar multiple of, 1220

subtraction of, 1221

symmetric, 1220

symmetric positive-deﬁnite, 832–835, 842

Toeplitz, 921 pr.

transpose of, 797 ex., 1217

transpose of, multithreaded, 792 ex.

tridiagonal, 1219

unit lower-triangular, 1219

unit upper-triangular, 1219

upper-triangular, 1219, 1225 ex.

Vandermonde, 902, 1226 pr.

matrix-chain multiplication, 370–378

MATRIX-CHAIN-MULTIPLY

MATRIX-CHAIN-ORDER, 375

matrix multiplication, 75–83, 1221

for all-pairs shortest paths, 686–693,

706–707

boolean, 832 ex.

and computing the determinant, 832 ex.

divide-and-conquer method for, 76–83

and LUP decomposition, 832 ex.

and matrix inversion, 828–831, 842

multithreaded algorithm for, 792–797,

806 pr.

Pan’s method for, 82 ex.

Strassen’s algorithm for, 79–83, 111–112

MATRIX-MULTIPLY, 371

matrix-vector multiplication, multithreaded,

785–787, 792 ex.

with race, 789–790

matroid, 437–443, 448 pr., 450, 642

for task scheduling, 443–446

MAT-VEC, 785

MAT-VEC-MAIN-LOOP, 786

MAT-VEC-WRONG, 790

MAX-CNF satisﬁability, 1127 ex.

MAX-CUT problem, 1127 ex.

MAX-FLOW-BY-SCALING, 763 pr.

max-ﬂow min-cut theorem, 723

max-heap, 152

building, 156–159

d-ary, 167 pr.

deletion from, 166 ex.

extracting the maximum key from, 163

in heapsort, 159–162

increasing a key in, 163–164

insertion into, 164

maximum key of, 163

as a max-priority queue, 162–166

mergeable, 250 n., 481 n., 505 n.

MAX-HEAPIFY, 154

MAX-HEAP-INSERT, 164

building a heap with, 166 pr.

max-heap property, 152

maintenance of, 154–156

maximal element, of a partially ordered set,

1165

maximal layers, 1045 pr.

maximal matching, 1110, 1135 pr.

maximal point, 1045 pr.

maximal subset, in a matroid, 438

maximization linear program, 846

and minimization linear programs, 852

maximum, 213

in binary search trees, 291

of a binomial distribution, 1207 ex.

in a bit vector with a superimposed binary

tree, 533

in a bit vector with a superimposed tree of

constant height, 535

1274 Index

ﬁnding, 214–215

in heaps, 163

in order-statistic trees, 347 ex.

in proto van Emde Boas structures, 544 ex.

in red-black trees, 311

in van Emde Boas trees, 550

MAXIMUM, 162–163, 230

maximum bipartite matching, 732–736,

747 ex., 766

Hopcroft-Karp algorithm for, 763 pr.

maximum degree, in a Fibonacci heap, 509,

523–526

maximum ﬂow, 708–766

Edmonds-Karp algorithm for, 727–730

Ford-Fulkerson method for, 714–731, 765

as a linear program, 860–861

and maximum bipartite matching, 732–736,

747 ex.

push-relabel algorithms for, 736–760, 765

relabel-to-front algorithm for, 748–760

scaling algorithm for, 762 pr., 765

updating, 762 pr.

maximum matching, 1135 pr.

maximum spanning tree, 1137 pr.

maximum-subarray problem, 68–75, 111

max-priority queue, 162

MAX-3-CNF satisﬁability, 1123–1124, 1139

MAYBE-MST-A, 641 pr.

MAYBE-MST-B, 641 pr.

MAYBE-MST-C, 641 pr.

mean, see expected value

mean weight of a cycle, 680 pr.

median, 213–227

multithreaded algorithm for, 805 ex.

of sorted lists, 223 ex.

of two sorted lists, 804 ex.

weighted, 225 pr.

median key, of a B-tree node, 493

median-of-3 method, 188 pr.

member of a set (2), 1158

membership

in proto van Emde Boas structures, 540–541

in Van Emde Boas trees, 550

memoization, 365, 387–389

MEMOIZED-CUT-ROD, 365

MEMOIZED-CUT-ROD-AUX, 366

MEMOIZED-MATRIX-CHAIN, 388

memory, 484

memory hierarchy, 24

MERGE, 31

mergeable heap, 481, 505

binomial heaps, 527 pr.

linked-list implementation of, 250 pr.

relaxed heaps, 530

running times of operations on, 506 ﬁg.

2-3-4 heaps, 529 pr.

see also Fibonacci heap

mergeable max-heap, 250 n., 481 n., 505 n.

mergeable min-heap, 250 n., 481 n., 505

MERGE-LISTS, 1129

merge sort, 12, 30–37

compared with insertion sort, 14 ex.

multithreaded algorithm for, 797–805, 812

use of insertion sort in, 39 pr.

MERGE-SORT, 34

MERGE-SORT

0

, 797

merging

of k sorted lists, 166 ex.

lower bounds for, 208 pr.

multithreaded algorithm for, 798–801

of two sorted arrays, 30

MILLER-RABIN, 970

Miller-Rabin primality test, 968–975, 983

MIN-GAP, 354 ex.

min-heap, 153

analyzed by potential method, 462 ex.

building, 156–159

d-ary, 706 pr.

in Dijkstra’s algorithm, 662

in Huffman’s algorithm, 433

in Johnson’s algorithm, 704

mergeable, 250 n., 481 n., 505

as a min-priority queue, 165 ex.

in Prim’s algorithm, 636

MIN-HEAPIFY, 156 ex.

MIN-HEAP-INSERT, 165 ex.

min-heap ordering, 507

min-heap property, 153, 507

maintenance of, 156 ex.

in treaps, 333 pr.

vs. binary-search-tree property, 289 ex.

minimization linear program, 846

and maximization linear programs, 852

minimum, 213

in binary search trees, 291

Index 1275

in a bit vector with a superimposed binary

tree, 533

in a bit vector with a superimposed tree of

constant height, 535

in B-trees, 497 ex.

in Fibonacci heaps, 511

ﬁnding, 214–215

off-line, 582 pr.

in order-statistic trees, 347 ex.

in proto van Emde Boas structures, 541–542

in red-black trees, 311

in 2-3-4 heaps, 529 pr.

in van Emde Boas trees, 550

MINIMUM, 162, 214, 230, 505

minimum-cost circulation, 896 pr.

minimum-cost ﬂow, 861–862

minimum-cost multicommodity ﬂow, 864 ex.

minimum-cost spanning tree, see minimum

spanning tree

minimum cut, 721, 731 ex.

minimum degree, of a B-tree, 489

minimum mean-weight cycle, 680 pr.

minimum node, of a Fibonacci heap, 508

minimum path cover, 761 pr.

minimum spanning tree, 624–642

in approximation algorithm for

traveling-salesman problem, 1112

Bor˙uvka’s algorithm for, 641

on dynamic graphs, 637 ex.

generic method for, 625–630

Kruskal’s algorithm for, 631–633

Prim’s algorithm for, 634–636

relation to matroids, 437, 439–440

second-best, 638 pr.

minimum-weight spanning tree, see minimum

spanning tree

minimum-weight vertex cover, 1124–1127,

1139

minor of a matrix, 1224

min-priority queue, 162

in constructing Huffman codes, 431

in Dijkstra’s algorithm, 661

in Prim’s algorithm, 634, 636

miss, 449 pr.

missing child, 1178

mod, 54, 928

modifying operation, 230

modular arithmetic, 54, 923 pr., 939–946

modular equivalence, 54, 1165 ex.

modular exponentiation, 956

MODULAR-EXPONENTIATION, 957

modular linear equations, 946–950

MODULAR-LINEAR-EQUATION-SOLVER,

949

modulo, 54, 928

Monge array, 110 pr.

monotone sequence, 168

monotonically decreasing, 53

monotonically increasing, 53

Monty Hall problem, 1195 ex.

move-to-front heuristic, 476 pr., 478

MST-KRUSKAL, 631

MST-PRIM, 634

MST-REDUCE, 639 pr.

much-greater-than (  
), 574

much-less-than (), 783

multicommodity ﬂow, 862–863

minimum-cost, 864 ex.

multicore computer, 772

multidimensional fast Fourier transform,

921 pr.

multigraph, 1172

converting to equivalent undirected graph,

593 ex.

multiple, 927

of an element modulo n, 946–950

least common, 939 ex.

scalar, 1220

multiple assignment, 21

multiple sources and sinks, 712

multiplication

of complex numbers, 83 ex.

divide-and-conquer method for, 920 pr.

of matrices, see matrix multiplication

of a matrix chain, 370–378

matrix-vector, multithreaded, 785–787,

789–790, 792 ex.

modulo n (n), 940

of polynomials, 899

multiplication method, 263–264

multiplicative group modulo n, 941

multiplicative inverse, modulo n, 949

multiply instruction, 23

MULTIPOP, 453

multiprocessor, 772

MULTIPUSH, 456 ex.

1276 Index

multiset, 1158 n.

multithreaded algorithm, 10, 772–812

for computing Fibonacci numbers, 774–780

for fast Fourier transform, 804 ex.

Floyd-Warshall algorithm, 797 ex.

for LU decomposition, 806 pr.

for LUP decomposition, 806 pr.

for matrix inversion, 806 pr.

for matrix multiplication, 792–797, 806 pr.

for matrix transpose, 792 ex., 797 ex.

for matrix-vector product, 785–787,

789–790, 792 ex.

for median, 805 ex.

for merge sorting, 797–805, 812

for merging, 798–801

for order statistics, 805 ex.

for partitioning, 804 ex.

for preﬁx computation, 807 pr.

for quicksort, 811 pr.

for reduction, 807 pr.

for a simple stencil calculation, 809 pr.

for solving systems of linear equations,

806 pr.

Strassen’s algorithm, 795–796

multithreaded composition, 784 ﬁg.

multithreaded computation, 777

multithreaded scheduling, 781–783

mutually exclusive events, 1190

mutually independent events, 1193

N (set of natural numbers), 1158

naive algorithm, for string matching, 988–990

NAIVE-STRING-MATCHER, 988

natural cubic spline, 840 pr.

natural numbers ( N ), 1158

keys interpreted as, 263

negative of a matrix, 1220

negative-weight cycle

and difference constraints, 667

and relaxation, 677 ex.

and shortest paths, 645, 653–654, 692 ex.,

700 ex.

negative-weight edges, 645–646

neighbor, 1172

neighborhood, 735 ex.

neighbor list, 750

nested parallelism, 776, 805 pr.

nesting boxes, 678 pr.

net ﬂow across a cut, 720

network

admissible, 749–750

ﬂow, see ﬂow network

residual, 715–719

for sorting, 811

NEXT-TO-TOP, 1031

NIL, 21

node, 1176

see also vertex

nonbasic variable, 855

nondeterministic multithreaded algorithm, 787

nondeterministic polynomial time, 1064 n.

see also NP

nonhamiltonian graph, 1061

noninstance, 1056 n.

noninvertible matrix, 1223

nonnegativity constraint, 851, 853

nonoverlappable string pattern, 1002 ex.

nonsaturating push, 739, 745

nonsingular matrix, 1223

nontrivial power, 933 ex.

nontrivial square root of 1, modulo n, 956

no-path property, 650, 672

normal equation, 837

norm of a vector, 1222

NOT function (:), 1071

not a set member (62), 1158

not equivalent (6), 54

NOT gate, 1070

NP (complexity class), 1049, 1064, 1066 ex.,

1105

NPC (complexity class), 1050, 1069

NP-complete, 1050, 1069

NP-completeness, 9–10, 1048–1105

of the circuit-satisﬁability problem,

1070–1077

of the clique problem, 1086–1089, 1105

of determining whether a boolean formula is

a tautology, 1086 ex.

of the formula-satisﬁability problem,

1079–1081, 1105

of the graph-coloring problem, 1103 pr.

of the half 3-CNF satisﬁability problem,

1101 ex.

of the hamiltonian-cycle problem,

1091–1096, 1105

of the hamiltonian-path problem, 1101 ex.

Index 1277

of the independent-set problem, 1101 pr.

of integer linear programming, 1101 ex.

of the longest-simple-cycle problem,

1101 ex.

proving, of a language, 1078–1079

of scheduling with proﬁts and deadlines,

1104 pr.

of the set-covering problem, 1122 ex.

of the set-partition problem, 1101 ex.

of the subgraph-isomorphism problem,

1100 ex.

of the subset-sum problem, 1097–1100

of the 3-CNF-satisﬁability problem,

1082–1085, 1105

of the traveling-salesman problem,

1096–1097

of the vertex-cover problem, 1089–1091,

1105

of 0-1 integer programming, 1100 ex.

NP-hard, 1069

n-set, 1161

n-tuple, 1162

null event, 1190

null tree, 1178

null vector, 1224

number-ﬁeld sieve, 984

numerical stability, 813, 815, 842

n-vector, 1218

o-notation, 50–51, 64

O-notation, 45 ﬁg., 47–48, 64

O

0

-notation, 62 pr.

e O-notation, 62 pr.

object, 21

allocation and freeing of, 243–244

array implementation of, 241–246

passing as parameter, 21

objective function, 664, 847, 851

objective value, 847, 851

oblivious compare-exchange algorithm, 208 pr.

occurrence of a pattern, 985

OFF-LINE-MINIMUM, 583 pr.

off-line problem

caching, 449 pr.

least common ancestors, 584 pr.

minimum, 582 pr.

Omega-notation, 45 ﬁg., 48–49, 64

1-approximation algorithm, 1107

one-pass method, 585

one-to-one correspondence, 1167

one-to-one function, 1167

on-line convex-hull problem, 1039 ex.

on-line hiring problem, 139–141

ON-LINE-MAXIMUM, 140

on-line multithreaded scheduler, 781

ON-SEGMENT, 1018

onto function, 1167

open-address hash table, 269–277

with double hashing, 272–274, 277 ex.

with linear probing, 272

with quadratic probing, 272, 283 pr.

open interval, 348

OpenMP, 774

optimal binary search tree, 397–404, 413

OPTIMAL-BST, 402

optimal objective value, 851

optimal solution, 851

optimal subset, of a matroid, 439

optimal substructure

of activity selection, 416

of binary search trees, 399–400

in dynamic programming, 379–384

of the fractional knapsack problem, 426

in greedy algorithms, 425

of Huffman codes, 435

of longest common subsequences, 392–393

of matrix-chain multiplication, 373

of rod-cutting, 362

of shortest paths, 644–645, 687, 693–694

of unweighted shortest paths, 382

of weighted matroids, 442

of the 0-1 knapsack problem, 426

optimal vertex cover, 1108

optimization problem, 359, 1050, 1054

approximation algorithms for, 10,

1106–1140

and decision problems, 1051

OR function (\_), 697, 1071

order

of a group, 945

linear, 1165

partial, 1165

total, 1165

ordered pair, 1161

ordered tree, 1177

order of growth, 28

1278 Index

order statistics, 213–227

dynamic, 339–345

multithreaded algorithm for, 805 ex.

order-statistic tree, 339–345

querying, 347 ex.

OR gate, 1070

origin, 1015

or, in pseudocode, 22

orthonormal, 842

OS-KEY-RANK, 344 ex.

OS-RANK, 342

OS-SELECT, 341

out-degree, 1169

outer product, 1222

output

of an algorithm, 5

of a combinational circuit, 1071

of a logic gate, 1070

overdetermined system of linear equations, 814

overﬂow

of a queue, 235

of a stack, 233

overﬂowing vertex, 736

discharge of, 751

overlapping intervals, 348

ﬁnding all, 354 ex.

point of maximum overlap, 354 pr.

overlapping rectangles, 354 ex.

overlapping subproblems, 384–386

overlapping-sufﬁx lemma, 987

P (complexity class), 1049, 1055, 1059,

1061 ex., 1105

package wrapping, 1037, 1047

page on a disk, 486, 499 ex., 502 pr.

pair, ordered, 1161

pairwise disjoint sets, 1161

pairwise independence, 1193

pairwise relatively prime, 931

palindrome, 405 pr.

Pan’s method for matrix multiplication, 82 ex.

parallel algorithm, 10, 772

see also multithreaded algorithm

parallel computer, 772

ideal, 779

parallel for, in pseudocode, 785–786

parallelism

logical, 777

of a multithreaded computation, 780

nested, 776

of a randomized multithreaded algorithm,

811 pr.

parallel loop, 785–787, 805 pr.

parallel-machine-scheduling problem, 1136 pr.

parallel preﬁx, 807 pr.

parallel random-access machine, 811

parallel slackness, 781

rule of thumb, 783

parallel, strands being logically in, 778

parameter, 21

costs of passing, 107 pr.

parent

in a breadth-ﬁrst tree, 594

in a multithreaded computation, 776

in a rooted tree, 1176

PARENT, 152

parenthesis structure of depth-ﬁrst search, 606

parenthesis theorem, 606

parenthesization of a matrix-chain product, 370

parse tree, 1082

partially ordered set, 1165

partial order, 1165

PARTITION, 171

PARTITION

0

, 186 pr.

partition function, 361 n.

partitioning, 171–173

around median of 3 elements, 185 ex.

Hoare’s method for, 185 pr.

multithreaded algorithm for, 804 ex.

randomized, 179

partition of a set, 1161, 1164

Pascal’s triangle, 1188 ex.

path, 1170

augmenting, 719–720, 763 pr.

critical, 657

ﬁnd, 569

hamiltonian, 1066 ex.

longest, 382, 1048

shortest, see shortest paths

simple, 1170

weight of, 643

PATH, 1051, 1058

path compression, 569

path cover, 761 pr.

path length, of a tree, 304 pr., 1180 ex.

path-relaxation property, 650, 673

Index 1279

pattern, in string matching, 985

nonoverlappable, 1002 ex.

pattern matching, see string matching

penalty, 444

perfect hashing, 277–282, 285

perfect linear speedup, 780

perfect matching, 735 ex.

permutation, 1167

bit-reversal, 472 pr., 918

Josephus, 355 pr.

k-permutation, 126, 1184

linear, 1229 pr.

in place, 126

random, 124–128

of a set, 1184

uniform random, 116, 125

permutation matrix, 1220, 1222 ex., 1226 ex.

LUP decomposition of, 827 ex.

PERMUTE-BY-CYCLIC, 129 ex.

PERMUTE-BY-SORTING, 125

PERMUTE-WITH-ALL, 129 ex.

PERMUTE-WITHOUT-IDENTITY, 128 ex.

persistent data structure, 331 pr., 482

PERSISTENT-TREE-INSERT, 331 pr.

PERT chart, 657, 657 ex.

P-FIB, 776

phase, of the relabel-to-front algorithm, 758

phi function (.n/), 943

PISANO-DELETE, 526 pr.

pivot

in linear programming, 867, 869–870,

878 ex.

in LU decomposition, 821

in quicksort, 171

PIVOT, 869

platter, 485

P-MATRIX-MULTIPLY-RECURSIVE, 794

P-MERGE, 800

P-MERGE-SORT, 803

pointer, 21

array implementation of, 241–246

trailing, 295

point-value representation, 901

polar angle, 1020 ex.

Pollard’s rho heuristic, 976–980, 980 ex., 984

POLLARD-RHO, 976

polygon, 1020 ex.

kernel of, 1038 ex.

star-shaped, 1038 ex.

polylogarithmically bounded, 57

polynomial, 55, 898

addition of, 898

asymptotic behavior of, 61 pr.

coefﬁcient representation of, 900

derivatives of, 922 pr.

evaluation of, 41 pr., 900, 905 ex., 923 pr.

interpolation by, 901, 906 ex.

multiplication of, 899, 903–905, 920 pr.

point-value representation of, 901

polynomial-growth condition, 113

polynomially bounded, 55

polynomially related, 1056

polynomial-time acceptance, 1058

polynomial-time algorithm, 927, 1048

polynomial-time approximation scheme, 1107

for maximum clique, 1134 pr.

polynomial-time computability, 1056

polynomial-time decision, 1059

polynomial-time reducibility (P), 1067,

1077 ex.

polynomial-time solvability, 1055

polynomial-time veriﬁcation, 1061–1066

POP, 233, 452

pop from a run-time stack, 188 pr.

positional tree, 1178

positive-deﬁnite matrix, 1225

post-ofﬁce location problem, 225 pr.

postorder tree walk, 287

potential function, 459

for lower bounds, 478

potential method, 459–463

for binary counters, 461–462

for disjoint-set data structures, 575–581,

582 ex.

for dynamic tables, 466–471

for Fibonacci heaps, 509–512, 517–518,

520–522

for the generic push-relabel algorithm, 746

for min-heaps, 462 ex.

for restructuring red-black trees, 474 pr.

for self-organizing lists with move-to-front,

476 pr.

for stack operations, 460–461

potential, of a data structure, 459

power

of an element, modulo n, 954–958

1280 Index

kth, 933 ex.

nontrivial, 933 ex.

power series, 108 pr.

power set, 1161

Pr f g (probability distribution), 1190

PRAM, 811

predecessor

in binary search trees, 291–292

in a bit vector with a superimposed binary

tree, 534

in a bit vector with a superimposed tree of

constant height, 535

in breadth-ﬁrst trees, 594

in B-trees, 497 ex.

in linked lists, 236

in order-statistic trees, 347 ex.

in proto van Emde Boas structures, 544 ex.

in red-black trees, 311

in shortest-paths trees, 647

in Van Emde Boas trees, 551–552

PREDECESSOR, 230

predecessor matrix, 685

predecessor subgraph

in all-pairs shortest paths, 685

in breadth-ﬁrst search, 600

in depth-ﬁrst search, 603

in single-source shortest paths, 647

predecessor-subgraph property, 650, 676

preemption, 447 pr.

preﬁx

of a sequence, 392

of a string ( ), 986

preﬁx code, 429

preﬁx computation, 807 pr.

preﬁx function, 1003–1004

preﬁx-function iteration lemma, 1007

preﬂow, 736, 765

preimage of a matrix, 1228 pr.

preorder, total, 1165

preorder tree walk, 287

presorting, 1043

Prim’s algorithm, 634–636, 642

with an adjacency matrix, 637 ex.

in approximation algorithm for

traveling-salesman problem, 1112

implemented with a Fibonacci heap, 636

implemented with a min-heap, 636

with integer edge weights, 637 ex.

similarity to Dijkstra’s algorithm, 634, 662

for sparse graphs, 638 pr.

primality testing, 965–975, 983

Miller-Rabin test, 968–975, 983

pseudoprimality testing, 966–968

primal linear program, 880

primary clustering, 272

primary memory, 484

prime distribution function, 965

prime number, 928

density of, 965–966

prime number theorem, 965

primitive root of Z

n

, 955

principal root of unity, 907

principle of inclusion and exclusion, 1163 ex.

PRINT-ALL-PAIRS-SHORTEST-PATH, 685

PRINT-CUT-ROD-SOLUTION, 369

PRINT-INTERSECTING-SEGMENTS, 1028 ex.

PRINT-LCS, 395

PRINT-OPTIMAL-PARENS, 377

PRINT-PATH, 601

PRINT-SET, 572 ex.

priority queue, 162–166

in constructing Huffman codes, 431

in Dijkstra’s algorithm, 661

heap implementation of, 162–166

lower bounds for, 531

max-priority queue, 162

min-priority queue, 162, 165 ex.

with monotone extractions, 168

in Prim’s algorithm, 634, 636

proto van Emde Boas structure

implementation of, 538–545

van Emde Boas tree implementation of,

531–560

see also binary search tree, binomial heap,

Fibonacci heap

probabilistically checkable proof, 1105, 1140

probabilistic analysis, 115–116, 130–142

of approximation algorithm for

MAX-3-CNF satisﬁability, 1124

and average inputs, 28

of average node depth in a randomly built

binary search tree, 304 pr.

of balls and bins, 133–134

of birthday paradox, 130–133

of bucket sort, 201–204, 204 ex.

of collisions, 261 ex., 282 ex.

Index 1281

of convex hull over a sparse-hulled

distribution, 1046 pr.

of ﬁle comparison, 995 ex.

of fuzzy sorting of intervals, 189 pr.

of hashing with chaining, 258–260

of height of a randomly built binary search

tree, 299–303

of hiring problem, 120–121, 139–141

of insertion into a binary search tree with

equal keys, 303 pr.

of longest-probe bound for hashing, 282 pr.

of lower bound for sorting, 205 pr.

of Miller-Rabin primality test, 971–975

and multithreaded algorithms, 811 pr.

of on-line hiring problem, 139–141

of open-address hashing, 274–276, 277 ex.

of partitioning, 179 ex., 185 ex., 187–188 pr.

of perfect hashing, 279–282

of Pollard’s rho heuristic, 977–980

of probabilistic counting, 143 pr.

of quicksort, 181–184, 187–188 pr., 303 ex.

of Rabin-Karp algorithm, 994

and randomized algorithms, 123–124

of randomized selection, 217–219, 226 pr.

of searching a compact list, 250 pr.

of slot-size bound for chaining, 283 pr.

of sorting points by distance from origin,

204 ex.

of streaks, 135–139

of universal hashing, 265–268

probabilistic counting, 143 pr.

probability, 1189–1196

probability density function, 1196

probability distribution, 1190

probability distribution function, 204 ex.

probe sequence, 270

probing, 270, 282 pr.

see also linear probing, quadratic probing,

double hashing

problem

abstract, 1054

computational, 5–6

concrete, 1055

decision, 1051, 1054

intractable, 1048

optimization, 359, 1050, 1054

solution to, 6, 1054–1055

tractable, 1048

procedure, 6, 16–17

product .

Q

/, 1148

Cartesian, 1162

cross, 1016

inner, 1222

of matrices, 1221, 1226 ex.

outer, 1222

of polynomials, 899

rule of, 1184

scalar ﬂow, 714 ex.

professional wrestler, 602 ex.

program counter, 1073

programming, see dynamic programming,

linear programming

proper ancestor, 1176

proper descendant, 1176

proper subgroup, 944

proper subset (), 1159

proto van Emde Boas structure, 538–545

cluster in, 538

compared with van Emde Boas trees, 547

deletion from, 544

insertion into, 544

maximum in, 544 ex.

membership in, 540–541

minimum in, 541–542

predecessor in, 544 ex.

successor in, 543–544

summary in, 540

PROTO-VEB-INSERT, 544

PROTO-VEB-MEMBER, 541

PROTO-VEB-MINIMUM, 542

proto-vEB structure, see proto van Emde Boas

structure

PROTO-VEB-SUCCESSOR, 543

prune-and-search method, 1030

pruning a Fibonacci heap, 529 pr.

P-SCAN-1, 808 pr.

P-SCAN-2, 808 pr.

P-SCAN-3, 809 pr.

P-SCAN-DOWN, 809 pr.

P-SCAN-UP, 809 pr.

pseudocode, 16, 20–22

pseudoinverse, 837

pseudoprime, 966–968

PSEUDOPRIME, 967

pseudorandom-number generator, 117

P-SQUARE-MATRIX-MULTIPLY, 793

1282 Index

P-TRANSPOSE, 792 ex.

public key, 959, 962

public-key cryptosystem, 958–965, 983

PUSH

push-relabel operation, 739

stack operation, 233, 452

push onto a run-time stack, 188 pr.

push operation (in push-relabel algorithms),

738–739

nonsaturating, 739, 745

saturating, 739, 745

push-relabel algorithm, 736–760, 765

basic operations in, 738–740

by discharging an overﬂowing vertex of

maximum height, 760 ex.

to ﬁnd a maximum bipartite matching,

747 ex.

gap heuristic for, 760 ex., 766

generic algorithm, 740–748

with a queue of overﬂowing vertices, 759 ex.

relabel-to-front algorithm, 748–760

quadratic function, 27

quadratic probing, 272, 283 pr.

quadratic residue, 982 pr.

quantile, 223 ex.

query, 230

queue, 232, 234–235

in breadth-ﬁrst search, 595

implemented by stacks, 236 ex.

linked-list implementation of, 240 ex.

priority, see priority queue

in push-relabel algorithms, 759 ex.

quicksort, 170–190

analysis of, 174–185

average-case analysis of, 181–184

compared with insertion sort, 178 ex.

compared with radix sort, 199

with equal element values, 186 pr.

good worst-case implementation of, 223 ex.

“killer adversary” for, 190

with median-of-3 method, 188 pr.

multithreaded algorithm for, 811 pr.

randomized version of, 179–180, 187 pr.

stack depth of, 188 pr.

tail-recursive version of, 188 pr.

use of insertion sort in, 185 ex.

worst-case analysis of, 180–181

QUICKSORT, 171

QUICKSORT

0

, 186 pr.

quotient, 928

R (set of real numbers), 1158

Rabin-Karp algorithm, 990–995, 1013

RABIN-KARP-MATCHER, 993

race, 787–790

RACE-EXAMPLE, 788

radix sort, 197–200

compared with quicksort, 199

RADIX-SORT, 198

radix tree, 304 pr.

RAM, 23–24

RANDOM, 117

random-access machine, 23–24

parallel, 811

randomized algorithm, 116–117, 122–130

and average inputs, 28

comparison sort, 205 pr.

for fuzzy sorting of intervals, 189 pr.

for hiring problem, 123–124

for insertion into a binary search tree with

equal keys, 303 pr.

for MAX-3-CNF satisﬁability, 1123–1124,

1139

Miller-Rabin primality test, 968–975, 983

multithreaded, 811 pr.

for partitioning, 179, 185 ex., 187–188 pr.

for permuting an array, 124–128

Pollard’s rho heuristic, 976–980, 980 ex.,

984

and probabilistic analysis, 123–124

quicksort, 179–180, 185 ex., 187–188 pr.

randomized rounding, 1139

for searching a compact list, 250 pr.

for selection, 215–220

universal hashing, 265–268

worst-case performance of, 180 ex.

RANDOMIZED-HIRE-ASSISTANT, 124

RANDOMIZED-PARTITION, 179

RANDOMIZED-QUICKSORT, 179, 303 ex.

relation to randomly built binary search

trees, 304 pr.

randomized rounding, 1139

RANDOMIZED-SELECT, 216

RANDOMIZE-IN-PLACE, 126

Index 1283

randomly built binary search tree, 299–303,

304 pr.

random-number generator, 117

random permutation, 124–128

uniform, 116, 125

RANDOM-SAMPLE, 130 ex.

random sampling, 129 ex., 179

RANDOM-SEARCH, 143 pr.

random variable, 1196–1201

indicator, see indicator random variable

range, 1167

of a matrix, 1228 pr.

rank

column, 1223

full, 1223

of a matrix, 1223, 1226 ex.

of a node in a disjoint-set forest, 569, 575,

581 ex.

of a number in an ordered set, 300, 339

in order-statistic trees, 341–343, 344–345 ex.

row, 1223

rate of growth, 28

ray, 1021 ex.

RB-DELETE, 324

RB-DELETE-FIXUP, 326

RB-ENUMERATE, 348 ex.

RB-INSERT, 315

RB-INSERT-FIXUP, 316

RB-JOIN, 332 pr.

RB-TRANSPLANT, 323

reachability in a graph ( ), 1170

real numbers ( R ), 1158

reconstructing an optimal solution, in dynamic

programming, 387

record, 147

rectangle, 354 ex.

recurrence, 34, 65–67, 83–113

solution by Akra-Bazzi method, 112–113

solution by master method, 93–97

solution by recursion-tree method, 88–93

solution by substitution method, 83–88

recurrence equation, see recurrence

recursion, 30

recursion tree, 37, 88–93

in proof of master theorem, 98–100

and the substitution method, 91–92

RECURSIVE-ACTIVITY-SELECTOR, 419

recursive case, 65

RECURSIVE-FFT, 911

RECURSIVE-MATRIX-CHAIN, 385

red-black tree, 308–338

augmentation of, 346–347

compared with B-trees, 484, 490

deletion from, 323–330

in determining whether any line segments

intersect, 1024

for enumerating keys in a range, 348 ex.

height of, 309

insertion into, 315–323

joining of, 332 pr.

maximum key of, 311

minimum key of, 311

predecessor in, 311

properties of, 308–312

relaxed, 311 ex.

restructuring, 474 pr.

rotation in, 312–314

searching in, 311

successor in, 311

see also interval tree, order-statistic tree

REDUCE, 807 pr.

reduced-space van Emde Boas tree, 557 pr.

reducibility, 1067–1068

reduction algorithm, 1052, 1067

reduction function, 1067

reduction, of an array, 807 pr.

reﬂexive relation, 1163

reﬂexivity of asymptotic notation, 51

region, feasible, 847

regularity condition, 95

rejection

by an algorithm, 1058

by a ﬁnite automaton, 996

RELABEL, 740

relabeled vertex, 740

relabel operation, in push-relabel algorithms,

740, 745

RELABEL-TO-FRONT, 755

relabel-to-front algorithm, 748–760

phase of, 758

relation, 1163–1166

relatively prime, 931

RELAX, 649

relaxation

of an edge, 648–650

linear programming, 1125

1284 Index

relaxed heap, 530

relaxed red-black tree, 311 ex.

release time, 447 pr.

remainder, 54, 928

remainder instruction, 23

repeated squaring

for all-pairs shortest paths, 689–691

for raising a number to a power, 956

repeat, in pseudocode, 20

repetition factor, of a string, 1012 pr.

REPETITION-MATCHER, 1013 pr.

representative of a set, 561

RESET, 459 ex.

residual capacity, 716, 719

residual edge, 716

residual network, 715–719

residue, 54, 928, 982 pr.

respecting a set of edges, 626

return edge, 779

return, in pseudocode, 22

return instruction, 23

reweighting

in all-pairs shortest paths, 700–702

in single-source shortest paths, 679 pr.

rho heuristic, 976–980, 980 ex., 984

.n/-approximation algorithm, 1106, 1123

RIGHT, 152

right child, 1178

right-conversion, 314 ex.

right horizontal ray, 1021 ex.

RIGHT-ROTATE, 313

right rotation, 312

right spine, 333 pr.

right subtree, 1178

rod-cutting, 360–370, 390 ex.

root

of a tree, 1176

of unity, 906–907

of Z

n

, 955

rooted tree, 1176

representation of, 246–249

root list, of a Fibonacci heap, 509

rotation

cyclic, 1012 ex.

in a red-black tree, 312–314

rotational sweep, 1030–1038

rounding, 1126

randomized, 1139

row-major order, 394

row rank, 1223

row vector, 1218

RSA public-key cryptosystem, 958–965, 983

RS-vEB tree, 557 pr.

rule of product, 1184

rule of sum, 1183

running time, 25

average-case, 28, 116

best-case, 29 ex., 49

expected, 28, 117

of a graph algorithm, 588

and multithreaded computation, 779–780

order of growth, 28

rate of growth, 28

worst-case, 27, 49

sabermetrics, 412 n.

safe edge, 626

SAME-COMPONENT, 563

sample space, 1189

sampling, 129 ex., 179

SAT, 1079

satellite data, 147, 229

satisﬁability, 1072, 1079–1081, 1105,

1123–1124, 1127 ex., 1139

satisﬁable formula, 1049, 1079

satisfying assignment, 1072, 1079

saturated edge, 739

saturating push, 739, 745

scalar ﬂow product, 714 ex.

scalar multiple, 1220

scaling

in maximum ﬂow, 762 pr., 765

in single-source shortest paths, 679 pr.

scan, 807 pr.

SCAN, 807 pr.

scapegoat tree, 338

schedule, 444, 1136 pr.

event-point, 1023

scheduler, for multithreaded computations,

777, 781–783, 812

centralized, 782

greedy, 782

work-stealing algorithm for, 812

scheduling, 443–446, 447 pr., 450, 1104 pr.,

1136 pr.

Schur complement, 820, 834

Index 1285

Schur complement lemma, 834

SCRAMBLE-SEARCH, 143 pr.

seam carving, 409 pr., 413

SEARCH, 230

searching, 22 ex.

binary search, 39 ex., 799–800

in binary search trees, 289–291

in B-trees, 491–492

in chained hash tables, 258

in compact lists, 250 pr.

in direct-address tables, 254

for an exact interval, 354 ex.

in interval trees, 350–353

linear search, 22 ex.

in linked lists, 237

in open-address hash tables, 270–271

in proto van Emde Boas structures, 540–541

in red-black trees, 311

in an unsorted array, 143 pr.

in Van Emde Boas trees, 550

search tree, see balanced search tree, binary

search tree, B-tree, exponential search

tree, interval tree, optimal binary search

tree, order-statistic tree, red-black tree,

splay tree, 2-3 tree, 2-3-4 tree

secondary clustering, 272

secondary hash table, 278

secondary storage

search tree for, 484–504

stacks on, 502 pr.

second-best minimum spanning tree, 638 pr.

secret key, 959, 962

segment, see directed segment, line segment

SEGMENTS-INTERSECT, 1018

SELECT, 220

selection, 213

of activities, 415–422, 450

and comparison sorts, 222

in expected linear time, 215–220

multithreaded, 805 ex.

in order-statistic trees, 340–341

in worst-case linear time, 220–224

selection sort, 29 ex.

selector vertex, 1093

self-loop, 1168

self-organizing list, 476 pr., 478

semiconnected graph, 621 ex.

sentinel, 31, 238–240, 309

sequence (h i)

bitonic, 682 pr.

ﬁnite, 1166

inﬁnite, 1166

inversion in, 41 pr., 122 ex., 345 ex.

probe, 270

sequential consistency, 779, 812

serial algorithm versus parallel algorithm, 772

serialization, of a multithreaded algorithm,

774, 776

series, 108 pr., 1146–1148

strands being logically in, 778

set (f g), 1158–1163

cardinality (j j), 1161

convex, 714 ex.

difference (), 1159

independent, 1101 pr.

intersection (\), 1159

member (2), 1158

not a member (62), 1158

union ([), 1159

set-covering problem, 1117–1122, 1139

weighted, 1135 pr.

set-partition problem, 1101 ex.

shadow of a point, 1038 ex.

shared memory, 772

Shell’s sort, 42

shift, in string matching, 985

shift instruction, 24

short-circuiting operator, 22

SHORTEST-PATH, 1050

shortest paths, 7, 643–707

all-pairs, 644, 684–707

Bellman-Ford algorithm for, 651–655

with bitonic paths, 682 pr.

and breadth-ﬁrst search, 597–600, 644

convergence property of, 650, 672–673

and difference constraints, 664–670

Dijkstra’s algorithm for, 658–664

in a directed acyclic graph, 655–658

in -dense graphs, 706 pr.

estimate of, 648

Floyd-Warshall algorithm for, 693–697,

700 ex., 706

Gabow’s scaling algorithm for, 679 pr.

Johnson’s algorithm for, 700–706

as a linear program, 859–860

and longest paths, 1048

1286 Index

by matrix multiplication, 686–693, 706–707

and negative-weight cycles, 645, 653–654,

692 ex., 700 ex.

with negative-weight edges, 645–646

no-path property of, 650, 672

optimal substructure of, 644–645, 687,

693–694

path-relaxation property of, 650, 673

predecessor-subgraph property of, 650, 676

problem variants, 644

and relaxation, 648–650

by repeated squaring, 689–691

single-destination, 644

single-pair, 381, 644

single-source, 643–683

tree of, 647–648, 673–676

triangle inequality of, 650, 671

in an unweighted graph, 381, 597

upper-bound property of, 650, 671–672

in a weighted graph, 643

sibling, 1176

side of a polygon, 1020 ex.

signature, 960

simple cycle, 1170

simple graph, 1170

simple path, 1170

longest, 382, 1048

simple polygon, 1020 ex.

simple stencil calculation, 809 pr.

simple uniform hashing, 259

simplex, 848

SIMPLEX, 871

simplex algorithm, 848, 864–879, 896–897

single-destination shortest paths, 644

single-pair shortest path, 381, 644

as a linear program, 859–860

single-source shortest paths, 643–683

Bellman-Ford algorithm for, 651–655

with bitonic paths, 682 pr.

and difference constraints, 664–670

Dijkstra’s algorithm for, 658–664

in a directed acyclic graph, 655–658

in -dense graphs, 706 pr.

Gabow’s scaling algorithm for, 679 pr.

as a linear program, 863 ex.

and longest paths, 1048

singleton, 1161

singly connected graph, 612 ex.

singly linked list, 236

see also linked list

singular matrix, 1223

singular value decomposition, 842

sink vertex, 593 ex., 709, 712

size

of an algorithm’s input, 25, 926–927,

1055–1057

of a binomial tree, 527 pr.

of a boolean combinational circuit, 1072

of a clique, 1086

of a set, 1161

of a subtree in a Fibonacci heap, 524

of a vertex cover, 1089, 1108

skip list, 338

slack, 855

slack form, 846, 854–857

uniqueness of, 876

slackness

complementary, 894 pr.

parallel, 781

slack variable, 855

slot

of a direct-access table, 254

of a hash table, 256

SLOW-ALL-PAIRS-SHORTEST-PATHS, 689

smoothed analysis, 897

?Socrates, 790

solution

to an abstract problem, 1054

basic, 866

to a computational problem, 6

to a concrete problem, 1055

feasible, 665, 846, 851

infeasible, 851

optimal, 851

to a system of linear equations, 814

sorted linked list, 236

see also linked list

sorting, 5, 16–20, 30–37, 147–212, 797–805

bubblesort, 40 pr.

bucket sort, 200–204

columnsort, 208 pr.

comparison sort, 191

counting sort, 194–197

fuzzy, 189 pr.

heapsort, 151–169

insertion sort, 12, 16–20

Index 1287

k-sorting, 207 pr.

lexicographic, 304 pr.

in linear time, 194–204, 206 pr.

lower bounds for, 191–194, 211, 531

merge sort, 12, 30–37, 797–805

by oblivious compare-exchange algorithms,

208 pr.

in place, 17, 148, 206 pr.

of points by polar angle, 1020 ex.

probabilistic lower bound for, 205 pr.

quicksort, 170–190

radix sort, 197–200

selection sort, 29 ex.

Shell’s sort, 42

stable, 196

table of running times, 149

topological, 8, 612–615, 623

using a binary search tree, 299 ex.

with variable-length items, 206 pr.

0-1 sorting lemma, 208 pr.

sorting network, 811

source vertex, 594, 644, 709, 712

span law, 780

spanning tree, 439, 624

bottleneck, 640 pr.

maximum, 1137 pr.

veriﬁcation of, 642

see also minimum spanning tree

span, of a multithreaded computation, 779

sparse graph, 589

all-pairs shortest paths for, 700–705

and Prim’s algorithm, 638 pr.

sparse-hulled distribution, 1046 pr.

spawn, in pseudocode, 776–777

spawn edge, 778

speedup, 780

of a randomized multithreaded algorithm,

811 pr.

spindle, 485

spine

of a string-matching automaton, 997 ﬁg.

of a treap, 333 pr.

splay tree, 338, 482

spline, 840 pr.

splitting

of B-tree nodes, 493–495

of 2-3-4 trees, 503 pr.

splitting summations, 1152–1154

spurious hit, 991

square matrix, 1218

SQUARE-MATRIX-MULTIPLY, 75, 689

SQUARE-MATRIX-MULTIPLY-RECURSIVE,

77

square of a directed graph, 593 ex.

square root, modulo a prime, 982 pr.

squaring, repeated

for all-pairs shortest paths, 689–691

for raising a number to a power, 956

stability

numerical, 813, 815, 842

of sorting algorithms, 196, 200 ex.

stack, 232–233

in Graham’s scan, 1030

implemented by queues, 236 ex.

linked-list implementation of, 240 ex.

operations analyzed by accounting method,

457–458

operations analyzed by aggregate analysis,

452–454

operations analyzed by potential method,

460–461

for procedure execution, 188 pr.

on secondary storage, 502 pr.

STACK-EMPTY, 233

standard deviation, 1200

standard encoding (h i), 1057

standard form, 846, 850–854

star-shaped polygon, 1038 ex.

start state, 995

start time, 415

state of a ﬁnite automaton, 995

static graph, 562 n.

static set of keys, 277

static threading, 773

stencil, 809 pr.

stencil calculation, 809 pr.

Stirling’s approximation, 57

storage management, 151, 243–244, 245 ex.,

261 ex.

store instruction, 23

straddle, 1017

strand, 777

ﬁnal, 779

independent, 789

initial, 779

logically in parallel, 778

1288 Index

logically in series, 778

Strassen’s algorithm, 79–83, 111–112

multithreaded, 795–796

streaks, 135–139

strictly decreasing, 53

strictly increasing, 53

string, 985, 1184

string matching, 985–1013

based on repetition factors, 1012 pr.

by ﬁnite automata, 995–1002

with gap characters, 989 ex., 1002 ex.

Knuth-Morris-Pratt algorithm for,

1002–1013

naive algorithm for, 988–990

Rabin-Karp algorithm for, 990–995, 1013

string-matching automaton, 996–1002,

1002 ex.

strongly connected component, 1170

decomposition into, 615–621, 623

STRONGLY-CONNECTED-COMPONENTS, 617

strongly connected graph, 1170

subgraph, 1171

predecessor, see predecessor subgraph

subgraph-isomorphism problem, 1100 ex.

subgroup, 943–946

subpath, 1170

subproblem graph, 367–368

subroutine

calling, 21, 23, 25 n.

executing, 25 n.

subsequence, 391

subset (), 1159, 1161

hereditary family of, 437

independent family of, 437

SUBSET-SUM, 1097

subset-sum problem

approximation algorithm for, 1128–1134,

1139

NP-completeness of, 1097–1100

with unary target, 1101 ex.

substitution method, 83–88

and recursion trees, 91–92

substring, 1184

subtract instruction, 23

subtraction of matrices, 1221

subtree, 1176

maintaining sizes of, in order-statistic trees,

343–344

success, in a Bernoulli trial, 1201

successor

in binary search trees, 291–292

in a bit vector with a superimposed binary

tree, 533

in a bit vector with a superimposed tree of

constant height, 535

ﬁnding ith, of a node in an order-statistic

tree, 344 ex.

in linked lists, 236

in order-statistic trees, 347 ex.

in proto van Emde Boas structures, 543–544

in red-black trees, 311

in Van Emde Boas trees, 550–551

SUCCESSOR, 230

such that (W), 1159

sufﬁx ( ), 986

sufﬁx function, 996

sufﬁx-function inequality, 999

sufﬁx-function recursion lemma, 1000

sum .

P

/, 1145

Cartesian, 906 ex.

inﬁnite, 1145

of matrices, 1220

of polynomials, 898

rule of, 1183

telescoping, 1148

SUM-ARRAYS, 805 pr.

SUM-ARRAYS

0

, 805 pr.

summary

in a bit vector with a superimposed tree of

constant height, 534

in proto van Emde Boas structures, 540

in van Emde Boas trees, 546

summation, 1145–1157

in asymptotic notation, 49–50, 1146

bounding, 1149–1156

formulas and properties of, 1145–1149

linearity of, 1146

summation lemma, 908

supercomputer, 772

superpolynomial time, 1048

supersink, 712

supersource, 712

surjection, 1167

SVD, 842

sweeping, 1021–1029, 1045 pr.

rotational, 1030–1038

Index 1289

sweep line, 1022

sweep-line status, 1023–1024

symbol table, 253, 262, 265

symmetric difference, 763 pr.

symmetric matrix, 1220, 1222 ex., 1226 ex.

symmetric positive-deﬁnite matrix, 832–835,

842

symmetric relation, 1163

symmetry of ‚-notation, 52

sync, in pseudocode, 776–777

system of difference constraints, 664–670

system of linear equations, 806 pr., 813–827,

840 pr.

TABLE-DELETE, 468

TABLE-INSERT, 464

tail

of a binomial distribution, 1208–1215

of a linked list, 236

of a queue, 234

tail recursion, 188 pr., 419

TAIL-RECURSIVE-QUICKSORT, 188 pr.

target, 1097

Tarjan’s off-line least-common-ancestors

algorithm, 584 pr.

task, 443

Task Parallel Library, 774

task scheduling, 443–446, 448 pr., 450

tautology, 1066 ex., 1086 ex.

Taylor series, 306 pr.

telescoping series, 1148

telescoping sum, 1148

testing

of primality, 965–975, 983

of pseudoprimality, 966–968

text, in string matching, 985

then clause, 20 n.

Theta-notation, 44–47, 64

thread, 773

Threading Building Blocks, 774

3-CNF, 1082

3-CNF-SAT, 1082

3-CNF satisﬁability, 1082–1085, 1105

approximation algorithm for, 1123–1124,

1139

and 2-CNF satisﬁability, 1049

3-COLOR, 1103 pr.

3-conjunctive normal form, 1082

tight constraint, 865

time, see running time

time domain, 898

time-memory trade-off, 365

timestamp, 603, 611 ex.

Toeplitz matrix, 921 pr.

to, in pseudocode, 20

TOP, 1031

top-down method, for dynamic programming,

365

top of a stack, 232

topological sort, 8, 612–615, 623

in computing single-source shortest paths in

a dag, 655

TOPOLOGICAL-SORT, 613

total order, 1165

total path length, 304 pr.

total preorder, 1165

total relation, 1165

tour

bitonic, 405 pr.

Euler, 623 pr., 1048

of a graph, 1096

track, 486

tractability, 1048

trailing pointer, 295

transition function, 995, 1001–1002, 1012 ex.

transitive closure, 697–699

and boolean matrix multiplication, 832 ex.

of dynamic graphs, 705 pr., 707

TRANSITIVE-CLOSURE, 698

transitive relation, 1163

transitivity of asymptotic notation, 51

TRANSPLANT, 296, 323

transpose

conjugate, 832 ex.

of a directed graph, 592 ex.

of a matrix, 1217

of a matrix, multithreaded, 792 ex.

transpose symmetry of asymptotic notation, 52

traveling-salesman problem

approximation algorithm for, 1111–1117,

1139

bitonic euclidean, 405 pr.

bottleneck, 1117 ex.

NP-completeness of, 1096–1097

with the triangle inequality, 1112–1115

without the triangle inequality, 1115–1116

1290 Index

traversal of a tree, 287, 293 ex., 342, 1114

treap, 333 pr., 338

TREAP-INSERT, 333 pr.

tree, 1173–1180

AA-trees, 338

AVL, 333 pr., 337

binary, see binary tree

binomial, 527 pr.

bisection of, 1181 pr.

breadth-ﬁrst, 594, 600

B-trees, 484–504

decision, 192–193

depth-ﬁrst, 603

diameter of, 602 ex.

dynamic, 482

free, 1172–1176

full walk of, 1114

fusion, 212, 483

heap, 151–169

height-balanced, 333 pr.

height of, 1177

interval, 348–354

k-neighbor, 338

minimum spanning, see minimum spanning

tree

optimal binary search, 397–404, 413

order-statistic, 339–345

parse, 1082

recursion, 37, 88–93

red-black, see red-black tree

rooted, 246–249, 1176

scapegoat, 338

search, see search tree

shortest-paths, 647–648, 673–676

spanning, see minimum spanning tree,

spanning tree

splay, 338, 482

treap, 333 pr., 338

2-3, 337, 504

2-3-4, 489, 503 pr.

van Emde Boas, 531–560

walk of, 287, 293 ex., 342, 1114

weight-balanced trees, 338

TREE-DELETE, 298, 299 ex., 323–324

tree edge, 601, 603, 609

TREE-INSERT, 294, 315

TREE-MAXIMUM, 291

TREE-MINIMUM, 291

TREE-PREDECESSOR, 292

TREE-SEARCH, 290

TREE-SUCCESSOR, 292

tree walk, 287, 293 ex., 342, 1114

trial, Bernoulli, 1201

trial division, 966

triangle inequality, 1112

for shortest paths, 650, 671

triangular matrix, 1219, 1222 ex., 1225 ex.

trichotomy, interval, 348

trichotomy property of real numbers, 52

tridiagonal linear systems, 840 pr.

tridiagonal matrix, 1219

trie (radix tree), 304 pr.

y-fast, 558 pr.

TRIM, 1130

trimming a list, 1130

trivial divisor, 928

truth assignment, 1072, 1079

truth table, 1070

TSP, 1096

tuple, 1162

twiddle factor, 912

2-CNF-SAT, 1086 ex.

2-CNF satisﬁability, 1086 ex.

and 3-CNF satisﬁability, 1049

two-pass method, 571

2-3-4 heap, 529 pr.

2-3-4 tree, 489

joining, 503 pr.

splitting, 503 pr.

2-3 tree, 337, 504

unary, 1056

unbounded linear program, 851

unconditional branch instruction, 23

uncountable set, 1161

underdetermined system of linear equations,

814

underﬂow

of a queue, 234

of a stack, 233

undirected graph, 1168

articulation point of, 621 pr.

biconnected component of, 621 pr.

bridge of, 621 pr.

clique in, 1086

coloring of, 1103 pr., 1180 pr.

Index 1291

computing a minimum spanning tree in,

624–642

converting to, from a multigraph, 593 ex.

d-regular, 736 ex.

grid, 760 pr.

hamiltonian, 1061

independent set of, 1101 pr.

matching of, 732

nonhamiltonian, 1061

vertex cover of, 1089, 1108

see also graph

undirected version of a directed graph, 1172

uniform hashing, 271

uniform probability distribution, 1191–1192

uniform random permutation, 116, 125

union

of dynamic sets, see uniting

of languages, 1058

of sets ([), 1159

UNION, 505, 562

disjoint-set-forest implementation of, 571

linked-list implementation of, 565–567,

568 ex.

union by rank, 569

unique factorization of integers, 931

unit (1), 928

uniting

of Fibonacci heaps, 511–512

of heaps, 506

of linked lists, 241 ex.

of 2-3-4 heaps, 529 pr.

unit lower-triangular matrix, 1219

unit-time task, 443

unit upper-triangular matrix, 1219

unit vector, 1218

universal collection of hash functions, 265

universal hashing, 265–268

universal sink, 593 ex.

universe, 1160

of keys in van Emde Boas trees, 532

universe size, 532

unmatched vertex, 732

unsorted linked list, 236

see also linked list

until, in pseudocode, 20

unweighted longest simple paths, 382

unweighted shortest paths, 381

upper bound, 47

upper-bound property, 650, 671–672

upper median, 213

upper square root

"

p

, 546

upper-triangular matrix, 1219, 1225 ex.

valid shift, 985

value

of a ﬂow, 710

of a function, 1166

objective, 847, 851

value over replacement player, 411 pr.

Vandermonde matrix, 902, 1226 pr.

van Emde Boas tree, 531–560

cluster in, 546

compared with proto van Emde Boas

structures, 547

deletion from, 554–556

insertion into, 552–554

maximum in, 550

membership in, 550

minimum in, 550

predecessor in, 551–552

with reduced space, 557 pr.

successor in, 550–551

summary in, 546

Var Œ (variance), 1199

variable

basic, 855

entering, 867

leaving, 867

nonbasic, 855

in pseudocode, 21

random, 1196–1201

slack, 855

see also indicator random variable

variable-length code, 429

variance, 1199

of a binomial distribution, 1205

of a geometric distribution, 1203

VEB-EMPTY-TREE-INSERT, 553

vEB tree, see van Emde Boas tree

VEB-TREE-DELETE, 554

VEB-TREE-INSERT, 553

VEB-TREE-MAXIMUM, 550

VEB-TREE-MEMBER, 550

VEB-TREE-MINIMUM, 550

VEB-TREE-PREDECESSOR, 552

VEB-TREE-SUCCESSOR, 551

1292 Index

vector, 1218, 1222–1224

convolution of, 901

cross product of, 1016

orthonormal, 842

in the plane, 1015

Venn diagram, 1160

veriﬁcation, 1061–1066

of spanning trees, 642

veriﬁcation algorithm, 1063

vertex

articulation point, 621 pr.

attributes of, 592

capacity of, 714 ex.

in a graph, 1168

intermediate, 693

isolated, 1169

overﬂowing, 736

of a polygon, 1020 ex.

relabeled, 740

selector, 1093

vertex cover, 1089, 1108, 1124–1127, 1139

VERTEX-COVER, 1090

vertex-cover problem

approximation algorithm for, 1108–1111,

1139

NP-completeness of, 1089–1091, 1105

vertex set, 1168

violation, of an equality constraint, 865

virtual memory, 24

Viterbi algorithm, 408 pr.

VORP, 411 pr.

walk of a tree, 287, 293 ex., 342, 1114

weak duality, 880–881, 886 ex., 895 pr.

weight

of a cut, 1127 ex.

of an edge, 591

mean, 680 pr.

of a path, 643

weight-balanced tree, 338, 473 pr.

weighted bipartite matching, 530

weighted matroid, 439–442

weighted median, 225 pr.

weighted set-covering problem, 1135 pr.

weighted-union heuristic, 566

weighted vertex cover, 1124–1127, 1139

weight function

for a graph, 591

in a weighted matroid, 439

while, in pseudocode, 20

white-path theorem, 608

white vertex, 594, 603

widget, 1092

wire, 1071

WITNESS, 969

witness, to the compositeness of a number, 968

work law, 780

work, of a multithreaded computation, 779

work-stealing scheduling algorithm, 812

worst-case running time, 27, 49

Yen’s improvement to the Bellman-Ford

algorithm, 678 pr.

y-fast trie, 558 pr.

Young tableau, 167 pr.

Z (set of integers), 1158

Z n (equivalence classes modulo n), 928

Z

n

(elements of multiplicative group

modulo n), 941

Z C

n

(nonzero elements of Z n), 967

zero matrix, 1218

zero of a polynomial modulo a prime, 950 ex.

0-1 integer programming, 1100 ex., 1125

0-1 knapsack problem, 425, 427 ex., 1137 pr.,

1139

0-1 sorting lemma, 208 pr.

zonk, 1195 ex.

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt

tttttttttttttttttt