Algorithms

Jeff Erickson

0th edition (pre-publication draft) — December 30, 2018
½th edition (pre-publication draft) — April 9, 2019
1st paperback edition — June 13, 2019

1 2 3 4 5 6 7 8 9 — 27 26 25 24 23 22 21 20 **19**

ISBN: 978-1-792-64483-2 (paperback)

© Copyright 2019 Jeff Erickson

This work is available under a Creative Commons Attribution 4.0 International License.
For license details, see http://creativecommons.org/licenses/by/4.0/.

Download this book at http://jeffe.cs.illinois.edu/teaching/algorithms/
or http://algorithms.wtf
or https://archive.org/details/Algorithms-Jeff-Erickson

Please report errors at https://github.com/jeffgerickson/algorithms

Portions of our programming are mechanically reproduced,
and we now begin our broadcast day.

For Kim, Kay, and Hannah
with love and admiration

And for Erin
with thanks
for breaking her promise

Incipit prologus in libro alghoarismi de practica arismetrice.

— Ioannis Hispalensis [John of Seville?],
Liber algorismi de pratica arismetrice (c.1135)

Shall I tell you, my friend, how you will come to understand it?
Go and write a book upon it.

— Henry Home, Lord Kames (1696–1782),
in a letter to Sir Gilbert Elliot

The individual is always mistaken. He designed many things, and drew in other persons as coadjutors, quarrelled with some or all, blundered much, and something is done; all are a little advanced, but the individual is always mistaken. It turns out somewhat new and very unlike what he promised himself.

— Ralph Waldo Emerson, "Experience", *Essays, Second Series* (1844)

What I have outlined above is the content of a book the realization of whose basic plan and the incorporation of whose details would perhaps be impossible; what I have written is a second or third draft of a preliminary version of this book

— Michael Spivak, preface of the first edition of
Differential Geometry, Volume I (1970)

Preface

About This Book

This textbook grew out of a collection of lecture notes that I wrote for various algorithms classes at the University of Illinois at Urbana-Champaign, which I have been teaching about once a year since January 1999. Spurred by changes of our undergraduate theory curriculum, I undertook a major revision of my notes in 2016; this book consists of a subset of my revised notes on the most fundamental course material, mostly reflecting the algorithmic content of our new required junior-level theory course.

Prerequisites

The algorithms classes I teach at Illinois have two significant prerequisites: a course on discrete mathematics and a course on fundamental data structures. Consequently, this textbook is probably not suitable for most students as a *first*

course in data structures and algorithms. In particular, I assume at least passing familiarity with the following specific topics:

- **Discrete mathematics:** High-school algebra, logarithm identities, naive set theory, Boolean algebra, first-order predicate logic, sets, functions, equivalences, partial orders, modular arithmetic, recursive definitions, trees (as abstract objects, not data structures), graphs (vertices and edges, not function plots).
- **Proof techniques:** direct, indirect, contradiction, exhaustive case analysis, and induction (especially "strong" and "structural" induction). Chapter 0 uses induction, and whenever Chapter $n-1$ uses induction, so does Chapter n.
- **Iterative programming concepts:** variables, conditionals, loops, records, indirection (addresses/pointers/references), subroutines, recursion. I do not assume fluency in any particular programming language, but I do assume experience with at least one language that supports both indirection and recursion.
- **Fundamental abstract data types:** scalars, sequences, vectors, sets, stacks, queues, maps/dictionaries, ordered maps/dictionaries, priority queues.
- **Fundamental data structures:** arrays, linked lists (single and double, linear and circular), binary search trees, at least one form of *balanced* binary search tree (such as AVL trees, red-black trees, treaps, skip lists, or splay trees), hash tables, binary heaps, and most importantly, the difference between this list and the previous list.
- **Fundamental computational problems:** elementary arithmetic, sorting, searching, enumeration, tree traversal (preorder, inorder, postorder, level-order, and so on).
- **Fundamental algorithms:** elementary algorism, sequential search, binary search, sorting (selection, insertion, merge, heap, quick, radix, and so on), breadth- and depth-first search in (at least binary) trees, and most importantly, the difference between this list and the previous list.
- **Elementary algorithm analysis:** Asymptotic notation (o, O, Θ, Ω, ω), translating loops into sums and recursive calls into recurrences, evaluating simple sums and recurrences.
- **Mathematical maturity:** facility with abstraction, formal (especially recursive) definitions, and (especially inductive) proofs; writing and following mathematical arguments; recognizing and avoiding syntactic, semantic, and/or logical nonsense.

The book *briefly* covers some of this prerequisite material when it arises in context, but more as a reminder than a good introduction. For a more thorough overview, I strongly recommend the following freely available references:

- Margaret M. Fleck. *Building Blocks for Theoretical Computer Science*. Version 1.3 (January 2013) or later available from http://mfleck.cs.illinois.edu/building-blocks/.
- Eric Lehman, F. Thomson Leighton, and Albert R. Meyer. *Mathematics for Computer Science*. June 2018 revision available from https://courses.csail.mit.edu/6.042/spring18/. (I strongly recommend searching for the most recent revision.)
- Pat Morin. *Open Data Structures*. Edition 0.1Gβ (January 2016) or later available from http://opendatastructures.org/.
- Don Sheehy. *A Course in Data Structures and Object-Oriented Design*. February 2019 or later revision available from https://donsheehy.github.io/datastructures/.

Additional References

Please do not restrict yourself to this or any other single reference. Authors and readers bring their own perspectives to any intellectual material; no instructor "clicks" with every student, or even with every very strong student. Finding the author that most effectively gets *their* intuition into *your* head takes some effort, but that effort pays off handsomely in the long run.

The following references have been particularly valuable sources of intuition, examples, exercises, and inspiration; this is not meant to be a complete list.

- Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. (I used this textbook as an undergraduate at Rice and again as a masters student at UC Irvine.)
- Boaz Barak. *Introduction to Theoretical Computer Science*. Textbook draft, most recently revised June 2019. (Not your grandfather's theoretical CS textbook, and so much the better for it; the fact that it's free is a delightful bonus.)
- Thomas Cormen, Charles Leiserson, Ron Rivest, and Cliff Stein. *Introduction to Algorithms*, third edition. MIT Press/McGraw-Hill, 2009. (I used the first edition as a teaching assistant at Berkeley.)
- Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2006. (Probably the closest in content to this book, but considerably less verbose.)
- Jeff Edmonds. *How to Think about Algorithms*. Cambridge University Press, 2008.
- Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

- Michael T. Goodrich and Roberto Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley & Sons, 2002.
- Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison-Wesley, 2005. Borrow it from the library if you can.
- Donald Knuth. *The Art of Computer Programming*, volumes 1–4A. Addison-Wesley, 1997 and 2011. (My parents gave me the first three volumes for Christmas when I was 14. Alas, I didn't actually read them until *much* later.)
- Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989. (I used this textbook as a teaching assistant at Berkeley.)
- Ian Parberry. *Problems on Algorithms*. Prentice-Hall, 1995 (out of print). Downloadable from https://larc.unt.edu/ian/books/free/license.html after you agree to make a small charitable donation. Please honor your agreement.
- Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, 2011.
- Robert Endre Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.
- Class notes from my own algorithms classes at Berkeley, especially those taught by Dick Karp and Raimund Seidel.
- Lecture notes, slides, homeworks, exams, video lectures, research papers, blog posts, StackExchange questions and answers, podcasts, and full-fledged MOOCs made freely available on the web by innumerable colleagues around the world.

About the Exercises

Each chapter ends with several exercises, most of which I have used at least once in a homework assignment, discussion/lab section, or exam. The exercises are *not* ordered by increasing difficulty, but (generally) clustered by common techniques or themes. Some problems are annotated with symbols as follows:

- ♥Red hearts indicate particularly challenging problems; many of these have appeared on qualifying exams for PhD students at Illinois. A small number of *really* hard problems are marked with ♥large hearts.
- ♦Blue diamonds indicate problems that require familiarity with material from later chapters, but thematically belong where they are. Problems that require familiarity with *earlier* material are not marked, however; the book, like life, is cumulative.
- ♣Green clubs indicate problems that require familiarity with material outside the scope of this book, such as finite-state machines, linear algebra, probability, or planar graphs. These are rare.
- ♠Black spades indicate problems that require a significant amount of grunt work and/or coding. These are rare.

- ★Orange stars indicate that you are eating Lucky Charms that were manufactured before 1998. Ew.

These exercises are designed as opportunities to practice, not as targets for their own sake. The goal of each problem is not to solve that specific problem, but to exercise a certain set of skills, or to practice solving a certain *type* of problem. Partly for this reason, I don't provide solutions to the exercises; the solutions are not the point. In particular, there is no "instructor's manual"; if you can't solve a problem yourself, you probably shouldn't assign it to your students. That said, you can probably find solutions to whatever homework problems I've assigned *this* semester on the web page of whatever course I'm teaching. And nothing is stopping *you* from writing an instructor's manual!

Steal This Book!

This book is published under a Creative Commons Licence that allows you to use, redistribute, adapt, and remix its contents *without my permission*, as long as you point back to the original source. A complete electronic version of this book is freely available at any of the following locations:

- The book web site: http://jeffe.cs.illinois.edu/teaching/algorithms/
- The mnemonic shortcut: http://algorithms.wtf
- The bug-report site: https://github.com/jeffgerickson/algorithms
- The Internet Archive: https://archive.org/details/Algorithms-Jeff-Erickson

The book web site also contains several hundred pages of additional lecture notes on related and more advanced material, as well as a near-complete archive of past homeworks, exams, discussion/lab problems, and other teaching resources. Whenever I teach an algorithms class, I revise, update, and sometimes cull my teaching materials, so you may find more recent revisions on the web page of whatever course I am currently teaching.

Whether you are a student or an instructor, you are more than welcome to use any subset of this textbook or my other lecture notes in your own classes, without asking my permission—that's why I put them on the web! However, please also cite this book, either by name or with a link back to http://algorithms.wtf; this is *especially* important if you are a student, and you use my course materials to help with your homework. (Please also check with your instructor.)

However, if you are an instructor, I strongly encourage you to supplement these with additional material *that you write yourself*. Writing the material yourself will strengthen your mastery and in-class presentation of the material, which will in turn improve your students' mastery of the material. It will also get you past the frustration of dealing with the parts of this book that you don't like. All textbooks are ~~crap~~ imperfect, and this one is no exception.

Finally, **please make whatever you write freely, easily, and globally available on the open web**—not hidden behind the gates of a learning management system or some other type of paywall—so that students and instructors elsewhere can benefit from your unique insights. In particular, if you develop useful resources that directly complement this textbook, such as slides, videos, or solution manuals, please let me know so that I can add links to your resources from the book web site.

Acknowledgments

This textbook draws heavily on the contributions of countless algorithms students, teachers, and researchers. In particular, I am immensely grateful to more than three thousand Illinois students who have used my lecture notes as a primary reference, offered useful (if sometimes painful) criticism, and suffered through some truly awful early drafts. Thanks also to many colleagues and students around the world who have used these notes in their own classes and have sent helpful feedback and bug reports.

I am particularly grateful for the feedback and contributions (especially exercises) from my amazing teaching assistants:

> Aditya Ramani, Akash Gautam, Alex Steiger, Alina Ene, Amir Nayyeri, Asha Seetharam, Ashish Vulimiri, Ben Moseley, Brad Sturt, Brian Ensink, Chao Xu, Charlie Carlson, Chris Neihengen, Connor Clark, Dan Bullok, Dan Cranston, Daniel Khashabi, David Morrison, Ekta Manaktala, Erin Wolf Chambers, Gail Steitz, Gio Kao, Grant Czajkowski, Hsien-Chih Chang, Igor Gammer, Jacob Laurel, John Lee, Johnathon Fischer, Junqing Deng, Kent Quanrud, Kevin Milans, Kevin Small, Konstantinos Koiliaris, Kyle Fox, Kyle Jao, Lan Chen, Mark Idleman, Michael Bond, Mitch Harris, Naveen Arivazhagen, Nick Bachmair, Nick Hurlburt, Nirman Kumar, Nitish Korula, Patrick Lin, Phillip Shih, Rachit Agarwal, Reza Zamani-Nasab, Rishi Talreja, Rob McCann, Sahand Mozaffari, Shalan Naqvi, Shripad Thite, Spencer Gordon, Srihita Vatsavaya, Subhro Roy, Tana Wattanawaroon, Umang Mathur, Vipul Goyal, Yasu Furakawa, and Yipu Wang.

I've also been helped tremendously by many discussions with faculty colleagues at Illinois: Alexandra Kolla, Cinda Heeren, Edgar Ramos, Herbert Edelsbrunner, Jason Zych, Kim Whittlesey, Lenny Pitt, Madhu Parasarathy, Mahesh Viswanathan, Margaret Fleck, Shang-Hua Teng, Steve LaValle, and especially Chandra Chekuri, Ed Reingold, and Sariel Har-Peled.

Of course this book owes a great debt to the people who taught me this algorithms stuff in the first place: Bob Bixby and Michael Pearlman at Rice; David Eppstein, Dan Hirschberg, and George Lueker at Irvine; and Abhiram Ranade, Dick Karp, Manuel Blum, Mike Luby, and Raimund Seidel at Berkeley.

I stole the first iteration of the overall course structure, and the idea to write up my own lecture notes in the first place, from Herbert Edelsbrunner; the idea of turning a subset of my notes into a book from Steve LaValle; and several components of the book design from Robert Ghrist.

Caveat Lector!

Of course, none of those people should be blamed for any flaws in the resulting book. Despite many rounds of revision and editing, this book contains several mistakes, bugs, gaffes, omissions, snafus, kludges, typos, mathos, grammaros, thinkos, brain farts, poor design decisions, historical inaccuracies, anachronisms, inconsistencies, exaggerations, dithering, blather, distortions, oversimplifications, redundancy, logorrhea, nonsense, garbage, cruft, junk, and outright lies, **all of which are entirely Steve Skiena's fault.**

I maintain an issue tracker at https://github.com/jeffgerickson/algorithms, where readers like you can submit bug reports, feature requests, and general feedback on the book. Please let me know if you find an error of any kind, whether mathematical, grammatical, historical, typographical, cultural, or otherwise, whether in the main text, in the exercises, or in my other course materials. (Steve is unlikely to care.) Of course, all other feedback is also welcome!

Enjoy!

— *Jeff*

It is traditional for the author to magnanimously accept the blame for whatever deficiencies remain. I don't. Any errors, deficiencies, or problems in this book are somebody else's fault, but I would appreciate knowing about them so as to determine who is to blame.

— Steven S. Skiena, *The Algorithm Design Manual* (1997)

No doubt this statement will be followed by an annotated list of all textbooks, and why each one is crap.

— Adam Contini, MetaFilter, January 4, 2010

Table of Contents

Preface **i**
About This Book i
Prerequisites i
Additional References iii
About the Exercises iv
Steal This Book! v
Acknowledgments vi
Caveat Lector! vii

Table of Contents **ix**

0 Introduction **1**
0.1 What is an algorithm? 1
0.2 Multiplication 3

Lattice Multiplication • Duplation and Mediation • Compass and Straightedge
0.3 Congressional Apportionment 8
0.4 A Bad Example 10
0.5 Describing Algorithms 11
Specifying the Problem • Describing the Algorithm
0.6 Analyzing Algorithms 14
Correctness • Running Time
Exercises 17

1 Recursion **21**
1.1 Reductions 21
1.2 Simplify and Delegate 22
1.3 Tower of Hanoi 24
1.4 Mergesort 26
Correctness • Analysis
1.5 Quicksort 29
Correctness • Analysis
1.6 The Pattern 31
1.7 Recursion Trees 31
♥Ignoring Floors and Ceilings Is Okay, Honest
1.8 ♥Linear-Time Selection 35
Quickselect • Good pivots • Analysis • Sanity Checking
1.9 Fast Multiplication 40
1.10 Exponentiation 42
Exercises 44

2 Backtracking **71**
2.1 N Queens 71
2.2 Game Trees 74
2.3 Subset Sum 76
Correctness • Analysis • Variants
2.4 The General Pattern 79
2.5 Text Segmentation (*Interpunctio Verborum*) 80
Index Formulation • ♥Analysis • Variants
2.6 Longest Increasing Subsequence 86
2.7 Longest Increasing Subsequence, Take 2 89
2.8 Optimal Binary Search Trees 91
♥Analysis
Exercises 93

3 Dynamic Programming **97**

3.1 Mātrāvṛtta 97
Backtracking Can Be Slow • Memo(r)ization: Remember Everything • Dynamic Programming: Fill Deliberately • Don't Remember Everything After All
3.2 ♥Aside: Even Faster Fibonacci Numbers 103
Whoa! Not so fast!
3.3 *Interpunctio Verborum Redux* 105
3.4 The Pattern: Smart Recursion 105
3.5 Warning: Greed is Stupid 107
3.6 Longest Increasing Subsequence 109
First Recurrence: Is This Next? • Second Recurrence: What's Next?
3.7 Edit Distance 111
Recursive Structure • Recurrence • Dynamic Programming
3.8 Subset Sum 116
3.9 Optimal Binary Search Trees 117
3.10 Dynamic Programming on Trees 120
Exercises 123

4 Greedy Algorithms 159
4.1 Storing Files on Tape 159
4.2 Scheduling Classes 161
4.3 General Pattern 164
4.4 Huffman Codes 165
4.5 Stable Matching 170
Some Bad Ideas • The Boston Pool and Gale-Shapley Algorithms • Running Time • Correctness • Optimality!
Exercises 176

5 Basic Graph Algorithms 187
5.1 Introduction and History 187
5.2 Basic Definitions 190
5.3 Representations and Examples 192
5.4 Data Structures 195
Adjacency Lists • Adjacency Matrices • Comparison
5.5 Whatever-First Search 199
Analysis
5.6 Important Variants 201
Stack: Depth-First • Queue: Breadth-First • Priority Queue: Best-First • Disconnected Graphs • Directed Graphs
5.7 Graph Reductions: Flood Fill 205
Exercises 207

6 Depth-First Search **225**
6.1 Preorder and Postorder 227
Classifying Vertices and Edges
6.2 Detecting Cycles 231
6.3 Topological Sort 232
Implicit Topological Sort
6.4 Memoization and Dynamic Programming 234
Dynamic Programming in Dags
6.5 Strong Connectivity 237
6.6 Strong Components in Linear Time 238
Kosaraju and Sharir's Algorithm • ♥Tarjan's Algorithm
Exercises 244

7 Minimum Spanning Trees **257**
7.1 Distinct Edge Weights 257
7.2 The Only Minimum Spanning Tree Algorithm 259
7.3 Borůvka's Algorithm 261
This is the MST Algorithm You Want
7.4 Jarník's ("Prim's") Algorithm 263
♥Improving Jarník's Algorithm
7.5 Kruskal's Algorithm 265
Exercises 268

8 Shortest Paths **273**
8.1 Shortest Path Trees 274
8.2 ♥Negative Edges 274
8.3 The Only SSSP Algorithm 276
8.4 Unweighted Graphs: Breadth-First Search 278
8.5 Directed Acyclic Graphs: Depth-First Search 282
8.6 Best-First: Dijkstra's Algorithm 284
No Negative Edges • ♥Negative Edges
8.7 Relax ALL the Edges: Bellman-Ford 289
Moore's Improvement • Dynamic Programming Formulation
Exercises 297

9 All-Pairs Shortest Paths **309**
9.1 Introduction 309
9.2 Lots of Single Sources 310
9.3 Reweighting 311
9.4 Johnson's Algorithm 312
9.5 Dynamic Programming 313
9.6 Divide and Conquer 315

9.7 Funny Matrix Multiplication 316
9.8 (Kleene-Roy-)Floyd-Warshall(-Ingerman) 318
Exercises 320

10 Maximum Flows & Minimum Cuts 327
10.1 Flows 328
10.2 Cuts 329
10.3 The Maxflow-Mincut Theorem 331
10.4 Ford and Fulkerson's augmenting-path algorithm 334
♥Irrational Capacities
10.5 Combining and Decomposing Flows 336
10.6 Edmonds and Karp's Algorithms 340
Fattest Augmenting Paths • Shortest Augmenting Paths
10.7 Further Progress 343
Exercises 344

11 Applications of Flows and Cuts 353
11.1 Edge-Disjoint Paths 353
11.2 Vertex Capacities and Vertex-Disjoint Paths 354
11.3 Bipartite Matching 355
11.4 Tuple Selection 357
Exam Scheduling
11.5 Disjoint-Path Covers 360
Minimal Faculty Hiring
11.6 Baseball Elimination 363
11.7 Project Selection 366
Exercises 368

12 NP-Hardness 379
12.1 A Game You Can't Win 379
12.2 P versus NP 381
12.3 NP-hard, NP-easy, and NP-complete 382
12.4 ♥Formal Definitions (*HC SVNT DRACONES*) 384
12.5 Reductions and Sat 385
12.6 3Sat (from CircuitSat) 388
12.7 Maximum Independent Set (from 3Sat) 390
12.8 The General Pattern 392
12.9 Clique and Vertex Cover (from Independent Set) 394
12.10 Graph Coloring (from 3Sat) 395
12.11 Hamiltonian Cycle 398
From Vertex Cover • From 3Sat • Variants and Extensions
12.12 Subset Sum (from Vertex Cover) 402

Caveat Reductor!
12.13 Other Useful NP-hard Problems 404
12.14 Choosing the Right Problem 407
12.15 A Frivolous Real-World Example 408
12.16 ♥On Beyond Zebra 412
Polynomial Space • Exponential Time • Excelsior!
Exercises 415

Index **442**

Index of People **446**

Index of Pseudocode **449**

Image Credits **451**

Colophon **453**

Hinc incipit algorismus. *Haec algorismus ars praesens dicitur in qua talibus indorum fruimur bis quinque figuris 0. 9. 8. 7. 6. 5. 4. 3. 2. 1.*

— Friar Alexander de Villa Dei, *Carmen de Algorismo* (c. 1220)

You are right to demand that an artist engage his work consciously, but you confuse two different things: solving the problem and correctly posing the question.

— Anton Chekhov, in a letter to A. S. Suvorin (October 27, 1888)

The more we reduce ourselves to machines in the lower things, the more force we shall set free to use in the higher.

— Anna C. Brackett, *The Technique of Rest* (1892)

And here I am at 2:30 a.m. writing about technique, in spite of a strong conviction that the moment a man begins to talk about technique that's proof that he is fresh out of ideas.

— Raymond Chandler, letter to Erle Stanley Gardner (May 5, 1939)

Good men don't need rules.
Today is not the day to find out why I have so many,

— The Doctor [Matt Smith], "A Good Man Goes to War", *Doctor Who* (2011)

0

Introduction

0.1 What is an algorithm?

An algorithm is an explicit, precise, unambiguous, mechanically-executable sequence of elementary instructions, usually intended to accomplish a specific purpose. For example, here is an algorithm for singing that annoying song "99 Bottles of Beer on the Wall", for arbitrary values of 99:

```
BottlesOfBeer(n):
  For i ← n down to 1
      Sing "i bottles of beer on the wall, i bottles of beer,"
      Sing "Take one down, pass it around, i − 1 bottles of beer on the wall."

  Sing "No bottles of beer on the wall, no bottles of beer,"
  Sing "Go to the store, buy some more, n bottles of beer on the wall."
```

The word "algorithm" does *not* derive, as algorithmophobic classicists might guess, from the Greek roots *arithmos* (ἀριθμός), meaning "number", and *algos*

(ἄλγος), meaning "pain". Rather, it is a corruption of the name of the 9th century Persian scholar Muḥammad ibn Mūsā al-Khwārizmī.[1] Al-Khwārizmī is perhaps best known as the writer of the treatise *Al-Kitāb al-mukhtaṣar fīhīsāb al-ğabr wa'l-muqābala*,[2] from which the modern word *algebra* derives. In a different treatise, al-Khwārizmī described the modern decimal system for writing and manipulating numbers—in particular, the use of a small circle or *ṣifr* to represent a missing quantity—which had been developed in India several centuries earlier. The methods described in this latter treatise, using either written figures or counting stones, became known in English as *algorism* or *augrym*, and its figures became known in English as *ciphers*.

Although both place-value notation and al-Khwārizmī's works were already known by some European scholars, the "Hindu-Arabic" numeric system was popularized in Europe by the medieval Italian mathematician and tradesman Leonardo of Pisa, better known as Fibonacci. Thanks in part to his 1202 book *Liber Abaci*,[3] written figures began to replace the counting table (then known as an *abacus*) and finger arithmetic[4] as the preferred platform for calculation[5] in Europe in the 13th century—*not* because written decimal figures were easier to learn or use, but because they provided an audit trail. Ciphers became common in Western Europe only with the advent of movable type, and truly ubiquitous only after cheap paper became plentiful in the early 19th century.

Eventually the word *algorism* evolved into the modern *algorithm*, via folk etymology from the Greek *arithmos* (and perhaps the previously mentioned *algos*).[6] Thus, until very recently, the word *algorithm* referred exclusively

[1]"Mohammad, father of Adbdulla, son of Moses, the Kwārizmian". Kwārizm is an ancient city, now called Khiva, in the Khorezm Province of Uzbekistan.

[2]"The Compendious Book on Calculation by Completion and Balancing"

[3]While it is tempting to translate the title *Liber Abaci* as "The Book of the Abacus", a more accurate translation is "The Book of Calculation". Both before and after Fibonacci, the Italian word *abaco* was used to describe anything related to numerical calculation—devices, methods, schools, books, and so on—much in the same way that "computer science" is used today in English, or as the Chinese phrase for "operations research" translates literally as "the study of using counting rods".

[4]☞ Reckoning with *digits*! ☜

[5]The word *calculate* derives from the Latin word *calculus*, meaning "small rock", referring to the stones on a counting table, or as Chaucer called them, *augrym stones*. In 440BCE, Herodotus wrote in his *Histories* that "The Greeks write and calculate (λογίζεσθαι ψήφοις, literally 'reckon with pebbles') from left to right; the Egyptians do the opposite. Yet they say that their way of writing is toward the right, and the Greek way toward the left." (Herodotus is strangely silent on which end of the egg the Egyptians ate first.)

[6]Some medieval sources claim that the Greek prefix "algo-" means "art" or "introduction". Others claim that algorithms were invented by a Greek philosopher, or a king of India, or perhaps a king of Spain, named "Algus" or "Algor" or "Argus". A few, possibly including Dante Alighieri, even identified the inventor with the mythological Greek shipbuilder and eponymous argonaut. It's unclear whether any of these risible claims were intended to be historically accurate, or merely mnemonic.

to mechanical techniques for place-value arithmetic using "Arabic" numerals. People trained in the fast and reliable execution of these procedures were called *algorists* or *computators*, or more simply, *computers*.

0.2 Multiplication

Although they have been a topic of formal academic study for only a few decades, algorithms have been with us since the dawn of civilization. Descriptions of step-by-step arithmetic computation are among the earliest examples of written human language, long predating the expositions by Fibonacci and al-Khwārizmī, or even the place-value notation they popularized.

Lattice Multiplication

The most familiar method for multiplying large numbers, at least for American students, is the ***lattice algorithm***. This algorithm was popularized by Fibonacci in *Liber Abaci*, who learned it from Arabic sources including al-Khwārizmī, who in turn learned it from Indian sources including Brahmagupta's 7th-century treatise *Brāhmasphuṭasiddhānta*, who may have learned it from Chinese sources. The oldest surviving descriptions of the algorithm appear in *The Mathematical Classic of Sunzi*, written in China between the 3rd and 5th centuries, and in Eutocius of Ascalon's commentaries on Archimedes' *Measurement of the Circle*, written around 500CE, but there is evidence that the algorithm was known much earlier. Eutocius credits the method to a lost treatise of Apollonius of Perga, who lived around 300BCE, entitled *Okytokion* (Ὠκυτόκιον).[7] The Sumerians recorded multiplication tables on clay tablets as early as 2600BCE, suggesting that they may have used the lattice algorithm.[8]

The lattice algorithm assumes that the input numbers are represented as explicit strings of digits; I'll assume here that we're working in base ten, but the algorithm generalizes immediately to any other base. To simplify notation,[9] the

[7]Literally "medicine that promotes quick and easy childbirth"! Pappus of Alexandria reproduced several excerpts of *Okytokion* about 200 years before Eutocius, but his description of the lattice multiplication algorithm (if he gave one) is *also* lost.

[8]There is ample evidence that ancient Sumerians calculated accurately with extremely large numbers using their base-60 place-value numerical system, but I am not aware of any surviving record of the actual methods they used. In addition to standard multiplication and reciprocal tables, tables listing the squares of integers from 1 to 59 have been found, leading some math historians to conjecture that Babylonians multiplied using an identity like $xy = ((x+y)^2 - x^2 - y^2)/2$. But this trick only works when $x + y < 60$; history is silent on how the Babylonians might have computed x^2 when $x \geq 60$.

[9]but at the risk of inflaming the historical enmity between Greece and Egypt, or Lilliput and Blefuscu, or Macs and PCs, or people who think zero is a natural number and people who are wrong

input consists of a pair of arrays $X[0..m-1]$ and $Y[0..n-1]$, representing the numbers

$$x = \sum_{i=0}^{m-1} X[i] \cdot 10^i \quad \text{and} \quad y = \sum_{j=0}^{n-1} Y[j] \cdot 10^j,$$

and similarly, the output consists of a single array $Z[0..m+n-1]$, representing the product

$$z = x \cdot y = \sum_{k=0}^{m+n-1} Z[k] \cdot 10^k.$$

The algorithm uses addition and *single-digit* multiplication as primitive operations. Addition can be performed using a simple for-loop. In practice, single-digit multiplication is performed using a lookup table, either carved into clay tablets, painted on strips of wood or bamboo, written on paper, stored in read-only memory, or memorized by the computator. The entire lattice algorithm can be summarized by the formula

$$x \cdot y \;=\; \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \left(X[i] \cdot Y[j] \cdot 10^{i+j} \right).$$

Different variants of the lattice algorithm evaluate the partial products $X[i] \cdot Y[j] \cdot 10^{i+j}$ in different orders and use different strategies for computing their sum. For example, in *Liber Abaco*, Fibonacci describes a variant that considers the mn partial products in increasing order of significance, as shown in modern pseudocode below.

```
FibonacciMultiply(X[0..m−1], Y[0..n−1]):
  hold ← 0
  for k ← 0 to n + m − 1
      for all i and j such that i + j = k
          hold ← hold + X[i] · Y[j]
      Z[k] ← hold mod 10
      hold ← ⌊hold/10⌋
  return Z[0..m + n − 1]
```

Fibonacci's algorithm is often executed by storing all the partial products in a two-dimensional table (often called a "tableau" or "grate" or "lattice") and then summing along the diagonals with appropriate carries, as shown on the right in Figure 0.1. American elementary-school students are taught to multiply one factor (the "multiplicand") by each digit in the other factor (the "multiplier"), writing down all the multiplicand-by-digit products before adding them up, as shown on the left in Figure 0.1. This was also the method described by Eutocius, although he fittingly considered the multiplier digits from left to right, as shown

in Figure 0.2. Both of these variants (and several others) are described and illustrated side by side in the anonymous 1458 textbook *L'Arte dell'Abbaco*, also known as the *Treviso Arithmetic*, the first *printed* mathematics book in the West.

Figure 0.1. Computing $934 \times 314 = 293276$ using "long" multiplication (with error-checking by casting out nines) and "lattice" multiplication, from *L'Arte dell'Abbaco* (1458). (See Image Credits at the end of the book.)

Figure 0.2. Eutocius's 6th-century calculation of $1172\frac{1}{8} \times 1172\frac{1}{8} = 1373877\frac{1}{64}$, in his commentary on Archimedes' *Measurement of the Circle*, transcribed (left) and translated into modern notation (right) by Johan Heiberg (1891). (See Image Credits at the end of the book.)

All of these variants of the lattice algorithm—and other similar variants described by Sunzi, al-Khwārizmī, Fibonacci, *L'Arte dell'Abbaco*, and many other sources—compute the product of any m-digit number and any n-digit number in $\boldsymbol{O(mn)}$ ***time***; the running time of every variant is dominated by the number of single-digit multiplications.

Duplation and Mediation

The lattice algorithm is not the oldest multiplication algorithm for which we have direct recorded evidence. An even older and arguably simpler algorithm, which does not rely on place-value notation, is sometimes called *Russian peasant multiplication*, *Ethiopian peasant multiplication*, or just ***peasant multiplication***.A

variant of this algorithm was copied into the Rhind papyrus by the Egyptian scribe Ahmes around 1650BCE, from a document he claimed was (then) about 350 years old.[10] This algorithm was still taught in elementary schools in Eastern Europe in the late 20th century; it was also commonly used by early digital computers that did not implement integer multiplication directly in hardware.

The peasant multiplication algorithm reduces the difficult task of multiplying arbitrary numbers to a sequence of four simpler operations: (1) determining parity (even or odd), (2) addition, (3) ***duplation*** (doubling a number), and (4) ***mediation*** (halving a number, rounding down).

```
PeasantMultiply(x, y):
  prod ← 0
  while x > 0
      if x is odd
          prod ← prod + y
      x ← ⌊x/2⌋
      y ← y + y
  return prod
```

x	y		$prod$
			0
123	+456	=	456
61	+912	=	1368
30	~~1824~~		
15	+3648	=	5016
7	+7296	=	12312
3	+14592	=	26904
1	+29184	=	**56088**

Figure 0.3. Multiplication by duplation and mediation

The correctness of this algorithm follows by induction from the following recursive identity, which holds for all non-negative integers x and y:

$$x \cdot y = \begin{cases} 0 & \text{if } x = 0 \\ \lfloor x/2 \rfloor \cdot (y + y) & \text{if } x \text{ is even} \\ \lfloor x/2 \rfloor \cdot (y + y) + y & \text{if } x \text{ is odd} \end{cases}$$

Arguably, this recurrence *is* the peasant multiplication algorithm. Don't let the iterative pseudocode fool you; the algorithm is fundamentally recursive!

As stated, PeasantMultiply performs $O(\log x)$ parity, addition, and mediation operations, but we can improve this bound to $O(\log \min\{x, y\})$ by swapping the two arguments when $x > y$. Assuming the numbers are represented using any reasonable place-value notation (like binary, decimal, Babylonian hexagesimal, Egyptian duodecimal, Roman numeral, Chinese counting rods, bead positions on an abacus, and so on), each operation requires at most $O(\log(xy)) = O(\log \max\{x, y\})$ single-digit operations, so the overall running time of the algorithm is $O(\log \min\{x, y\} \cdot \log \max\{x, y\}) = O(\log x \cdot \log y)$.

[10]The version of this algorithm actually used in ancient Egypt does not use mediation or parity, but it does use comparisons. To avoid halving, the algorithm pre-computes two tables by repeated doubling: one containing all the powers of 2 not exceeding x, the other containing the same powers of 2 multiplied by y. The powers of 2 that sum to x are then found by greedy subtraction, and the corresponding entries in the other table are added together to form the product.

In other words, this algorithm requires $\boldsymbol{O(mn)}$ ***time*** to multiply an m-digit number by an n-digit number; up to constant factors, this is the same running time as the lattice algorithm. This algorithm requires (a constant factor!) more paperwork to execute by hand than the lattice algorithm, but the necessary primitive operations are arguably easier for humans to perform. In fact, the two algorithms are equivalent when numbers are represented in binary.

Compass and Straightedge

Classical Greek geometers identified numbers (or more accurately, *magnitudes*) with line segments of the appropriate length, which they manipulated using two simple mechanical tools—the compass and the straightedge—versions of which had already been in common use by surveyors, architects, and other artisans for centuries. Using *only* these two tools, these scholars reduced several complex geometric constructions to the following primitive operations, starting with one or more identified reference points.

- Draw the unique line passing through two distinct identified points.
- Draw the unique circle centered at one identified point and passing through another.
- Identify the intersection point (if any) of two lines.
- Identify the intersection points (if any) of a line and a circle.
- Identify the intersection points (if any) of two circles.

In practice, Greek geometry students almost certainly drew their constructions on an *abax* (ἄβαξ), a table covered in dust or sand.[11] Centuries earlier, Egyptian surveyors carried out many of the same constructions using ropes to determine straight lines and circles on the ground.[12] However, Euclid and other Greek geometers presented compass and straightedge constructions as precise mathematical *abstractions*—points are *ideal* points; lines are *ideal* lines; and circles are *ideal* circles.

Figure 0.4 shows an algorithm, described in Euclid's *Elements* about 2500 years ago, for multiplying or dividing two magnitudes. The input consists of four distinct points A, B, C, and D, and the goal is to construct a point Z such that $|AZ| = |AC||AD|/|AB|$. In particular, if we define $|AB|$ to be our unit of length, then the algorithm computes the product of $|AC|$ and $|AD|$.

Notice that Euclid first defines a new primitive operation RightAngle by (as modern programmers would phrase it) writing a subroutine. The correctness

[11]The written numerals 1 through 9 were known in Europe at least two centuries before Fibonacci's *Liber Abaci* as "gobar numerals", from the Arabic word *ghubār* meaning dust, ultimately referring to the Indian practice of performing arithmetic on tables covered with sand. The Greek word ἄβαξ is the origin of the Latin *abacus*, which also originally referred to a sand table.

[12]Remember what "geometry" means? Democritus would later refer to these Egyptian surveyors, somewhat derisively, as *arpedonaptai* (ἀρπεδονάπται), meaning "rope-fasteners".

```
⟨⟨Construct the line perpendicular to ℓ passing through P.⟩⟩
RightAngle(ℓ, P):
  Choose a point A ∈ ℓ
  A, B ← Intersect(Circle(P, A), ℓ)
  C, D ← Intersect(Circle(A, B), Circle(B, A))
  return Line(C, D)

⟨⟨Construct a point Z such that |AZ| = |AC||AD|/|AB|.⟩⟩
MultiplyOrDivide(A, B, C, D):
  α ← RightAngle(Line(A, C), A)
  E ← Intersect(Circle(A, B), α)
  F ← Intersect(Circle(A, D), α)
  β ← RightAngle(Line(E, C), F)
  γ ← RightAngle(β, F)
  return Intersect(γ, Line(A, C))
```

Figure 0.4. Multiplication by compass and straightedge.

of the algorithm follows from the observation that triangles ACE and AZF are similar. The second and third lines of the main algorithm are ambiguous, because α intersects any circle centered at A at *two* distinct points, but the algorithm is actually correct no matter which intersection points are chosen for E and F.

Euclid's algorithm reduces the problem of multiplying two magnitudes (lengths) to a series of primitive compass-and-straightedge operations. These operations are difficult to implement precisely on a modern digital computer, but Euclid's algorithm wasn't *designed* for a digital computer. It was designed for the Platonic Ideal Geometer, wielding the Platonic Ideal Compass and the Platonic Ideal Straightedge, who could execute each operation perfectly in constant time *by definition*. In this model of computation, MultiplyOrDivide runs in $O(1)$ time!

0.3 Congressional Apportionment

Here is another real-world example of an algorithm of significant political importance. Article I, Section 2 of the United States Constitution requires that

> Representatives and direct Taxes shall be apportioned among the several States which may be included within this Union, according to their respective Numbers.... The Number of Representatives shall not exceed one for every thirty Thousand, but each State shall have at Least one Representative....

Because there are only a finite number of seats in the House of Representatives, *exact* proportional representation requires either shared or fractional representatives, neither of which are legal. As a result, over the next several decades, many different apportionment algorithms were proposed and used to round the ideal fractional solution fairly. The algorithm actually used today, called

the ***Huntington-Hill method*** or the ***method of equal proportions***, was first suggested by Census Bureau statistician Joseph Hill in 1911, refined by Harvard mathematician Edward Huntington in 1920, adopted into Federal law (2 U.S.C. §2a) in 1941, and survived a Supreme Court challenge in 1992.[13]

The Huntington-Hill method allocates representatives to states one at a time. First, in a preprocessing stage, each state is allocated one representative. Then in each iteration of the main loop, the next representative is assigned to the state with the highest *priority*. The priority of each state is defined to be $P/\sqrt{r(r+1)}$, where P is the state's population and r is the number of representatives already allocated to that state.

The algorithm is described in pseudocode in Figure 0.5. The input consists of an array $Pop[1..n]$ storing the populations of the n states and an integer R equal to the total number of representatives; the algorithm assumes $R \geq n$. (Currently, in the United States, $n = 50$ and $R = 435$.) The output array $Rep[1..n]$ records the number of representatives allocated to each state.

```
ApportionCongress(Pop[1..n], R):
  PQ ← NewPriorityQueue
  ⟨⟨Give every state its first representative⟩⟩
  for s ← 1 to n
      Rep[s] ← 1
      Insert(PQ, s, Pop[i]/√2)
  ⟨⟨Allocate the remaining n − R representatives⟩⟩
  for i ← 1 to n − R
      s ← ExtractMax(PQ)
      Rep[s] ← Rep[s] + 1
      priority ← Pop[s] / √(Rep[s](Rep[s] + 1))
      Insert(PQ, s, priority)
  return Rep[1..n]
```

Figure 0.5. The Huntington-Hill apportionment algorithm

This implementation of Huntington-Hill uses a priority queue that supports the operations NewPriorityQueue, Insert, and ExtractMax. (The actual law doesn't say anything about priority queues, of course.) The output of the algorithm, and therefore its correctness, does not depend *at all* on how this

[13] Overruling an earlier ruling by a federal district court, the Supreme Court unanimously held that ***any*** apportionment method adopted in good faith by Congress is constitutional (*United States Department of Commerce v. Montana*). The current congressional apportionment algorithm is described in gruesome detail at the U.S. Census Department web site http://www.census.gov/topics/public-sector/congressional-apportionment.html. A good history of the apportionment problem can be found at http://www.thirty-thousand.org/pages/Apportionment.htm. A report by the Congressional Research Service describing various apportionment methods is available at http://www.fas.org/sgp/crs/misc/R41382.pdf.

priority queue is implemented. The Census Bureau uses a sorted array, stored in a single column of an Excel spreadsheet, which is recalculated from scratch at every iteration. You (should have) learned a more efficient implementation in your undergraduate data structures class.

Similar apportionment algorithms are used in multi-party parliamentary elections around the world, where the number of seats allocated to each party is supposed to be proportional to the number of votes that party receives. The two most common are the *D'Hondt method*[14] and the *Webster–Sainte-Laguë method*,[15] which respectively use priorities $P/(r+1)$ and $P/(2r+1)$ in place of the square-root expression in Huntington-Hill. The Huntington-Hill method is essentially unique to the United States House of Representatives, thanks in part to the constitutional requirement that each state must be allocated at least one representative.

0.4 A Bad Example

As a prototypical example of a sequence of instructions that is *not* actually an algorithm, consider "Martin's algorithm":[16]

```
BeAMillionaireAndNeverPayTaxes():
  Get a million dollars.
  If the tax man comes to your door and says, "You have never paid taxes!"
      Say "I forgot."
```

Pretty simple, except for that first step; it's a doozy! A group of billionaire CEOs, Silicon Valley venture capitalists, or New York City real-estate hustlers might consider this an algorithm, because for them the first step is both unambiguous and trivial,[17] but for the rest of us poor slobs, Martin's procedure is too vague to be considered an actual algorithm. On the other hand, this is a perfect example of a ***reduction***—it *reduces* the problem of being a millionaire and never paying taxes to the "easier" problem of acquiring a million dollars. We'll see reductions over and over again in this book. As hundreds of businessmen and politicians have demonstrated, if you know how to solve the easier problem, a reduction tells you how to solve the harder one.

[14] developed by Thomas Jefferson in 1792, used for U.S. Congressional apportionment from 1792 to 1832, rediscovered by Belgian mathematician Victor D'Hondt in 1878, and refined by Swiss physicist Eduard Hagenbach-Bischoff in 1888.

[15] developed by Daniel Webster in 1832, used for U.S. Congressional apportionment from 1842 to 1911, rediscovered by French mathematician André Sainte-Laguë in 1910, and rediscovered again by German physicist Hans Schepers in 1980.

[16] Steve Martin, "You Can Be A Millionaire", *Saturday Night Live*, January 21, 1978. Also appears on *Comedy Is Not Pretty*, Warner Bros. Records, 1979.

[17] Something something secure quantum blockchain deep-learning something.

Martin's algorithm, like some of our previous examples, is not the kind of algorithm that computer scientists are used to thinking about, because it is phrased in terms of operations that are difficult for computers to perform. This book focuses (almost!) exclusively on algorithms that can be reasonably implemented on a standard digital computer. Each step in these algorithms is either directly supported by common programming languages (such as arithmetic, assignments, loops, or recursion) or something that you've already learned how to do (like sorting, binary search, tree traversal, or singing "*n* Bottles of Beer on the Wall").

0.5 Describing Algorithms

The skills required to effectively *design and analyze* algorithms are entangled with the skills required to effectively *describe* algorithms. At least in my classes, a complete description of any algorithm has four components:

- **What:** A precise specification of the problem that the algorithm solves.
- **How:** A precise description of the algorithm itself.
- **Why:** A proof that the algorithm solves the problem it is supposed to solve.
- **How fast:** An analysis of the running time of the algorithm.

It is not necessary (or even advisable) to *develop* these four components in this particular order. Problem specifications, algorithm descriptions, correctness proofs, and time analyses usually evolve simultaneously, with the development of each component informing the development of the others. For example, we may need to tweak the problem description to support a faster algorithm, or modify the algorithm to handle a tricky case in the proof of correctness. Nevertheless, *presenting* these components separately is usually clearest for the reader.

As with any writing, it's important to aim your descriptions at the right audience; I recommend writing for a competent but skeptical programmer *who is not as clever as you are*. Think of yourself six months ago. As you develop any new algorithm, you will naturally build up lots of intuition about the problem and about how your algorithm solves it, and your informal reasoning will be guided by that intuition. But anyone *reading* your algorithm later, or the code you derive from it, won't share your intuition or experience. Neither will your compiler. Neither will you six months from now. All they will have is your written description.

Even if you never have to explain your algorithms to anyone else, it's still important to develop them with an audience in mind. Trying to communicate clearly forces you to *think* more clearly. In particular, writing for a *novice* audience, who will interpret your words *exactly* as written, forces you to work

through fine details, no matter how "obvious" or "intuitive" your high-level ideas may seem at the moment. Similarly, writing for a *skeptical* audience forces you to develop robust arguments for correctness and efficiency, instead of trusting your intuition or your intelligence.[18]

I cannot emphasize this point enough: **Your primary job as an algorithm designer is *teaching other people* how and why your algorithms work.** If you can't communicate your ideas to other human beings, they may as well not exist. Producing correct and efficient executable code is an important but secondary goal. Convincing yourself, your professors, your (prospective) employers, your colleagues, or your students that you are smart is at best a distant third.

Specifying the Problem

Before we can even start developing a new algorithm, we have to agree on what problem our algorithm is supposed to solve. Similarly, before we can even start *describing* an algorithm, we have to *describe* the problem that the algorithm is supposed to solve.

Algorithmic problems are often presented using standard English, in terms of real-world objects. It's up to us, the algorithm designers, to restate these problems in terms of formal, abstract, *mathematical* objects—numbers, arrays, lists, graphs, trees, and so on—that we can reason about formally. We must also determine if the problem statement carries any hidden assumptions, and state those assumptions explicitly. (For example, in the song "n Bottles of Beer on the Wall", n is always a non-negative integer.[19])

We may need to refine our specification as we develop the algorithm. For example, our algorithm may require a particular input representation, or produce a particular output representation, that was left unspecified in the original informal problem description. Or our algorithm might actually solve a *more general* problem than we were originally asked to solve. (This is a common feature of recursive algorithms.)

The specification should include just enough detail that someone else could *use* our algorithm as a black box, without knowing how or why the algorithm actually works. In particular, we must describe the type *and meaning* of each input parameter, and exactly how the eventual output depends on the input parameters. On the other hand, our specification should *deliberately hide* any details that are *not* necessary to use the algorithm as a black box. Let that which does not matter truly slide.

[18] In particular, I assume that *you* are a skeptical novice!

[19] I've never heard anyone sing "$\sqrt{2}$ Bottles of Beer on the Wall." Occasionally I *have* heard set theorists singing "$\aleph_0$ bottles of beer on the wall", but for some reason they always gave up before the song was over.

For example, the lattice and duplation-and-mediation algorithms both solve the same problem: Given two non-negative integers x and y, each represented as an array of digits, compute the product $x \cdot y$, also represented as an array of digits. To someone *using* these algorithms, the choice of algorithm is completely irrelevant. On the other hand, the Greek straightedge-and-compass algorithm solves a *different problem*, because the input and output values are represented by line segments instead of arrays of digits.

Describing the Algorithm

Computer programs are concrete representations of algorithms, but algorithms are *not* programs. Rather, algorithms are abstract mechanical procedures that can be implemented in *any* programming language that supports the underlying primitive operations. The idiosyncratic syntactic details of your favorite programming language are utterly irrelevant; focusing on these will only distract you (and your readers) from what's *really* going on.[20] A good algorithm description is closer to what we should write in the *comments* of a real program than the code itself. Code is a poor medium for storytelling.

On the other hand, a plain English prose description is usually not a good idea either. Algorithms have lots of idiomatic structure—especially conditionals, loops, function calls, and recursion—that are far too easily hidden by unstructured prose. Colloquial English is full of ambiguities and shades of meaning, but algorithms must be described as unambiguously as possible. Prose is a poor medium for precision.

In my opinion, the clearest way to present an algorithm is using a combination of *pseudo*code and structured English. Pseudocode uses the *structure* of formal programming languages and mathematics to break algorithms into primitive steps; the primitive steps themselves can be written using mathematical notation, pure English, or an appropriate mixture of the two, *whatever is clearest*. Well-written pseudocode reveals the internal structure of the algorithm but hides irrelevant implementation details, making the algorithm easier to understand, analyze, debug, and implement.

[20] This is, of course, a matter of religious conviction. Armchair linguists argue incessantly over the *Sapir-Whorf hypothesis*, which states (more or less) that people think only in the categories imposed by their languages. According to an extreme formulation of this principle, some concepts in one language simply cannot be understood by speakers of other languages, not just because of technological advancement—How would you translate "jump the shark" or "Fortnite streamer" into Aramaic?—but because of inherent structural differences between languages and cultures. For a more skeptical view, see Steven Pinker's *The Language Instinct*. There is admittedly some strength to this idea when applied to different programming paradigms. (What's the Y combinator, again? How do templates work? What's an Abstract Factory?) Fortunately, those differences are too subtle to have any impact on the material in *this* book. For a compelling counterexample, see Chris Okasaki's monograph *Functional Data Structures* and its more recent descendants.

Whenever we describe an algorithm, our description should include every detail necessary to fully specify the algorithm, prove its correctness, and analyze its running time. At the same time, it should *exclude* any details that are *not* necessary to fully specify the algorithm, prove its correctness, and analyze its running time. (Slide.) At a more practical level, our description should allow a competent but skeptical programmer *who has not read this book* to quickly and correctly implement the algorithm in *their* favorite programming language, *without understanding why it works*.

I don't want to bore you with the rules I follow for writing pseudocode, but I must caution against one especially pernicious habit. ***Never*** describe repeated operations informally, as in "Do [this] first, then do [that] second, ***and so on***." or "Repeat ***this process*** until [something]". As anyone who has taken one of those frustrating "What comes next in this sequence?" tests already knows, describing the first few steps of an algorithm says little or nothing about what happens in later steps. If your algorithm has a loop, write it as a loop, and explicitly describe what happens in an *arbitrary* iteration. Similarly, if your algorithm is recursive, write it recursively, and explicitly describe the case boundaries and what happens in each case.

0.6 Analyzing Algorithms

It's not enough just to write down an algorithm and say "Behold!" We must also convince our audience (and ourselves!) that the algorithm actually does what it's supposed to do, and that it does so efficiently.

Correctness

In some application settings, it is acceptable for programs to behave correctly most of the time, on all "reasonable" inputs. Not in this book; we require algorithms that are *always* correct, for *all possible* inputs. Moreover, we must *prove* that our algorithms are correct; trusting our instincts, or trying a few test cases, isn't good enough. Sometimes correctness is truly obvious, especially for algorithms you've seen in earlier courses. On the other hand, "obvious" is all too often a synonym for "wrong". Most of the algorithms we discuss in this course require real work to prove correct. In particular, correctness proofs usually involve induction. We *like* induction. Induction is our *friend*.[21]

Of course, before we can formally prove that our algorithm does what it's supposed to do, we have to formally describe what it's supposed to do!

[21] If induction is *not* your friend, you will have a hard time with this book.

Running Time

The most common way of ranking different algorithms for the same problem is by how quickly they run. Ideally, we want the fastest possible algorithm for any particular problem. In many application settings, it is acceptable for programs to run efficiently most of the time, on all "reasonable" inputs. Not in this book; we require algorithms that *always* run efficiently, even in the worst case.

But how do we measure running time? As a specific example, how long does it take to sing the song BottlesOfBeer(n)? This is obviously a function of the input value n, but it also depends on how quickly you can sing. Some singers might take ten seconds to sing a verse; others might take twenty. Technology widens the possibilities even further. Dictating the song over a telegraph using Morse code might take a full minute per verse. Downloading an mp3 over the Web might take a tenth of a second per verse. Duplicating the mp3 in a computer's main memory might take only a few microseconds per verse.

What's important here is how the singing time changes as n grows. Singing BottlesOfBeer($2n$) requires about twice much time as singing BottlesOfBeer(n), no matter what technology is being used. This is reflected in the asymptotic singing time $\Theta(n)$.

We can measure time by counting how many times the algorithm executes a certain instruction or reaches a certain milestone in the "code". For example, we might notice that the word "beer" is sung three times in every verse of BottlesOfBeer, so the number of times you sing "beer" is a good indication of the total singing time. For this question, we can give an exact answer: BottlesOfBeer(n) mentions beer exactly $3n+3$ times.

Incidentally, there are *lots* of songs with quadratic singing time. This one is probably familiar to most English-speakers:

```
NDaysOfChristmas(gifts[2..n]):
  for i ← 1 to n
      Sing "On the ith day of Christmas, my true love gave to me"
      for j ← i down to 2
          Sing "j gifts[j],"
      if i > 1
          Sing "and"
      Sing "a partridge in a pear tree."
```

The input to NDaysOfChristmas is a list of $n-1$ gifts, represented here as an array. It's quite easy to show that the singing time is $\Theta(n^2)$; in particular, the singer mentions the name of a gift $\sum_{i=1}^{n} i = n(n+1)/2$ times (counting the partridge in the pear tree). It's also easy to see that during the first n days of Christmas, my true love gave to me exactly $\sum_{i=1}^{n}\sum_{j=1}^{i} j = n(n+1)(n+2)/6 = \Theta(n^3)$ gifts.

Other quadratic-time songs include "Old MacDonald Had a Farm", "There Was an Old Lady Who Swallowed a Fly", "Hole in the Bottom of the Sea", "Green Grow the Rushes O", "The Rattlin' Bog", "The Court Of King Caractacus","The Barley-Mow", "If I Were Not Upon the Stage", "Star Trekkin'","Ist das nicht ein Schnitzelbank?",[22]"Il Pulcino Pio", "Minkurinn í hænsnakofanum", "Echad Mi Yodea", and "Το κοκοράκι". For more examples, consult your favorite preschooler.

```
ALOUETTE(lapart[1..n]):
  Chantez « Alouette, gentille alouette, alouette, je te plumerai. »
  pour tout i de 1 à n
      Chantez « Je te plumerai lapart[i]. Je te plumerai lapart[i]. »
      pour tout j de i à 1 ⟨⟨à rebours⟩⟩
          Chantez « Et lapart[j]! Et lapart[j]! »
      Chantez « Alouette! Alouette! Aaaaaa… »
      Chantez « …alouette, gentille allouette, alouette, je te plumerai. »
```

A few songs have even more bizarre singing times. A fairly modern example is "The TELNET Song" by Guy Steele, which actually takes $\Theta(2^n)$ time to sing the first n verses; Steele recommended $n = 4$. Finally, there are some songs that *never* end.[23]

Except for "The TELNET Song", all of these songs are most naturally expressed as a small set of nested loops, so their ~~running~~ singing times can be computed using nested summations. The running time of a *recursive* algorithm is more easily expressed as a recurrence. For example, the peasant multiplication algorithm can be expressed recursively as follows:

$$x \cdot y = \begin{cases} 0 & \text{if } x = 0 \\ \lfloor x/2 \rfloor \cdot (y + y) & \text{if } x \text{ is even} \\ \lfloor x/2 \rfloor \cdot (y + y) + y & \text{if } x \text{ is odd} \end{cases}$$

Let $T(x, y)$ denote the number of parity, addition, and mediation operations required to compute $x \cdot y$. This function satisfies the recursive inequality $T(x, y) \leq T(\lfloor x/2 \rfloor, 2y) + 2$ with base case $T(0, y) = 0$. Techniques described in the next chapter imply the upper bound $T(x, y) = O(\log x)$.

Sometimes the running time of an algorithm depends on a particular implementation of some underlying data structure of subroutine. For example, the Huntington-Hill apportionment algorithm APPORTIONCONGRESS runs in $O(N + RI + (R - n)E)$ time, where N denotes the running time of NEWPRIORITYQUEUE, I denotes the running time of INSERT, and E denotes the running time

[22]Ja, das ist Otto von Schnitzelpusskrankengescheitmeyer!

[23]They just go on and on, my friend.

of ExtractMax. Under the reasonable assumption that $R \geq 2n$ (on average, each state gets at least two representatives), we can simplify this bound to $O(N + R(I + E))$. The precise running time depends on the implementation of the underlying priority queue. The Census Bureau implements the priority queue as an unsorted array, which gives us $N = I = \Theta(1)$ and $E = \Theta(n)$, so the Census Bureau's implementation of ApportionCongress runs in $\boldsymbol{O(Rn)}$ ***time***. However, if we implement the priority queue as a binary heap or a heap-ordered array, we have $N = \Theta(1)$ and $I = E = O(\log n)$, so the overall algorithm runs in $\boldsymbol{O(R \log n)}$ time.

Finally, sometimes we are interested in computational resources other than time, such as space, number of coin flips, number of cache or page faults, number of inter-process messages, or the number of gifts my true love gave to me. These resources can be analyzed using the same techniques used to analyze running time. For example, lattice multiplication of two n-digit numbers requires $O(n^2)$ space if we write down all the partial products before adding them, but only $O(n)$ space if we add them on the fly.

Exercises

0. Describe and analyze an efficient algorithm that determines, given a legal arrangement of standard pieces on a standard chess board, which player will win at chess from the given starting position if both players play perfectly. *[Hint: There is a trivial one-line solution!]*

♥1. (a) Identify (or write) a song that requires $\Theta(n^3)$ time to sing the first n verses.
   (b) Identify (or write) a song that requires $\Theta(n \log n)$ time to sing the first n verses.
   (c) Identify (or write) a song that requires some other weird amount of time to sing the first n verses.

2. Careful readers might complain that our analysis of songs like "n Bottles of Beer on the Wall" or "The n Days of Christmas" is overly simplistic, because larger numbers take longer to sing than shorter numbers. More generally, because there are only so many words of a given length, larger sets of words necessarily contain longer words.[24] We can more accurately estimate singing time by counting the number of *syllables* sung, rather than the number of *words*.

   (a) How long does it take to sing the integer n?

[24] Ja, das ist das Subatomarteilchenbeschleunigungsnaturmäßigkeitsuntersuchungsmaschine!

(b) How long does it take to sing "n Bottles of Beer on the Wall"?

(c) How long does it take to sing "The n Days of Christmas"?

As usual, express your answers in the form $O(f(n))$ for some function f.

3. The cumulative drinking song "The Barley Mow" has been sung throughout the British Isles for centuries. The song has many variants; Figure 0.6 contains pseudolyrics for one version traditionally sung in Devon and Cornwall, where *vessel*$[i]$ is the name of a vessel that holds 2^i ounces of beer.[25]

```
BarleyMow(n):
  "Here's a health to the barley-mow, my brave boys,"
  "Here's a health to the barley-mow!"

  "We'll drink it out of the jolly brown bowl,"
  "Here's a health to the barley-mow!"
  "Here's a health to the barley-mow, my brave boys,"
  "Here's a health to the barley-mow!"

  for i ← 1 to n
      "We'll drink it out of the vessel[i], boys,"
      "Here's a health to the barley-mow!"
      for j ← i downto 1
          "The vessel[j],"
      "And the jolly brown bowl!"
      "Here's a health to the barley-mow!"
      "Here's a health to the barley-mow, my brave boys,"
      "Here's a health to the barley-mow!"
```

Figure 0.6. "The Barley Mow".

(a) Suppose each name *vessel*$[i]$ is a single word, and you can sing four words a second. How long would it take you to sing BarleyMow(n)? (Give a tight asymptotic bound.)

(b) If you want to sing this song for arbitrarily large values of n, you'll have to make up your own vessel names. To avoid repetition, these names must become progressively longer as n increases. Suppose *vessel*$[n]$ has

[25] In practice, the song uses some subset of the following vessels; nipperkin, quarter-gill, half-a-gill, gill, quarter-pint, half-a-pint, pint, quart, pottle, gallon, half-anker, anker, firkin, half-barrel/kilderkin, barrel, hogshead, pipe/butt, tun, well, river, and ocean. With a few exceptions (especially at the end), every vessel in this list has twice the volume of its predecessor. Irish and Scottish versions of the song have slightly different lyrics, and they usually switch to people (barmaid, landlord, drayer, and so on) after "gallon".

An early version of the song entitled "Give us once a drink" appears in the play *Jack Drum's Entertainment (or the Comedie of Pasquill and Katherine)* written by John Marston around 1600. ("Giue vs once a drinke for and the black bole. Sing gentle Butler *bally moy*!") There is some disagreement whether Marston wrote the "high Dutch Song" specifically for the play, whether "bally moy" is a mondegreen for "barley mow" or vice versa, or whether it's actually the same song at all. These discussions are best had over n bottles of beer.

$\Theta(\log n)$ syllables, and you can sing six syllables per second. Now how long would it take you to sing BarleyMow(n)? (Give a tight asymptotic bound.)

(c) Suppose each time you mention the name of a vessel, you actually drink the corresponding amount of beer: one ounce for the jolly brown bowl, and 2^i ounces for each *vessel*$[i]$. Assuming for purposes of this problem that you are at least 21 years old, *exactly* how many ounces of beer would you drink if you sang BarleyMow(n)? (Give an *exact* answer, not just an asymptotic bound.)

4. Recall that the input to the Huntington-Hill algorithm ApportionCongress is an array $Pop[1..n]$, where $Pop[i]$ is the population of the ith state, and an integer R, the total number of representatives to be allotted. The output is an array $Rep[1..n]$, where $Rep[i]$ is the number of representatives allotted to the ith state by the algorithm.

The Huntington-Hill algorithm is sometimes described in a way that avoids the use of priority queues entirely. The top-level algorithm "guesses" a positive real number D, called the *divisor*, and then runs the following subroutine to compute an apportionment. The variable q is the ideal *quota* of representatives allocated to a state for the given divisor D; the actual number of representatives allocated is always either $\lceil q \rceil$ or $\lfloor q \rfloor$.

```
HHGuess(Pop[1..n], R, D):
  reps ← 0
  for i ← 1 to n
      q ← Pop[i]/D
      if q · q < ⌈q⌉ · ⌊q⌋
          Rep[i] ← ⌊q⌋
      else
          Rep[i] ← ⌈q⌉
      reps ← reps + Rep[i]
  return reps
```

There are three possibilities for the final return value *reps*. If $reps < R$, we did not allocate enough representatives, which (at least intuitively) means our divisor D was too small. If $reps > R$, we allocated too many representatives, which (at least intuitively) means our divisor D was too large. Finally, if $reps = R$, we can return the array $Rep[1..n]$ as the final apportionment. In practice, we can compute a valid apportionment (with $reps = R$) by calling HHGuess with a small number of integer divisors close to the *standard* divisor $D = P/R$.

In the following problems, let $P = \sum_{i=1}^{n} Pop[i]$ denote the total population of all n states, and assume that $n \le R \le P$.

(a) Show that calling HHGUESS with the standard divisor $D = P/R$ does *not* necessarily yield a valid apportionment.

(b) Prove that if HHGUESS returns the same value of *reps* for two different divisors D and D', it also computes the same allocation $Rep[1..n]$ for both of those divisors.

(c) Prove that if HHGUESS returns the correct value R, it computes the same allocation $Rep[1..n]$ as our earlier algorithm APPORTIONCONGRESS.

(d) Prove that a "correct" divisor D does not necessarily exist! That is, describe inputs $Pop[1..n]$ and R, where $n \leq R \leq P$, such that for *every* real number $D > 0$, the number of representatives allocated by HHGUESS is not equal to R. *[Hint: What happens if we change $<$ to $\leq$ in the fourth line of HHGUESS?]*

The control of a large force is the same principle as the control of a few men: it is merely a question of dividing up their numbers.

— Sun Zi, *The Art of War* (c. 400CE), translated by Lionel Giles (1910)

Our life is frittered away by detail.... Simplify, simplify.

— Henry David Thoreau, *Walden* (1854)

Now, don't ask me what Voom is. I never will know.
But, boy! Let me tell you, it DOES clean up snow!

— Dr. Seuss [Theodor Seuss Geisel], *The Cat in the Hat Comes Back* (1958)

Do the hard jobs first. The easy jobs will take care of themselves.

— attributed to Dale Carnegie

1

Recursion

1.1 Reductions

Reduction is the single most common technique used in designing algorithms. Reducing one problem X to another problem Y means to write an algorithm for X that uses an algorithm for Y as a black box or subroutine. Crucially, the correctness of the resulting algorithm for X cannot depend in any way on *how* the algorithm for Y works. The only thing we can assume is that the black box solves Y correctly. The inner workings of the black box are simply *none of our business*; they're somebody else's problem. It's often best to literally think of the black box as functioning purely by magic.

For example, the peasant multiplication algorithm described in the previous chapter reduces the problem of multiplying two arbitrary positive integers to three simpler problems: addition, mediation (halving), and parity-checking. The algorithm relies on an abstract "positive integer" data type that supports those three operations, but the correctness of the multiplication algorithm does not

depend on the precise data representation (tally marks, clay tokens, Babylonian hexagesimal, quipu, counting rods, Roman numerals, finger positions, augrym stones, gobar numerals, binary, negabinary, Gray code, balanced ternary, phinary, quater-imaginary, . . .), or on the precise implementations of those operations. Of course, the *running time* of the multiplication algorithm depends on the *running time* of the addition, mediation, and parity operations, but that's a separate issue from *correctness*. Most importantly, we can create a more efficient multiplication algorithm just by switching to a more efficient number representation (from tally marks to place-value notation, for example).

Similarly, the Huntington-Hill algorithm reduces the problem of apportioning Congress to the problem of maintaining a priority queue that supports the operations INSERT and EXTRACTMAX. The abstract data type "priority queue" is a black box; the correctness of the apportionment algorithm does not depend on any specific priority queue data structure. Of course, the *running time* of the apportionment algorithm depends on the *running time* of the INSERT and EXTRACTMAX algorithms, but that's a separate issue from the *correctness* of the algorithm. The beauty of the reduction is that we can create a more efficient apportionment algorithm by simply swapping in a new priority queue data structure. Moreover, the designer of that data structure does not need to know or care that it will be used to apportion Congress.

When we design algorithms, we may not know exactly how the basic building blocks we use are implemented, or how our algorithms might be used as building blocks to solve even bigger problems. That ignorance is uncomfortable for many beginners, but it is both unavoidable and extremely useful. Even when you do know precisely how your components work, it is often *extremely* helpful to pretend that you don't.

1.2 Simplify and Delegate

Recursion is a particularly powerful kind of reduction, which can be described loosely as follows:

- If the given instance of the problem can be solved directly, solve it directly.
- Otherwise, reduce it to one or more ***simpler instances of the same problem***.

If the self-reference is confusing, it may be helpful to imagine that someone else is going to solve the simpler problems, just as you would assume for other types of reductions. I like to call that someone else the ***Recursion Fairy***. Your *only* task is to *simplify* the original problem, or to solve it directly when simplification is either unnecessary or impossible; the Recursion Fairy will solve all the simpler subproblems for you, using Methods That Are None Of Your Business So *Butt*

Out.[1] Mathematically sophisticated readers might recognize the Recursion Fairy by its more formal name: the ***Induction Hypothesis***.

There is one mild technical condition that must be satisfied in order for any recursive method to work correctly: There must be no infinite sequence of reductions to simpler and simpler instances. Eventually, the recursive reductions must lead to an elementary ***base case*** that can be solved by some other method; otherwise, the recursive algorithm will loop forever. The most common way to satisfy this condition is to reduce to one or more ***smaller*** instances of the same problem. For example, if the original input is a skreeble with n glurps, the input to each recursive call should be a skreeble with strictly less than n glurps. Of course this is impossible if the skreeble has no glurps at all—You can't have negative glurps; that would be silly!—so in that case we must grindlebloff the skreeble using some other method.

We've already seen one instance of this pattern in the peasant multiplication algorithm, which is based directly on the following recursive identity.

$$x \cdot y = \begin{cases} 0 & \text{if } x = 0 \\ \lfloor x/2 \rfloor \cdot (y + y) & \text{if } x \text{ is even} \\ \lfloor x/2 \rfloor \cdot (y + y) + y & \text{if } x \text{ is odd} \end{cases}$$

The same recurrence can be expressed algorithmically as follows:

```
PeasantMultiply(x, y):
  if x = 0
      return 0
  else
      x' ← ⌊x/2⌋
      y' ← y + y
      prod ← PeasantMultiply(x', y')    ⟨⟨Recurse!⟩⟩
      if x is odd
          prod ← prod + y
      return prod
```

A lazy Egyptian scribe could execute this algorithm by computing x' and y', *asking a more junior scribe to multiply x' and y'*, and then possibly adding y to the junior scribe's response. The junior scribe's problem is simpler because $x' < x$, and repeatedly decreasing a positive integer eventually leads to 0. How the junior scribe actually computes $x' \cdot y'$ is none of the senior scribe's business (and it's none of your business, either).

[1]When I was an undergraduate, I attributed recursion to "elves" instead of the Recursion Fairy, referring to the Brothers Grimm story about an old shoemaker who leaves his work unfinished when he goes to bed, only to discover upon waking that elves ("Wichtelmänner") have finished everything overnight. Someone more entheogenically experienced than I might recognize these Rekursionswichtelmänner as Terence McKenna's "self-transforming machine elves".

1.3 Tower of Hanoi

The Tower of Hanoi puzzle was first published—as an actual physical puzzle!—by the French teacher and recreational mathematician Édouard Lucas in 1883,[2] under the pseudonym "N. Claus (de Siam)" (an anagram of "Lucas d'Amiens"). The following year, Henri de Parville described the puzzle with the following remarkable story:[3]

> In the great temple at Benares[4]. . . beneath the dome which marks the centre of the world, rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed sixty-four discs of pure gold, the largest disc resting on the brass plate, and the others getting smaller and smaller up to the top one. This is the Tower of Bramah. Day and night unceasingly the priests transfer the discs from one diamond needle to another according to the fixed and immutable laws of Bramah, which require that the priest on duty must not move more than one disc at a time and that he must place this disc on a needle so that there is no smaller disc below it. When the sixty-four discs shall have been thus transferred from the needle on which at the creation God placed them to one of the other needles, tower, temple, and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.

Figure 1.1. The (8-disk) Tower of Hanoi puzzle

Of course, as good computer scientists, our first instinct on reading this story is to substitute the variable n for the hardwired constant 64. And because most physical instances of the puzzle are made of wood instead of diamonds and gold, I will call the three possible locations for the disks "pegs" instead of

[2]Lucas later claimed to have invented the puzzle in 1876.

[3]This English translation is taken from W. W. Rouse Ball's 1892 book *Mathematical Recreations and Essays*.

[4]The "great temple at Benares" is almost certainly the Kashi Vishvanath Temple in Varanasi, Uttar Pradesh, India, located approximately 2400km west-north-west of Hà Nội, Việt Nam, where the fictional N. Claus supposedly resided. Coincidentally, the French Army invaded Hanoi in 1883, the same year Lucas released his puzzle, ultimately leading to its establishment as the capital of French Indochina.

"needles". How can we move a tower of n disks from one peg to another, using a third spare peg as an occasional placeholder, without ever placing a disk on top of a smaller disk?

As N. Claus (de Siam) pointed out in the pamphlet included with his puzzle, the secret to solving this puzzle is to think recursively. Instead of trying to solve the entire puzzle at once, let's concentrate on moving just the largest disk. We can't move it at the beginning, because all the other disks are in the way. So first we have to move those $n-1$ smaller disks to the spare peg. Once that's done, we can move the largest disk directly to its destination. Finally, to finish the puzzle, we have to move the $n-1$ smaller disks from the spare peg to their destination.

Figure 1.2. The Tower of Hanoi algorithm; ignore everything but the bottom disk.

So now all we have to figure out is how to—

NO!! STOP!!

That's it! We're done! We've successfully reduced the n-disk Tower of Hanoi problem to two instances of the $(n-1)$-disk Tower of Hanoi problem, which we can gleefully hand off to the Recursion Fairy—or to carry Lucas's metaphor further, to the junior monks at the temple. *Our* job is finished. If we didn't trust the junior monks, we wouldn't have hired them; let them do their job in peace.

Our reduction does make one subtle but extremely important assumption: *There is a largest disk*. Our recursive algorithm works for any *positive* number of disks, but it breaks down when $n = 0$. We must handle that case using a different method. Fortunately, the monks at Benares, being good Buddhists, are quite adept at moving zero disks from one peg to another in no time at all, by doing nothing.

Figure 1.3. The vacuous base case for the Tower of Hanoi algorithm. There is no spoon.

It may be tempting to think about how all those smaller disks move around—or more generally, what happens when the recursion is unrolled—but really, don't do it. For most recursive algorithms, unrolling the recursion is neither necessary nor helpful. Our ***only*** task is to reduce the problem instance we're given to one or more simpler instances, or to solve the problem directly if such a reduction is impossible. Our recursive Tower of Hanoi algorithm is trivially correct when $n = 0$. For any $n \geq 1$, the Recursion Fairy correctly moves the top $n - 1$ disks (more formally, the Inductive Hypothesis implies that our recursive algorithm correctly moves the top $n - 1$ disks) so our algorithm is correct.

The recursive Hanoi algorithm is expressed in pseudocode in Figure 1.4. The algorithm moves a stack of n disks from a source peg (*src*) to a destination peg (*dst*) using a third temporary peg (*tmp*) as a placeholder. Notice that the algorithm correctly does nothing at all when $n = 0$.

```
HANOI(n, src, dst, tmp):
  if n > 0
      HANOI(n − 1, src, tmp, dst)     ⟨⟨Recurse!⟩⟩
      move disk n from src to dst
      HANOI(n − 1, tmp, dst, src)     ⟨⟨Recurse!⟩⟩
```

Figure 1.4. A recursive algorithm to solve the Tower of Hanoi

Let $T(n)$ denote the number of moves required to transfer n disks—the running time of our algorithm. Our vacuous base case implies that $T(0) = 0$, and the more general recursive algorithm implies that $T(n) = 2T(n-1) + 1$ for any $n \geq 1$. By writing out the first several values of $T(n)$, we can easily guess that $\boldsymbol{T(n) = 2^n - 1}$; a straightforward induction proof implies that this guess is correct. In particular, moving a tower of 64 disks requires $2^{64} - 1 =$ 18,446,744,073,709,551,615 individual moves. Thus, even at the impressive rate of one move per second, the monks at Benares will be at work for approximately 585 billion years ("plus de *cinq milliards de siècles*") before tower, temple, and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.

1.4 Mergesort

Mergesort is one of the earliest algorithms designed for general-purpose stored-program computers. The algorithm was developed by John von Neumann in 1945, and described in detail in a publication with Herman Goldstine in 1947, as one of the first non-numerical programs for the EDVAC.[5]

[5]Goldstine and von Neumann actually described an non-recursive variant now usually called bottom-up mergesort. At the time, large data sets were sorted by special-purpose machines—almost all built by IBM—that manipulated punched cards using variants of binary radix sort. Von

1. Divide the input array into two subarrays of roughly equal size.
2. Recursively mergesort each of the subarrays.
3. Merge the newly-sorted subarrays into a single sorted array.

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Divide:	S	O	R	T	I	N	G	E	X	A	M	P	L
Recurse Left:	I	N	O	R	S	T	G	E	X	A	M	P	L
Recurse Right:	I	N	O	R	S	T	A	E	G	L	M	P	X
Merge:	A	E	G	I	L	M	N	O	P	R	S	T	X

Figure 1.5. A mergesort example.

The first step is completely trivial—just divide the array size by two—and we can delegate the second step to the Recursion Fairy. All the real work is done in the final merge step. A complete description of the algorithm is given in Figure 1.6; to keep the recursive structure clear, I've extracted the merge step into an independent subroutine. The merge algorithm is also recursive—identify the first element of the output array, and then recursively merge the rest of the input arrays.

```
MergeSort(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MergeSort(A[1..m])          ⟨⟨Recurse!⟩⟩
      MergeSort(A[m+1..n])        ⟨⟨Recurse!⟩⟩
      Merge(A[1..n], m)
```

```
Merge(A[1..n], m):
  i ← 1;  j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i+1
      else if i > m
          B[k] ← A[j];  j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i+1
      else
          B[k] ← A[j];  j ← j+1
  for k ← 1 to n
      A[k] ← B[k]
```

Figure 1.6. Mergesort

Correctness

To prove that this algorithm is correct, we apply our old friend induction twice, first to the Merge subroutine then to the top-level Mergesort algorithm.

Lemma 1.1. *Merge correctly merges the subarrays $A[1..m]$ and $A[m+1..n]$, assuming those subarrays are sorted in the input.*

Neumann argued (successfully!) that because the EDVAC could sort faster than IBM's dedicated sorters, "without human intervention or need for additional equipment", the EDVAC was an "all purpose" machine, and special-purpose sorting machines were no longer necessary.

Proof: Let $A[1..n]$ be any array and m any integer such that the subarrays $A[1..m]$ and $A[m+1..n]$ are sorted. We prove that for all k from 0 to n, the last $n-k-1$ iterations of the main loop correctly merge $A[i..m]$ and $A[j..n]$ into $B[k..n]$. The proof proceeds by induction on $n-k+1$, the number of elements remaining to be merged.

If $k > n$, the algorithm correctly merges the two empty subarrays by doing absolutely nothing. (This is the base case of the inductive proof.) Otherwise, there are four cases to consider for the kth iteration of the main loop.

- If $j > n$, then subarray $A[j..n]$ is empty, so $\min\big(A[i..m] \cup A[j..n]\big) = A[i]$.
- If $i > m$, then subarray $A[i..m]$ is empty, so $\min\big(A[i..m] \cup A[j..n]\big) = A[j]$.
- Otherwise, if $A[i] < A[j]$, then $\min\big(A[i..m] \cup A[j..n]\big) = A[i]$.
- Otherwise, we must have $A[i] \geq A[j]$, and $\min\big(A[i..m] \cup A[j..n]\big) = A[j]$.

In all four cases, $B[k]$ is correctly assigned the smallest element of $A[i..m] \cup A[j..n]$. In the two cases with the assignment $B[k] \leftarrow A[i]$, the Recursion Fairy correctly merges—sorry, I mean the Induction Hypothesis implies that the last $n-k$ iterations of the main loop correctly merge $A[i+1..m]$ and $A[j..n]$ into $B[k+1..n]$. Similarly, in the other two cases, the Recursion Fairy also correctly merges the rest of the subarrays. □

Theorem 1.2. *MergeSort correctly sorts any input array $A[1..n]$.*

Proof: We prove the theorem by induction on n. If $n \leq 1$, the algorithm correctly does nothing. Otherwise, the Recursion Fairy correctly sorts—sorry, I mean the induction hypothesis implies that our algorithm correctly sorts the two smaller subarrays $A[1..m]$ and $A[m+1..n]$, after which they are correctly Merged into a single sorted array (by Lemma 1.1). □

Analysis

Because the MergeSort algorithm is recursive, its running time is naturally expressed as a recurrence. Merge clearly takes $O(n)$ time, because it's a simple for-loop with constant work per iteration. We immediately obtain the following recurrence for MergeSort:

$$T(n) = T\big(\lceil n/2 \rceil\big) + T\big(\lfloor n/2 \rfloor\big) + O(n).$$

As in most divide-and-conquer recurrences, we can safely strip out the floors and ceilings (using a technique called *domain transformations* described later in this chapter), giving us the simpler recurrence $T(n) = 2T(n/2) + O(n)$. The "all levels equal" case of the recursion tree method (also described later in this chapter) immediately implies the closed-form solution $\boldsymbol{T(n) = O(n \log n)}$. Even if you are not (yet) familiar with recursion trees, you can verify the solution $T(n) = O(n \log n)$ by induction.

1.5 Quicksort

Quicksort is another recursive sorting algorithm, discovered by Tony Hoare in 1959 and first published in 1961. In this algorithm, the hard work is splitting the array into smaller subarrays *before* recursion, so that merging the sorted subarrays is trivial.

1. Choose a *pivot* element from the array.
2. Partition the array into three subarrays containing the elements smaller than the pivot, the pivot element itself, and the elements larger than the pivot.
3. Recursively quicksort the first and last subarrays.

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Choose a pivot:	S	O	R	T	I	N	G	E	X	A	M	P	L
Partition:	A	G	O	E	I	N	L	M	P	T	X	S	R
Recurse Left:	A	E	G	I	L	M	N	O	P	T	X	S	R
Recurse Right:	A	E	G	I	L	M	N	O	P	R	S	T	X

Figure 1.7. A quicksort example.

More detailed pseudocode is given in Figure 1.8. In the PARTITION subroutine, the input parameter p is the index of the pivot element in the unsorted array; the subroutine partitions the array and returns the new index of the pivot element. There are many different efficient partitioning algorithms; the one I'm presenting here is attributed to Nico Lomuto.[6] The variable ℓ counts the number of items in the array that are ℓess than the pivot element.

```
QUICKSORT(A[1..n]):
  if (n > 1)
      Choose a pivot element A[p]
      r ← PARTITION(A, p)
      QUICKSORT(A[1..r − 1])      ⟨⟨Recurse!⟩⟩
      QUICKSORT(A[r + 1..n])      ⟨⟨Recurse!⟩⟩
```

```
PARTITION(A[1..n], p):
  swap A[p] ↔ A[n]
  ℓ ← 0                  ⟨⟨#items < pivot⟩⟩
  for i ← 1 to n − 1
      if A[i] < A[n]
          ℓ ← ℓ + 1
          swap A[ℓ] ↔ A[i]
  swap A[n] ↔ A[ℓ + 1]
  return ℓ + 1
```

Figure 1.8. Quicksort

Correctness

Just like mergesort, proving that QUICKSORT is correct requires two separate induction proofs: one to prove that PARTITION correctly partitions the array, and

[6] Hoare proposed a more complicated "two-way" partitioning algorithm that has some practical advantages over Lomuto's algorithm. On the other hand, Hoare's partitioning algorithm is one of the places off-by-one errors go to die.

the other to prove that QuickSort correctly sorts *assuming* Partition is correct. To prove Partition is correct, we need to prove the following loop invariant: At the end of each iteration of the main loop, everything in the subarray $A[1..\ell]$ is ℓess than $A[n]$, and nothing in the subarray $A[\ell+1..i]$ is less than $A[n]$. I'll leave the remaining straightforward but tedious details as exercises for the reader.

Analysis

The analysis of quicksort is also similar to that of mergesort. Partition clearly runs in $O(n)$ time, because it's a simple for-loop with constant work per iteration. For QuickSort, we get a recurrence that depends on r, the *rank* of the chosen pivot element:

$$T(n) = T(r-1) + T(n-r) + O(n)$$

If we could somehow always magically choose the pivot to be the *median* element of the array A, we would have $r = \lceil n/2 \rceil$, the two subproblems would be as close to the same size as possible, the recurrence would become

$$T(n) \;=\; T\big(\lceil n/2 \rceil - 1\big) + T\big(\lfloor n/2 \rfloor\big) + O(n) \;\leq\; 2T(n/2) + O(n),$$

and we'd have $T(n) = O(n \log n)$ using either the recursion tree method or the even simpler "Oh yeah, we already solved that recurrence for mergesort" method.

In fact, as we will see later in this chapter, we *can* actually locate the median element in an unsorted array in linear time, but the algorithm is fairly complicated, and the hidden constant in the $O(\cdot)$ notation is large enough to make the resulting sorting algorithm impractical. In practice, most programmers settle for something simple, like choosing the first or last element of the array. In this case, r can take any value between 1 and n, so we have

$$T(n) = \max_{1 \leq r \leq n} \big(T(r-1) + T(n-r) + O(n)\big).$$

In the worst case, the two subproblems are completely unbalanced—either $r = 1$ or $r = n$—and the recurrence becomes $T(n) \leq T(n-1) + O(n)$. The solution is $\boldsymbol{T(n) = O(n^2)}$.

Another common heuristic is called "median of three"—choose three elements (usually at the beginning, middle, and end of the array), and take the median of those three elements as the pivot. Although this heuristic is somewhat more efficient in practice than just choosing one element, especially when the array is already (nearly) sorted, we can still have $r = 2$ or $r = n-1$ in the worst case. With the median-of-three heuristic, the recurrence becomes $T(n) \leq T(1) + T(n-2) + O(n)$, whose solution is still $T(n) = O(n^2)$.

Intuitively, the pivot element should "usually" fall somewhere in the middle of the array, say with rank between $n/10$ and $9n/10$. This observation suggests that the "average-case" running time should be $O(n \log n)$. Although this intuition can be formalized, the most common formalization makes the completely unrealistic assumption that all permutations of the input array are equally likely. Real world data *may* be random, but it is not random in any way that we can predict in advance, and it is *certainly* not uniform![7]

Occasionally people also consider "best case" running time for some reason. We won't.

1.6 The Pattern

Both mergesort and quicksort follow a general three-step pattern called ***divide and conquer***:

1. **Divide** the given instance of the problem into several *independent smaller* instances of *exactly* the same problem.
2. **Delegate** each smaller instance to the Recursion Fairy.
3. **Combine** the solutions for the smaller instances into the final solution for the given instance.

If the size of any instance falls below some constant threshold, we abandon recursion and solve the problem directly, by brute force, in constant time.

Proving a divide-and-conquer algorithm correct almost always requires induction. Analyzing the running time requires setting up and solving a recurrence, which usually (but unfortunately not always!) can be solved using recursion trees.

1.7 Recursion Trees

So what are these "recursion trees" I keep talking about? Recursion trees are a simple, general, pictorial tool for solving divide-and-conquer recurrences. A recursion tree is a rooted tree with one node for each recursive subproblem. The *value* of each node is the amount of time spent on the corresponding subproblem *excluding* recursive calls. Thus, the overall running time of the algorithm is the sum of the values of all nodes in the tree.

To make this idea more concrete, imagine a divide-and-conquer algorithm that spends $O(f(n))$ time on non-recursive work, and then makes r recursive

[7] On the other hand, if we choose the pivot index p uniformly at random, then QUICKSORT runs in $O(n \log n)$ time *with high probability*, for *every* possible input array. The key difference is that the randomness is controlled by our algorithm, not by the All-Powerful Malicious Adversary who gives us input data after reading our code. The analysis of randomized quicksort is unfortunately outside the scope of this book, but you can find relevant lecture notes at http://algorithms.wtf/.

calls, each on a problem of size n/c. Up to constant factors (which we can hide in the $O(\,)$ notation), the running time of this algorithm is governed by the recurrence

$$T(n) = r\,T(n/c) + f(n).$$

The root of the recursion tree for $T(n)$ has value $f(n)$ and r children, each of which is the root of a (recursively defined) recursion tree for $T(n/c)$. Equivalently, a recursion tree is a complete r-ary tree where each node at depth d contains the value $f(n/c^d)$. (Feel free to assume that n is an integer power of c, so that n/c^d is always an integer, although in fact this doesn't matter.)

In practice, I recommend drawing out the first two or three levels of the tree, as in Figure 1.9.

Figure 1.9. A recursion tree for the recurrence $T(n) = r\,T(n/c) + f(n)$

The leaves of the recursion tree correspond to the base case(s) of the recurrence. Because we're only looking for asymptotic bounds, the precise base case doesn't actually matter; we can safely assume $T(n) = 1$ for all $n \le n_0$, where n_0 is an arbitrary positive constant. In particular, we can choose whatever value of n_0 is most convenient for our analysis. For this example, I'll choose $n_0 = 1$.

Now $T(n)$ is the sum of all values in the recursion tree; we can evaluate this sum by considering the tree level-by-level. For each integer i, the ith level of the tree has exactly r^i nodes, each with value $f(n/c^i)$. Thus,

$$T(n) = \sum_{i=0}^{L} r^i \cdot f(n/c^i) \tag{Σ}$$

where L is the depth of the tree. Our base case $n_0 = 1$ immediately implies $\boldsymbol{L = \log_c n}$, because $n/c^L = n_0 = 1$. It follows that the number of leaves in

the recursion tree is exactly $r^L = r^{\log_c n} = n^{\log_c r}$. Thus, the last term in the level-by-level sum (Σ) is $n^{\log_c r} \cdot f(1) = O(n^{\log_c r})$, because $f(1) = O(1)$.

There are three common cases where the level-by-level series (Σ) is especially easy to evaluate:

- **Decreasing:** If the series *decays exponentially*—every term is a constant factor smaller than the previous term—then $T(n) = O(f(n))$. In this case, the sum is dominated by the value at the root of the recursion tree.
- **Equal:** If all terms in the series are equal, we immediately have $T(n) = O(f(n) \cdot L) = O(f(n) \log n)$. (The constant c vanishes into the $O(\,)$ notation.)
- **Increasing:** If the series *grows exponentially*—every term is a constant factor larger than the previous term—then $T(n) = O(n^{\log_c r})$. In this case, the sum is dominated by the number of leaves in the recursion tree.

In the first and third cases, only the largest term in the geometric series matters; all other terms are swallowed up by the $O(\cdot)$ notation. In the decreasing case, we don't even have to compute L; the asymptotic upper bound would still hold if the recursion tree were infinite!

As an elementary example, if we draw out the first few levels of the recursion tree for the (simplified) mergesort recurrence $T(n) = 2T(n/2) + O(n)$, we discover that all levels are equal, which immediately implies $T(n) = O(n \log n)$.

Figure 1.10. The recursion tree for mergesort

The recursion tree technique can also be used for algorithms where the recursive subproblems have different sizes. For example, if we could somehow implement quicksort so that the pivot *always* lands in the middle third of the sorted array, the worst-case running time would satisfy the recurrence

$$T(n) \leq T(n/3) + T(2n/3) + O(n).$$

This recurrence might look scary, but it's actually pretty tame. If we draw out a few levels of the resulting recursion tree, we quickly realize that the sum of values on any level is *at most* n—deeper levels might be missing some nodes—and the entire tree has depth $\log_{3/2} n = O(\log n)$. It immediately follows that $T(n) = O(n \log n)$. (Moreover, the number of *full* levels in the recursion

tree is $\log_3 n = \Omega(\log n)$, so this conservative analysis can be improved by at most a constant factor, which for our purposes means not at all.) The fact that the recursion tree is unbalanced simply doesn't matter.

As a more extreme example, the worst-case recurrence for quicksort $T(n) = T(n-1) + T(1) + O(n)$ gives us a completely unbalanced recursion tree, where one child of each internal node is a leaf. The level-by-level sum doesn't fall into any of our three default categories, but we can still derive the solution $T(n) = O(n^2)$ by observing that every level value is at most n and there are at most n levels. (Again, this conservative analysis is tight, because $n/2$ levels each have value at least $n/2$.)

Figure 1.11. Recursion trees for quicksort with good pivots (left) and with worst-case pivots (right)

♥Ignoring Floors and Ceilings Is Okay, Honest

Careful readers might object that our analysis brushes an important detail under the rug. The running time of mergesort doesn't *really* obey the recurrence $T(n) = 2T(n/2) + O(n)$; after all, the input size n might be odd, and what could it possibly mean to sort an array of size $42\frac{1}{2}$ or $17\frac{7}{8}$? The actual mergesort recurrence is somewhat messier:

$$T(n) = T\big(\lceil n/2 \rceil\big) + T\big(\lfloor n/2 \rfloor\big) + O(n).$$

Sure, we could check that $T(n) = O(n \log n)$ using induction, but the necessary calculations would be awful. Fortunately, there is a simple technique for removing floors and ceilings from recurrences, called *domain transformation*.

- First, because we are deriving an upper bound, we can safely overestimate $T(n)$, once by pretending that the two subproblem sizes are equal, and again to eliminate the ceiling:[8]

$$T(n) \;\le\; 2T\big(\lceil n/2 \rceil\big) + n \;\le\; 2T(n/2 + 1) + n.$$

[8]Formally, we are treating T as a function over the *reals*, not just over the integers, that satisfies the given recurrence with the base case $T(n) = C$ for all $n \le n_0$, for some real numbers $C \ge 0$ and $n_0 > 0$ whose values don't matter. If n happens to be an integer, then $T(n)$ coincides with the running time of an algorithm on an input of size n, but that doesn't matter, either.

- Second, we define a new function $S(n) = T(n+\alpha)$, choosing the constant α so that $S(n)$ satisfies the simpler recurrence $S(n) \leq 2S(n/2) + O(n)$. To find the correct constant α, we derive a recurrence for S from our given recurrence for T:

$$\begin{aligned} S(n) &= T(n+\alpha) && \text{[definition of } S\text{]} \\ &\leq 2T(n/2+\alpha/2+1)+n+\alpha && \text{[recurrence for } T\text{]} \\ &= 2S(n/2-\alpha/2+1)+n+\alpha && \text{[definition of } S\text{]} \end{aligned}$$

  Setting $\alpha = 2$ simplifies this recurrence to $S(n) \leq 2S(n/2) + n + 2$, which is exactly what we wanted.

- Finally, the recursion tree method implies $S(n) = O(n \log n)$, and therefore

$$T(n) = S(n-2) = O((n-2)\log(n-2)) = O(n \log n),$$

  exactly as promised.

Similar domain transformations can be used to remove floors, ceilings, and even lower order terms from *any* divide and conquer recurrence. But now that we realize this, we don't need to bother grinding through the details ever again! From now on, faced with any divide-and-conquer recurrence, I will silently brush floors and ceilings and lower-order terms under the rug, and I encourage you to do the same.

♥1.8 Linear-Time Selection

During our discussion of quicksort, I claimed in passing that we can find the median of an unsorted array in linear time. The first such algorithm was discovered by Manuel Blum, Bob Floyd, Vaughan Pratt, Ron Rivest, and Bob Tarjan in the early 1970s. Their algorithm actually solves the more general problem of selecting the kth smallest element in an n-element array, given the array and the integer k as input, using a variant of an algorithm called *quickselect* or *one-armed quicksort*. Quickselect was first described by Tony Hoare in 1961, literally on the same page where he first published quicksort.

Quickselect

The generic quickselect algorithm chooses a pivot element, partitions the array using the same Partition subroutine as QuickSort, and then recursively searches *only one* of the two subarrays, specifically, the one that contains the kth smallest element of the original input array. Pseudocode for quickselect is given in Figure 1.12.

```
QuickSelect(A[1..n], k):
  if n = 1
      return A[1]
  else
      Choose a pivot element A[p]
      r ← Partition(A[1..n], p)
      if k < r
          return QuickSelect(A[1..r−1], k)
      else if k > r
          return QuickSelect(A[r+1..n], k−r)
      else
          return A[r]
```

Figure 1.12. Quickselect, or one-armed quicksort

This algorithm has two important features. First, just like quicksort, the correctness of quickselect does not depend on how the pivot is chosen. Second, even if we really only care about selecting *medians* (the special case $k = n/2$), Hoare's recursive strategy requires us to consider the more general selection problem; the median of the input array $A[1..n]$ is almost never the median of either of the two smaller subarrays $A[1..r-1]$ or $A[r+1..n]$.

The worst-case running time of QuickSelect obeys a recurrence similar to QuickSort. We don't know the value of r, or which of the two subarrays we'll recursively search, so we have to assume the worst.

$$T(n) \leq \max_{1 \leq r \leq n} \max\{T(r-1), T(n-r)\} + O(n)$$

We can simplify the recurrence slightly by letting ℓ denote the length of the recursive subproblem:

$$T(n) \leq \max_{0 \leq \ell \leq n-1} T(\ell) + O(n)$$

If the chosen pivot element is always either the smallest or largest element in the array, the recurrence simplifies to $T(n) = T(n-1) + O(n)$, which implies $T(n) = O(n^2)$. (The recursion tree for this recurrence is just a simple path.)

Good pivots

We could avoid this quadratic worst-case behavior if we could *somehow* magically choose a ***good*** pivot, meaning $\ell \leq \alpha n$ for some constant $\alpha < 1$. In this case, the recurrence would simplify to

$$T(n) \leq T(\alpha n) + O(n).$$

This recurrence expands into a decreasing geometric series, which is dominated by its largest term, so $T(n) = O(n)$. (Again, the recursion tree is just a simple path. The constant in the $O(n)$ running time depends on the constant α.)

In other words, if we could somehow quickly find an element that's even *close* to the median in linear time, we could find the *exact* median in linear time. So now all we need is an Approximate Median Fairy. The Blum-Floyd-Pratt-Rivest-Tarjan algorithm chooses a good quickselect pivot by ***recursively*** computing the median of a carefully-chosen subset of the input array. The Approximate Median Fairy is just the Recursion Fairy in disguise!

Specifically, we divide the input array into $\lceil n/5 \rceil$ *blocks*, each containing exactly 5 elements, except possibly the last. (If the last block isn't full, just throw in a few ∞s.) We compute the median of each block by brute force, collect those medians into a new array $M[1..\lceil n/5 \rceil]$, and then *recursively* compute the median of this new array. Finally, we use the median of the block medians (called "*mom*" in the pseudocode below) as the quickselect pivot.

```
MomSelect(A[1..n], k):
  if n ≤ 25   ⟨⟨or whatever⟩⟩
      use brute force
  else
      m ← ⌈n/5⌉
      for i ← 1 to m
          M[i] ← MedianOfFive(A[5i − 4..5i])   ⟨⟨Brute force!⟩⟩
      mom ← MomSelect(M[1..m], ⌊m/2⌋)          ⟨⟨Recursion!⟩⟩

      r ← Partition(A[1..n], mom)

      if k < r
          return MomSelect(A[1..r − 1], k)     ⟨⟨Recursion!⟩⟩
      else if k > r
          return MomSelect(A[r + 1..n], k − r) ⟨⟨Recursion!⟩⟩
      else
          return mom
```

MomSelect uses recursion for two different purposes; the first time to choose a pivot element (*mom*), and the second time to search through the entries on one side of that pivot.

Analysis

But why is this fast? The first key insight is that ***the median of medians is a good pivot***. Mom is larger than $\lfloor \lceil n/5 \rceil / 2 \rfloor - 1 \approx n/10$ block medians, and each block median is larger than two other elements in its block. Thus, mom is bigger than at least $3n/10$ elements in the input array; symmetrically, mom is smaller than at least $3n/10$ elements. Thus, in the worst case, the second recursive call searches an array of size at most $7n/10$.

We can visualize the algorithm's behavior by drawing the input array as a $5 \times \lceil n/5 \rceil$ grid, which each column represents five consecutive elements. For purposes of illustration, imagine that we sort every column from top down, and then we sort the columns by their middle element. (Let me emphasize that *the algorithm does not actually do this!*) In this arrangement, the median-of-medians is the element closest to the center of the grid.

The left half of the first three rows of the grid contains $3n/10$ elements, each of which is smaller than mom. If the element we're looking for is larger than mom, our algorithm will throw away *everything* smaller than mom, including those $3n/10$ elements, before recursing. Thus, the input to the recursive subproblem contains at most $7n/10$ elements. A symmetric argument implies that if our target element is smaller than mom, we discard at least $3n/10$ elements larger than mom, so the input to our recursive subproblem has at most $7n/10$ elements.

Okay, so mom is a good pivot, but our algorithm still makes *two* recursive calls instead of just one; how do we prove linear time? The second key insight is that the *total* size of the two recursive subproblems is a constant factor smaller than the size of the original input array. The worst-case running time of the algorithm obeys the recurrence

$$T(n) \leq T(n/5) + T(7n/10) + O(n).$$

If we draw out the recursion tree for this recurrence, we observe that the total work at each level of the recursion tree is at most $9/10$ the total work at the previous level. Thus, the level sums decay exponentially, giving us the solution $T(n) = O(n)$. (Again, the fact that the recursion tree is unbalanced is completely immaterial.) Hooray! Thanks, Mom!

Figure 1.13. The recursion trees for MomSelect and a similar selection algorithm with blocks of size 3

Sanity Checking

At this point, many students ask about that magic constant 5. Why did we choose that particular block size? The answer is that 5 is the smallest odd block size that gives us exponential decay in the recursion-tree analysis! (Even block sizes introduce additional complications.) If we had used blocks of size 3 instead, the running-time recurrence would be

$$T(n) \leq T(n/3) + T(2n/3) + O(n).$$

We've seen this recurrence before! Every level of the recursion tree has total value *at most* n, and the depth of the recursion tree is $\log_{3/2} n = O(\log n)$, so the solution to this recurrence is $T(n) \leq O(n \log n)$. (Moreover, this analysis is tight, because the recursion tree has $\log_3 n$ complete levels.) Median-of-medians selection using 3-element blocks is no faster than sorting.

Finer analysis reveals that the constant hidden by the $O(\,)$ notation is quite large, even if we count only comparisons. Selecting the median of 5 elements requires at most 6 comparisons, so we need at most $6n/5$ comparisons to set up the recursive subproblem. Naïvely partitioning the array after the recursive call would require $n-1$ comparisons, but we already know $3n/10$ elements larger than the pivot and $3n/10$ elements smaller than the pivot, so partitioning actually requires only $2n/5$ additional comparisons. Thus, a more precise recurrence for the worst-case number of comparisons is

$$T(n) \leq T(n/5) + T(7n/10) + 8n/5.$$

The recursion tree method implies the upper bound

$$T(n) \leq \frac{8n}{5} \sum_{i \geq 0} \left(\frac{9}{10}\right)^i = \frac{8n}{5} \cdot 10 = 16n.$$

In practice, median-of-medians selection is not as slow as this worst-case analysis predicts—getting a worst-case pivot at every level of recursion is incredibly unlikely—but it is still slower than sorting for even moderately large arrays.[9]

[9] In fact, the right way to choose the pivot element in practice is to choose it *uniformly at random*. Then the expected number of comparisons required to find the median is at most $4n$. See my randomized algorithms lecture notes at http://algorithms.wtf for more details.

1.9 Fast Multiplication

In the previous chapter, we saw two ancient algorithms for multiplying two n-digit numbers in $O(n^2)$ time: the grade-school lattice algorithm and the Egyptian peasant algorithm.

Maybe we can get a more efficient algorithm by splitting the digit arrays in half and exploiting the following identity:

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd$$

This recurrence immediately suggests the following divide-and-conquer algorithm to multiply two n-digit numbers x and y. Each of the four sub-products ac, bc, ad, and bd is computed recursively, but the multiplications in the last line are *not* recursive, because we can multiply by a power of ten by shifting the digits to the left and filling in the correct number of zeros, all in $O(n)$ time.

```
SplitMultiply(x, y, n):
  if n = 1
      return x · y
  else
      m ← ⌈n/2⌉
      a ← ⌊x/10^m⌋;  b ← x mod 10^m        ⟨⟨x = 10^m a + b⟩⟩
      c ← ⌊y/10^m⌋;  d ← y mod 10^m        ⟨⟨y = 10^m c + d⟩⟩
      e ← SplitMultiply(a, c, m)
      f ← SplitMultiply(b, d, m)
      g ← SplitMultiply(b, c, m)
      h ← SplitMultiply(a, d, m)
      return 10^(2m) e + 10^m (g + h) + f
```

Correctness of this algorithm follows easily by induction. The running time for this algorithm follows the recurrence

$$T(n) = 4T(\lceil n/2 \rceil) + O(n).$$

The recursion tree method transforms this recurrence into an *increasing* geometric series, which implies $T(n) = O(n^{\log_2 4}) = O(n^2)$. In fact, this algorithm multiplies each digit of x with each digit of y, just like the lattice algorithm. So I guess that didn't work. Too bad. It was a nice idea.

In the mid-1950s, Andrei Kolmogorov, one of the giants of 20th century mathematics, publicly conjectured that there is *no* algorithm to multiply two n-digit numbers in subquadratic time. Kolmogorov organized a seminar at Moscow University in 1960, where he restated his "n^2 conjecture" and posed several related problems that he planned to discuss at future meetings. Almost exactly a week later, a 23-year-old student named Anatoliĭ Karatsuba presented Kolmogorov with a remarkable counterexample. According to Karatsuba himself,

Figure 1.14. The recursion tree for naïve divide-and-conquer multiplication

> After the seminar I told Kolmogorov about the new algorithm and about the disproof of the n^2 conjecture. Kolmogorov was very agitated because this contradicted his very plausible conjecture. At the next meeting of the seminar, Kolmogorov himself told the participants about my method, and at that point the seminar was terminated.

Karatsuba observed that the middle coefficient $bc+ad$ can be computed from the other two coefficients ac and bd using only *one* more recursive multiplication, via the following algebraic identity:

$$ac + bd - (a-b)(c-d) = bc + ad$$

This trick lets us replace the four recursive calls in the previous algorithm with only three recursive calls, as shown below:

```
FastMultiply(x, y, n):
  if n = 1
      return x · y
  else
      m ← ⌈n/2⌉
      a ← ⌊x/10^m⌋;  b ← x mod 10^m        ⟨⟨x = 10^m a + b⟩⟩
      c ← ⌊y/10^m⌋;  d ← y mod 10^m        ⟨⟨y = 10^m c + d⟩⟩
      e ← FastMultiply(a, c, m)
      f ← FastMultiply(b, d, m)
      g ← FastMultiply(a − b, c − d, m)
      return 10^(2m) e + 10^m (e + f − g) + f
```

The running time of Karatsuba's FastMultiply algorithm follows the recurrence

$$T(n) \leq 3T(\lceil n/2 \rceil) + O(n)$$

Once again, the recursion tree method transforms this recurrence into an increasing geometric series, but the new solution is only $T(n) = O(n^{\log_2 3}) = \boldsymbol{O(n^{1.58496})}$, a significant improvement over our earlier quadratic time bound.[10]

[10] My presentation simplifies the actual history slightly. In fact, Karatsuba proposed an algorithm based on the formula $(a+b)(c+d) - ac - bd = bc + ad$. This algorithm also runs in $O(n^{\lg 3})$ time, but the actual recurrence is slightly messier: $a-b$ and $c-d$ are still m-digit numbers, but $a+b$ and $c+d$ might each have $m+1$ digits. The simplification presented here is due to Donald Knuth.

Karatsuba's algorithm arguably launched the design and analysis of algorithms as a formal field of study.

Figure 1.15. The recursion tree for Karatsuba's divide-and-conquer multiplication algorithm

We can take Karatsuba's idea even further, splitting the numbers into more pieces and combining them in more complicated ways, to obtain even faster multiplication algorithms. Andrei Toom discovered an infinite family of algorithms that split any integer into k parts, each with n/k digits, and then compute the product using only $2k-1$ recursive multiplications; Toom's algorithms were further simplified by Stephen Cook in his PhD thesis. For any fixed k, the Toom-Cook algorithm runs in $O(n^{1+1/(\lg k)})$ time, where the hidden constant in the $O(\cdot)$ notation depends on k.

Ultimately, this divide-and-conquer strategy led Gauss (yes, really) to the discovery of the ***Fast Fourier transform***.[11] The basic FFT algorithm itself runs in $O(n \log n)$ time; however, using FFTs for integer multiplication incurs some small additional overhead. The first FFT-based integer multiplication algorithm, published by Arnold Schönhage and Volker Strassen in 1971, runs in $O(n \log n \log\log n)$ time. Schönhage-Strassen remained the theoretically fastest integer multiplication algorithm for several decades, before Martin Fürer discovered the first of a long series of technical improvements. Finally, in 2019, David Harvey and Joris van der Hoeven published an algorithm that runs in $O(n \log n)$ time.[12]

1.10 Exponentiation

Given a number a and a positive integer n, suppose we want to compute a^n. The standard naïve method is a simple for-loop that performs $n-1$ multiplications by a:

[11] See http://algorithms.wtf for lecture notes on Fast Fourier transforms.

[12] Schönhage-Strassen is actually the fastest algorithm *in practice* for multiplying integers with more than about 75000 digits; the more recent algorithms of Fürer, Harvey, van der Hoeven, and others would be faster "in practice" only for integers with more digits than there are particles in the universe.

```
SlowPower(a, n):
  x ← a
  for i ← 2 to n
      x ← x · a
  return x
```

This iterative algorithm requires n multiplications.

The input parameter a could be an integer, or a rational, or a floating point number. In fact, it doesn't need to be a number at all, as long as it's something that we know how to multiply. For example, the same algorithm can be used to compute powers modulo some finite number (an operation commonly used in cryptography algorithms) or to compute powers of matrices (an operation used to evaluate recurrences and to compute shortest paths in graphs). Because we don't know what kind of object we're multiplying, we *can't* know how much time a single multiplication requires, so we're forced to analyze the running time in terms of the number of multiplications.

There is a much faster divide-and-conquer method, originally proposed by the Indian prosodist Piṅgala in the 2nd century BCE, which uses the following simple recursive formula:

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ (a^{n/2})^2 & \text{if } n > 0 \text{ and } n \text{ is even} \\ (a^{\lfloor n/2 \rfloor})^2 \cdot a & \text{otherwise} \end{cases}$$

```
PiṅgalaPower(a, n):
  if n = 1
      return a
  else
      x ← PiṅgalaPower(a, ⌊n/2⌋)
      if n is even
          return x · x
      else
          return x · x · a
```

The total number of multiplications performed by this algorithm satisfies the recurrence $T(n) \leq T(n/2) + 2$. The recursion-tree method immediately give us the solution $T(n) = O(\log n)$.

A nearly identical exponentiation algorithm can also be derived directly from the Egyptian peasant multiplication algorithm from the previous chapter, by replacing addition with multiplication (and in particular, replacing duplation with squaring).

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ (a^2)^{n/2} & \text{if } n > 0 \text{ and } n \text{ is even} \\ (a^2)^{\lfloor n/2 \rfloor} \cdot a & \text{otherwise} \end{cases}$$

```
PeasantPower(a, n):
  if n = 1
      return a
  else if n is even
      return PeasantPower(a^2, n/2)
  else
      return PeasantPower(a^2, ⌊n/2⌋) · a
```

This algorithm—which might reasonably be called "squaring and mediation"—also performs only $O(\log n)$ multiplications.

Both of these algorithms are asymptotically optimal; any algorithm that computes a^n *must* perform at least $\Omega(\log n)$ multiplications, because each multiplication at most doubles the largest power computed so far. In fact, when n is a power of two, both of these algorithms require exactly $\log_2 n$ multiplications, which is *exactly* optimal. However, there are slightly faster methods for other values of n. For example, PiṅgalaPower and PeasantPower each compute a^{15} using six multiplications, but in fact only five multiplications are necessary:

- Piṅgala: $a \to a^2 \to a^3 \to a^6 \to a^7 \to a^{14} \to a^{15}$
- Peasant: $a \to a^2 \to a^4 \to a^8 \to a^{12} \to a^{14} \to a^{15}$
- Optimal: $a \to a^2 \to a^3 \to a^5 \to a^{10} \to a^{15}$

It is a long-standing open question whether the absolute minimum number of multiplications for a given exponent n can be computed efficiently.

Exercises

Tower of Hanoi

1. Prove that the original recursive Tower of Hanoi algorithm performs *exactly* the same sequence of moves—the same disks, to and from the same pegs, in the same order—as each of the following non-recursive algorithms. The pegs are labeled 0, 1, and 2, and our problem is to move a stack of n disks from peg 0 to peg 2 (as shown on page 24).

   (a) If n is even, swap pegs 1 and 2. At the ith step, make the only legal move that avoids peg i mod 3. If there is no legal move, then all disks are on peg i mod 3, and the puzzle is solved.

   (b) For the first move, move disk 1 to peg 1 if n is even and to peg 2 if n is odd. Then repeatedly make the only legal move that involves a different disk from the previous move. If no such move exists, the puzzle is solved.

   (c) Pretend that disks $n+1$, $n+2$, and $n+3$ are at the bottom of pegs 0, 1, and 2, respectively. Repeatedly make the only legal move that satisfies the following constraints, until no such move is possible.

- Do not place an odd disk directly on top of another odd disk.
- Do not place an even disk directly on top of another even disk.
- Do not undo the previous move.

(d) Let $\rho(n)$ denote the smallest integer k such that $n/2^k$ is not an integer. For example, $\rho(42) = 2$, because $42/2^1$ is an integer but $42/2^2$ is not. (Equivalently, $\rho(n)$ is one more than the position of the least significant 1 in the binary representation of n.) Because its behavior resembles the marks on a ruler, $\rho(n)$ is sometimes called the *ruler function*.

```
RulerHanoi(n):
  i ← 1
  while ρ(i) ≤ n
      if n − i is even
          move disk ρ(i) forward        ⟨⟨0 → 1 → 2 → 0⟩⟩
      else
          move disk ρ(i) backward       ⟨⟨0 → 2 → 1 → 0⟩⟩
      i ← i + 1
```

2. The Tower of Hanoi is a relatively recent descendant of a much older mechanical puzzle known as the Chinese linked rings, Baguenaudier, Cardan's Rings, Meleda, Patience, Tiring Irons, Prisoner's Lock, Spin-Out, and many other names. This puzzle was already well known in both China and Europe by the 16th century. The Italian mathematician Luca Pacioli described the 7-ring puzzle and its solution in his unpublished treatise *De Viribus Quantitatis*, written between 1498 and 1506;[13] only a few years later, the Ming-dynasty poet Yang Shen described the 9-ring puzzle as "a toy for women and children". The puzzle is apocryphally attributed to a 2nd-century Chinese general, who gave the puzzle to his wife to occupy her time while he was away at war.

Figure 1.16. The 7-ring Baguenaudier, from *Récréations Mathématiques* by Édouard Lucas (1891) (See Image Credits at the end of the book.)

[13] *De Viribus Quantitatis* [*On the Powers of Numbers*] is an important early work on recreational mathematics and perhaps the oldest surviving treatise on magic. Pacioli is better known for *Summa de Aritmetica*, a near-complete encyclopedia of late 15th-century mathematics, which included the first description of double-entry bookkeeping.

The Baguenaudier puzzle has many physical forms, but one of the most common consists of a long metal loop and several rings, which are connected to a solid base by movable rods. The loop is initially threaded through the rings as shown in Figure 1.16; the goal of the puzzle is to remove the loop.

More abstractly, we can model the puzzle as a sequence of bits, one for each ring, where the ith bit is `1` if the loop passes through the ith ring and `0` otherwise. (Here we index the rings from right to left, as shown in Figure 1.16.) The puzzle allows two legal moves:

- You can always flip the 1st (= rightmost) bit.
- If the bit string ends with exactly z `0`s, you can flip the $(z+2)$th bit.

The goal of the puzzle is to transform a string of n `1`s into a string of n `0`s. For example, the following sequence of 21 moves solves the 5-ring puzzle:

$$\begin{aligned}
\texttt{11111} &\xrightarrow{1} \texttt{11110} \xrightarrow{3} \texttt{11010} \xrightarrow{1} \texttt{11011} \xrightarrow{2} \texttt{11001} \xrightarrow{1} \texttt{11000} \\
&\xrightarrow{5} \texttt{01000} \xrightarrow{1} \texttt{01001} \xrightarrow{2} \texttt{01011} \xrightarrow{1} \texttt{01010} \xrightarrow{3} \texttt{01110} \\
&\xrightarrow{1} \texttt{01111} \xrightarrow{2} \texttt{01101} \xrightarrow{1} \texttt{01100} \xrightarrow{4} \texttt{00100} \xrightarrow{1} \texttt{00101} \\
&\xrightarrow{2} \texttt{00111} \xrightarrow{1} \texttt{00110} \xrightarrow{3} \texttt{00010} \xrightarrow{1} \texttt{00011} \xrightarrow{2} \texttt{00001} \xrightarrow{1} \texttt{00000}
\end{aligned}$$

♦(a) Call a sequence of moves *reduced* if no move is the inverse of the previous move. Prove that for any non-negative integer n, there is *exactly one* reduced sequence of moves that solves the n-ring Baguenaudier puzzle. *[Hint: This problem is much easier if you're already familiar with graphs.]*

(b) Describe an algorithm to solve the Baguenaudier puzzle. Your input is the number of rings n; your algorithm should print a reduced sequence of moves that solves the puzzle. For example, given the integer 5 as input, your algorithm should print the sequence $1, 3, 1, 2, 1, 5, 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1$.

(c) *Exactly* how many moves does your algorithm perform, as a function of n? Prove your answer is correct.

3. A less familiar chapter in the Tower of Hanoi's history is its brief relocation of the temple from Benares to Pisa in the early 13th century.[14] The relocation was organized by the wealthy merchant-mathematician Leonardo Fibonacci, at the request of the Holy Roman Emperor Frederick II, who had heard reports of the temple from soldiers returning from the Crusades. The Towers of Pisa and their attendant monks became famous, helping to establish Pisa as a dominant trading center on the Italian peninsula.

[14] Portions of this story are actually true.

Unfortunately, almost as soon as the temple was moved, one of the diamond needles began to lean to one side. To avoid the possibility of the leaning tower falling over from too much use, Fibonacci convinced the priests to adopt a more relaxed rule: ***Any number of disks on the leaning needle can be moved together to another needle in a single move.*** It was still forbidden to place a larger disk on top of a smaller disk, and disks had to be moved one at a time *onto* the leaning needle or between the two vertical needles.

Figure 1.17. The Towers of Pisa. In the fifth move, two disks are taken off the leaning needle.

Thanks to Fibonacci's new rule, the priests could bring about the end of the universe somewhat faster from Pisa than they could from Benares. Fortunately, the temple was moved from Pisa back to Benares after the newly crowned Pope Gregory IX excommunicated Frederick II, making the local priests less sympathetic to hosting foreign heretics with strange mathematical habits. Soon afterward, a bell tower was erected on the spot where the temple once stood; it too began to lean almost immediately.

Describe an algorithm to transfer a stack of n disks from one *vertical* needle to the other *vertical* needle, using the smallest possible number of moves. *Exactly* how many moves does your algorithm perform?

4. Consider the following restricted variants of the Tower of Hanoi puzzle In each problem, the pegs are numbered 0, 1, and 2, and your task is to move a stack of n disks from peg 0 to peg 2, exactly as in problem 1.

   (a) Suppose you are forbidden to move any disk directly between peg 1 and peg 2; *every* move must involve peg 0. Describe an algorithm to solve this version of the puzzle in as few moves as possible. *Exactly* how many moves does your algorithm make?

   ♣♥(b) Suppose you are only allowed to move disks from peg 0 to peg 2, from peg 2 to peg 1, or from peg 1 to peg 0. Equivalently, suppose the pegs are arranged in a circle and numbered in clockwise order, and you are only allowed to move disks counterclockwise. Describe an algorithm to solve this version of the puzzle in as few moves as possible. How many moves does your algorithm make?

Figure 1.18. The first several moves in a counterclockwise Towers of Hanoi solution.

♣♥(c) Finally, suppose your only restriction is that you may never move a disk directly from peg 0 to peg 2. Describe an algorithm to solve this version of the puzzle in as few moves as possible. How many moves does your algorithm make? *[Hint: Matrices! This variant is considerably harder to analyze than the other two.]*

5. Consider the following more complex variant of the Tower of Hanoi puzzle The puzzle has a row of k pegs, numbered from 1 to k. In a single turn, you are allowed to move the smallest disk on peg i to either peg $i-1$ or peg $i+1$, for any index i; as usual, you are not allowed to place a bigger disk on a smaller disk. Your mission is to move a stack of n disks from peg 1 to peg k.

(a) Describe a recursive algorithm for the case $k = 3$. *Exactly* how many moves does your algorithm make? (This is exactly the same as problem 4(a).)

(b) Describe a recursive algorithm for the case $k = n + 1$ that requires at most $O(n^3)$ moves. *[Hint: Use part (a).]*

♥(c) Describe a recursive algorithm for the case $k = n + 1$ that requires at most $O(n^2)$ moves. *[Hint: Don't use part (a).]*

♥(d) Describe a recursive algorithm for the case $k = \sqrt{n}$ that requires at most a polynomial number of moves. (Which polynomial??)

♥(e) Describe and analyze a recursive algorithm for arbitrary n and k. How small must k be (as a function of n) so that the number of moves is bounded by a polynomial in n?

Recursion Trees

6. Use recursion trees to solve each of the following recurrences.

$$A(n) = 2A(n/4) + \sqrt{n} \qquad B(n) = 2B(n/4) + n \qquad C(n) = 2C(n/4) + n^2$$
$$D(n) = 3D(n/3) + \sqrt{n} \qquad E(n) = 3E(n/3) + n \qquad F(n) = 3F(n/3) + n^2$$
$$G(n) = 4G(n/2) + \sqrt{n} \qquad H(n) = 4H(n/2) + n \qquad I(n) = 4I(n/2) + n^2$$

7. Use recursion trees to solve each of the following recurrences.

   (j) $J(n) = J(n/2) + J(n/3) + J(n/6) + n$

   (k) $K(n) = K(n/2) + 2K(n/3) + 3K(n/4) + n^2$

   (l) $L(n) = L(n/15) + L(n/10) + 2L(n/6) + \sqrt{n}$

♥8. Use recursion trees to solve each of the following recurrences.

   (m) $M(n) = 2M(n/2) + O(n \log n)$

   (n) $N(n) = 2N(n/2) + O(n/\log n)$

   (p) $P(n) = \sqrt{n}\,P(\sqrt{n}) + n$

   (q) $Q(n) = \sqrt{2n}\,Q(\sqrt{2n}) + \sqrt{n}$

Sorting

9. Suppose you are given a stack of n pancakes of different sizes. You want to sort the pancakes so that smaller pancakes are on top of larger pancakes. The only operation you can perform is a *flip*—insert a spatula under the top k pancakes, for some integer k between 1 and n, and flip them all over.

Figure 1.19. Flipping the top four pancakes.

   (a) Describe an algorithm to sort an arbitrary stack of n pancakes using $O(n)$ flips. *Exactly* how many flips does your algorithm perform in the worst case?[15] *[Hint: This problem has nothing to do with the Tower of Hanoi.]*

[15] The *exact* worst-case optimal number of flips required to sort n pancakes (either burned or unburned) is an long-standing open problem; just do the best you can.

(b) For every positive integer n, describe a stack of n pancakes that requires $\Omega(n)$ flips to sort.

(c) Now suppose one side of each pancake is burned. Describe an algorithm to sort an arbitrary stack of n pancakes, so that the burned side of every pancake is facing down, using $O(n)$ flips. *Exactly* how many flips does your algorithm perform in the worst case?

10. Recall that the *median-of-three* heuristic examines the first, last, and middle element of the array, and uses the median of those three elements as a quicksort pivot. Prove that quicksort with the median-of-three heuristic requires $\Omega(n^2)$ time to sort an array of size n in the worst case. Specifically, for any integer n, describe a permutation of the integers 1 through n, such that in every recursive call to median-of-three-quicksort, the pivot is always the second smallest element of the array. Designing this permutation requires intimate knowledge of the Partition subroutine.

(a) As a warm-up exercise, assume that the Partition subroutine is *stable*, meaning it preserves the existing order of all elements smaller than the pivot, and it preserves the existing order of all elements smaller than the pivot.

♥(b) Assume that the Partition subroutine uses the specific algorithm listed on page 29, which is *not* stable.

11. (a) Hey, Moe! Hey, Larry! Prove that the following algorithm actually sorts its input!

```
StoogeSort(A[0..n−1]):
  if n = 2 and A[0] > A[1]
      swap A[0] ↔ A[1]
  else if n > 2
      m = ⌈2n/3⌉
      StoogeSort(A[0..m−1])
      StoogeSort(A[n−m..n−1])
      StoogeSort(A[0..m−1])
```

(b) Would StoogeSort still sort correctly if we replaced $m = \lceil 2n/3 \rceil$ with $m = \lfloor 2n/3 \rfloor$? Justify your answer.

(c) State a recurrence (including the base case(s)) for the number of comparisons executed by StoogeSort.

(d) Solve the recurrence, and prove that your solution is correct. *[Hint: Ignore the ceiling.]*

(e) Prove that the number of *swaps* executed by StoogeSort is at most $\binom{n}{2}$.

12. The following cruel and unusual sorting algorithm was proposed by Gary Miller:

```
CRUEL(A[1..n]):
  if n > 1
      CRUEL(A[1..n/2])
      CRUEL(A[n/2+1..n])
      UNUSUAL(A[1..n])
```

```
UNUSUAL(A[1..n]):
  if n = 2
      if A[1] > A[2]                      ⟨⟨the only comparison!⟩⟩
          swap A[1] ↔ A[2]
  else
      for i ← 1 to n/4                    ⟨⟨swap 2nd and 3rd quarters⟩⟩
          swap A[i + n/4] ↔ A[i + n/2]
      UNUSUAL(A[1..n/2])                  ⟨⟨recurse on left half⟩⟩
      UNUSUAL(A[n/2+1..n])                ⟨⟨recurse on right half⟩⟩
      UNUSUAL(A[n/4+1..3n/4])             ⟨⟨recurse on middle half⟩⟩
```

The comparisons performed by this algorithm do not depend at all on the values in the input array; such a sorting algorithm is called ***oblivious***. Assume for this problem that the input size n is always a power of 2.

(a) Prove by induction that CRUEL correctly sorts any input array. *[Hint: Consider an array that contains $n/4$ 1s, $n/4$ 2s, $n/4$ 3s, and $n/4$ 4s. Why is this special case enough?]*

(b) Prove that CRUEL would *not* correctly sort if we removed the for-loop from UNUSUAL.

(c) Prove that CRUEL would *not* correctly sort if we swapped the last two lines of UNUSUAL.

(d) What is the running time of UNUSUAL? Justify your answer.

(e) What is the running time of CRUEL? Justify your answer.

13. An ***inversion*** in an array $A[1..n]$ is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$. The number of inversions in an n-element array is between 0 (if the array is sorted) and $\binom{n}{2}$ (if the array is sorted backward). Describe and analyze an algorithm to count the number of inversions in an n-element array in $O(n \log n)$ time. *[Hint: Modify mergesort.]*

14. (a) Suppose you are given two sets of n points, one set $\{p_1, p_2, \ldots, p_n\}$ on the line $y = 0$ and the other set $\{q_1, q_2, \ldots, q_n\}$ on the line $y = 1$. Create a set of n line segments by connect each point p_i to the corresponding point q_i. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect, in $O(n \log n)$ time. *[Hint: See the previous problem.]*

(b) Now suppose you are given two sets $\{p_1, p_2, \ldots, p_n\}$ and $\{q_1, q_2, \ldots, q_n\}$ of n points *on the unit circle*. Connect each point p_i to the corresponding

point q_i. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect in $O(n \log^2 n)$ time. *[Hint: Use your solution to part (a).]*

♥(c) Describe an algorithm for part (b) that runs in $O(n \log n)$ time. *[Hint: Use your solution from part (b)!]*

Figure 1.20. Eleven intersecting pairs of segments with endpoints on parallel lines, and ten intersecting pairs of segments with endpoints on a circle.

15. (a) Describe an algorithm that sorts an input array $A[1..n]$ by calling a subroutine SQRTSORT(k), which sorts the subarray $A\left[k+1..k+\sqrt{n}\right]$ in place, given an arbitrary integer k between 0 and $n-\sqrt{n}$ as input. (To simplify the problem, assume that $\sqrt{n}$ is an integer.) Your algorithm is ***only*** allowed to inspect or modify the input array by calling SQRTSORT; in particular, your algorithm must not directly compare, move, or copy array elements. How many times does your algorithm call SQRTSORT in the worst case?

♣(b) Prove that your algorithm from part (a) is optimal up to constant factors. In other words, if $f(n)$ is the number of times your algorithm calls SQRTSORT, prove that no algorithm can sort using $o(f(n))$ calls to SQRTSORT.

(c) Now suppose SQRTSORT is implemented recursively, by calling your sorting algorithm from part (a). For example, at the second level of recursion, the algorithm is sorting arrays roughly of size $n^{1/4}$. What is the worst-case running time of the resulting sorting algorithm? (To simplify the analysis, assume that the array size n has the form 2^{2^k}, so that repeated square roots are always integers.)

Selection

16. Suppose we are given a set S of n items, each with a *value* and a *weight*. For any element $x \in S$, we define two subsets

- $S_{<x}$ is the set of elements of S whose value is less than the value of x.
- $S_{>x}$ is the set of elements of S whose value is more than the value of x.

For any subset $R \subseteq S$, let $w(R)$ denote the sum of the weights of elements in R. The ***weighted median*** of R is any element x such that $w(S_{<x}) \leq w(S)/2$ and $w(S_{>x}) \leq w(S)/2$.

Describe and analyze an algorithm to compute the weighted median of a given weighted set in $O(n)$ time. Your input consists of two unsorted arrays $S[1..n]$ and $W[1..n]$, where for each index i, the ith element has value $S[i]$ and weight $W[i]$. You may assume that all values are distinct and all weights are positive.

17. (a) Describe an algorithm to determine in $O(n)$ time whether an arbitrary array $A[1..n]$ contains more than $n/4$ copies of any value.

    (b) Describe and analyze an algorithm to determine, given an arbitrary array $A[1..n]$ and an integer k, whether A contains more than k copies of any value. Express the running time of your algorithm as a function of both n and k.

    Do not use hashing, or radix sort, or any other method that depends on the precise input values, as opposed to their order.

18. Describe an algorithm to compute the median of an array $A[1..5]$ of distinct numbers using at most 6 comparisons. Instead of writing pseudocode, describe your algorithm using a ***decision tree***: A binary tree where each internal node contains a comparison of the form "$A[i] \gtrless A[j]$?" and each leaf contains an index into the array.

Figure 1.21. Finding the median of a 3-element array using at most 3 comparisons

19. Consider the generalization of the Blum-Floyd-Pratt-Rivest-Tarjan Mom-Select algorithm shown in Figure 1.22, which partitions the input array into $\lceil n/b \rceil$ blocks of size b, instead of $\lceil n/5 \rceil$ blocks of size 5, but is otherwise identical.

```
MombSelect(A[1..n], k):
  if n ≤ b^2
      use brute force
  else
      m ← ⌈n/b⌉
      for i ← 1 to m
          M[i] ← MedianOfB(A[b(i−1)+1..bi])
      mom_b ← MombSelect(M[1..m], ⌊m/2⌋)
      r ← Partition(A[1..n], mom_b)
      if k < r
          return MombSelect(A[1..r−1], k)
      else if k > r
          return MombSelect(A[r+1..n], k−r)
      else
          return mom_b
```

Figure 1.22. A parametrized family of selection algorithms; see problem 19.

(a) State a recurrence for the running time of $\text{Mom}_b\text{Select}$, assuming that b is a constant (so the subroutine MedianOfB runs in $O(1)$ time). In particular, how do the sizes of the recursive subproblems depend on the constant b? Consider even b and odd b separately.

(b) What is the worst-case running time of $\text{Mom}_1\text{Select}$? *[Hint: This is a trick question.]*

♣♥(c) What is the worst-case running time of $\text{Mom}_2\text{Select}$? *[Hint: This is an unfair question!]*

♥(d) What is the worst-case running time of $\text{Mom}_3\text{Select}$? Finding an upper bound on the running time is straightforward; the hard part is showing that this analysis is actually tight. *[Hint: See problem 10.]*

♥(e) What is the worst-case running time of $\text{Mom}_4\text{Select}$? Again, the hard part is showing that the analysis cannot be improved.[16]

(f) For any constants $b \geq 5$, the algorithm $\text{Mom}_b\text{Select}$ runs in $O(n)$ time, but different values of b lead to different constant factors. Let $M(b)$ denote the minimum number of comparisons required to find the median of b numbers. The exact value of $M(b)$ is known only for $b \leq 13$:

b	1	2	3	4	5	6	7	8	9	10	11	12	13
$M(b)$	0	1	3	4	6	8	10	12	14	16	18	20	23

[16] The median of four elements is either the second smallest or the second largest. In 2014, Ke Chen and Adrian Dumitrescu proved that if we modify $\text{Mom}_4\text{Select}$ to find second-smallest elements when $k < n/2$ and second-largest elements when $k > n/2$, the resulting algorithm runs in $O(n)$ time! See their paper "Select with Groups of 3 or 4 Takes Linear Time" (WADS 2015, arXiv:1409.3600) for details.

For each b between 5 and 13, find an upper bound on the running time of $\text{MOM}_b\text{SELECT}$ of the form $T(n) \le \alpha_b n$ for some explicit constant α_b. (For example, on page 39 we showed that $\alpha_5 \le 16$.)

(g) Which value of b yields the smallest constant α_b? *[Hint: This is a trick question!]*

20. Prove that the variant of the Blum-Floyd-Pratt-Rivest-Tarjan SELECT algorithm shown in Figure 1.23, which uses an extra layer of small medians to choose the main pivot, runs in $O(n)$ time.

```
MomomSelect(A[1..n], k):
  if n ≤ 81
      use brute force
  else
      m ← ⌈n/3⌉
      for i ← 1 to m
          M[i] ← MedianOf3(A[3i − 2..3i])
      mm ← ⌈m/3⌉
      for j ← 1 to mm
          Mom[j] ← MedianOf3(M[3j − 2..3j])
      momom ← MomomSelect(Mom[1..mm], ⌊mm/2⌋)
      r ← Partition(A[1..n], momom)
      if k < r
          return MomomSelect(A[1..r − 1], k)
      else if k > r
          return MomomSelect(A[r + 1..n], k − r)
      else
          return momom
```

Figure 1.23. Selection by median of moms; see problem 20).

21. (a) Suppose we are given two sorted arrays $A[1..n]$ and $B[1..n]$. Describe an algorithm to find the median element in the union of A and B in $\Theta(\log n)$ time. You can assume that the arrays contain no duplicate elements.

(b) Suppose we are given two sorted arrays $A[1..m]$ and $B[1..n]$ and an integer k. Describe an algorithm to find the kth smallest element in $A \cup B$ in $\Theta(\log(m+n))$ time. For example, if $k = 1$, your algorithm should return the smallest element of $A \cup B$.) *[Hint: Use your solution to part (a).]*

♥(c) Now suppose we are given *three* sorted arrays $A[1..n]$, $B[1..n]$, and $C[1..n]$, and an integer k. Describe an algorithm to find the kth smallest element in $A \cup B \cup C$ in $O(\log n)$ time.

(d) Finally, suppose we are given a two dimensional array $A[1..m, 1..n]$ in which every row $A[i, \cdot]$ is sorted, and an integer k. Describe an algorithm to find the kth smallest element in A as quickly as possible. How does the running time of your algorithm depend on m? *[Hint: Solve problem 16 first.]*

Arithmetic

22. In 1854, archaeologists discovered Sumerians clay tablets, carved around 2000BCE, that list the squares of integers up to 59. This discovery led some scholars to conjecture that ancient Sumerians performed multiplication by reduction to squaring, using an identity like $x \cdot y = (x^2 + y^2 - (x-y)^2)/2$. Unfortunately, those same scholars are silent on how the Sumerians supposedly squared larger numbers. Four thousand years later, we can finally rescue these Sumerian mathematicians from their lives of drudgery through the power of recursion!

    (a) Describe a variant of Karatsuba's algorithm that squares any n-digit number in $O(n^{\lg 3})$ time, by reducing to squaring three $\lceil n/2 \rceil$-digit numbers. (Karatsuba actually did this in 1960.)

    (b) Describe a recursive algorithm that squares any n-digit number in $O(n^{\log_3 6})$ time, by reducing to squaring six $\lceil n/3 \rceil$-digit numbers.

    ♥(c) Describe a recursive algorithm that squares any n-digit number in $O(n^{\log_3 5})$ time, by reducing to squaring only *five* $(n/3 + O(1))$-digit numbers. *[Hint: What is $(a+b+c)^2 + (a-b+c)^2$?]*

23. (a) Describe and analyze a variant of Karatsuba's algorithm that multiplies any m-digit number and any n-digit number, for any $n \geq m$, in $O(nm^{\lg 3 - 1})$ time.

    (b) Describe an algorithm to compute the decimal representation of 2^n in $O(n^{\lg 3})$ time, using the algorithm from part (a) as a subroutine. (The standard algorithm that computes one digit at a time requires $\Theta(n^2)$ time.)

    (c) Describe a divide-and-conquer algorithm to compute the decimal representation of an arbitrary n-bit binary number in $O(n^{\lg 3})$ time. *[Hint: Watch out for an extra log factor in the running time.]*

    ♥(d) Suppose we can multiply two n-digit numbers in $O(M(n))$ time. Describe an algorithm to compute the decimal representation of an arbitrary n-bit binary number in $O(M(n) \log n)$ time. *[Hint: The analysis is the hard part; use a domain transformation.]*

24. Consider the following classical recursive algorithm for computing the factorial $n!$ of a non-negative integer n:

```
FACTORIAL(n):
  if n = 0
      return 1
  else
      return n · FACTORIAL(n − 1)
```

(a) How many multiplications does this algorithm perform?

(b) How many bits are required to write $n!$ in binary? Express your answer in the form $\Theta(f(n))$, for some familiar function $f(n)$. *[Hint:* $(n/2)^{n/2} < n! < n^n$*.]*

(c) Your answer to (b) should convince you that the number of multiplications is *not* a good estimate of the actual running time of FACTORIAL. We can multiply any k-digit number and any l-digit number in $O(k \cdot l)$ time using either the lattice algorithm or duplation and mediation. What is the running time of FACTORIAL if we use this multiplication algorithm as a subroutine?

(d) The following recursive algorithm also computes the factorial function, but using a different grouping of the multiplications:

```
FALLING(n, m):                    ⟨⟨Compute n!/(n − m)!⟩⟩
  if m = 0
      return 1
  else if m = 1
      return n
  else
      return FALLING(n, ⌊m/2⌋) · FALLING(n − ⌊m/2⌋, ⌈m/2⌉)
```

What is the running time of FALLING(n, n) if we use grade-school multiplication? *[Hint: As usual, ignore the floors and ceilings.]*

(e) Describe and analyze a variant of Karatsuba's algorithm that multiplies any k-digit number and any l-digit number, for any $k \geq l$, in $O(k \cdot l^{\lg 3 - 1}) = O(k \cdot l^{0.585})$ time.

♥(f) What are the running times of FACTORIAL(n) and FALLING(n, n) if we use the modified Karatsuba multiplication from part (e)?

25. The ***greatest common divisor*** of two positive integer x and y, denoted $\mathbf{gcd}(x, y)$, is the largest integer d such that both x/d and y/d are integers. Euclid's *Elements*, written around 300BCE, describes the following recursive algorithm to compute $\gcd(x, y)$: [17]

[17] Euclid's algorithm is sometimes incorrectly described as the oldest recursive algorithm, or even the oldest *nontrivial* algorithm, even though the Egyptian duplation and mediation algorithm—which is both nontrivial and recursive—predates Euclid by at least 1500 years.

```
EuclidGCD(x, y):
  if x = y
      return x
  else if x > y
      return EuclidGCD(x − y, y)
  else
      return EuclidGCD(x, y − x)
```

(a) Prove that EuclidGCD correctly computes $\gcd(x, y)$.[18] Specifically:

  i. Prove that EuclidGCD(x, y) divides both x and y.

  ii. Prove that every divisor of x and y is a divisor of EuclidGCD(x, y).

(b) What is the worst-case running time of EuclidGCD(x, y), as a function of x and y? (Assume that computing $x - y$ requires $O(\log x + \log y)$ time.)

(c) Prove that the following algorithm also computes $\gcd(x, y)$:

```
FastEuclidGCD(x, y):
  if y = 0
      return x
  else if x > y
      return FastEuclidGCD(y, x mod y)
  else
      return FastEuclidGCD(x, y mod x)
```

(d) What is the worst-case running time of FastEuclidGCD(x, y), as a function of x and y? (Assume that computing $x \bmod y$ takes $O(\log x \cdot \log y)$ time.)

(e) Prove that the following algorithm also computes $\gcd(x, y)$:

```
BinaryGCD(x, y):
  if x = y
      return x
  else if x and y are both even
      return 2 · BinaryGCD(x/2, y/2)
  else if x is even
      return BinaryGCD(x/2, y)
  else if y is even
      return BinaryGCD(x, y/2)
  else if x > y
      return BinaryGCD((x − y)/2, y)
  else
      return BinaryGCD(x, (y − x)/2)
```

[18] Euclid did not do this. Proposition 1 in *Elements* Book VII states that if EuclidGCD$(x, y) = 1$, then x and y are relatively prime (that is, $\gcd(x, y) = 1$), but the proof only considers the special case $x \bmod (y \bmod (x \bmod y)) = 1$. Proposition 2 states that if x and y are *not* relatively prime, then EuclidGCD$(x, y) = \gcd(x, y)$, but the proof only considers the special cases $\gcd(x, y) = y$ and $\gcd(x, y) = y \bmod (x \bmod y)$. Finally, these two Propositions do not make a complete proof that EuclidGCD is correct. Don't be like Euclid.

(f) What is the worst-case running time of BinaryGCD(x, y), as a function of x and y? (Assume that computing $x - y$ takes $O(\log x + \log y)$ time, and computing $z/2$ requires $O(\log z)$ time.)

Arrays

26. Suppose you are given a $2^n \times 2^n$ checkerboard with one (arbitrarily chosen) square removed. Describe and analyze an algorithm to compute a tiling of the board by without gaps or overlaps by **L**-shaped tiles, each composed of 3 squares. Your input is the integer n and two n-bit integers representing the row and column of the missing square. The output is a list of the positions and orientations of $(4^n - 1)/3$ tiles. Your algorithm should run in $O(4^n)$ time. *[Hint: First prove that such a tiling always exists.]*

27. You are a visitor at a political convention (or perhaps a faculty meeting) with n delegates; each delegate is a member of exactly one political party. It is impossible to tell which political party any delegate belongs to; in particular, you will be summarily ejected from the convention if you ask. However, you can determine whether any pair of delegates belong to the *same* party by introducing them to each other. Members of the same political party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.[19]

    (a) Suppose more than half of the delegates belong to the same political party. Describe an efficient algorithm that identifies all members of this majority party.

    (b) Now suppose there are more than two parties, but one party has a *plurality*: more people belong to that party than to any other party. Present a practical procedure to precisely pick the people from the plurality political party as parsimoniously as possible, presuming the plurality party is composed of at least p people. Pretty please.

28. Smullyan Island has three types of inhabitants: *knights* always speak the truth; *knaves* always lie; and *normals* sometimes speak the truth and sometimes don't. Everyone on the island knows everyone else's name and type (knight, knave, or normal). You want to learn the type of every inhabitant.

    You can ask any inhabitant to tell you the type of any other inhabitant. Specifically, if you ask "Hey X, what is Y's type?" then X will respond as follows:

[19] Real-world politics is much messier than this simplified model, but this is a theory book!

- If X is a knight, then X will respond with Y's correct type.
- If X is a knave, then X could respond with *either* of the types that Y is *not*.
- If X is a normal, then X could respond with *any* of the three types.

The inhabitants will ignore any questions not of this precise form; in particular, you may not ask an inhabitant about their own type. Asking the same inhabitant the same question multiple times always yields the same answer, so there's no point in asking any question more than once.

(a) Suppose you know that a strict majority of inhabitants are knights. Describe an efficient algorithm to identify the type of every inhabitant.

(b) Prove that if at most half the inhabitants are knights, it is impossible to determine the type of every inhabitant.

29. Most graphics hardware includes support for a low-level operation called *blit*, or **bl**ock **t**ransfer, which quickly copies a rectangular chunk of a pixel map (a two-dimensional array of pixel values) from one location to another. This is a two-dimensional version of the standard C library function `memcpy()`.

Suppose we want to rotate an $n \times n$ pixel map 90° clockwise. One way to do this, at least when n is a power of two, is to split the pixel map into four $n/2 \times n/2$ blocks, move each block to its proper position using a sequence of five blits, and then recursively rotate each block. (Why five? For the same reason the Tower of Hanoi puzzle needs a third peg.) Alternately, we could *first* recursively rotate the blocks and *then* blit them into place.

Figure 1.24. Two algorithms for rotating a pixel map.

(a) Prove that both versions of the algorithm are correct when n is a power of 2.

(b) *Exactly* how many blits does the algorithm perform when n is a power of 2?

(c) Describe how to modify the algorithm so that it works for arbitrary n, not just powers of 2. How many blits does your modified algorithm perform?

(d) What is your algorithm's running time if a $k \times k$ blit takes $O(k^2)$ time?

(e) What if a $k \times k$ blit takes only $O(k)$ time?

30. An array $A[0..n-1]$ of n distinct numbers is ***bitonic*** if there are unique indices i and j such that $A[(i-1) \bmod n] < A[i] > A[(i+1) \bmod n]$ and

Figure 1.25. The first rotation algorithm (blit then recurse) in action. (See Image Credits at the end of the book.)

$A[(j-1) \bmod n] > A[j] < A[(j+1) \bmod n]$. In other words, a bitonic sequence either consists of an increasing sequence followed by a decreasing sequence, or can be circularly shifted to become so. For example,

4	6	9	8	7	5	1	2	3

is bitonic, but

3	6	9	8	7	5	1	2	4

is *not* bitonic.

Describe and analyze an algorithm to find the *smallest* element in an n-element bitonic array in $O(\log n)$ time. You may assume that the numbers in the input array are distinct.

31. Suppose we are given an array $A[1..n]$ of n distinct integers, which could be positive, negative, or zero, sorted in increasing order so that $A[1] < A[2] < \cdots < A[n]$.

    (a) Describe a fast algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists.

    (b) Suppose we know in advance that $A[1] > 0$. Describe an even faster algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists. *[Hint: This is **really** easy.]*

32. Suppose we are given an array $A[1..n]$ with the special property that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. We say that an element $A[x]$ is a *local minimum* if it is less than or equal to both its neighbors, or more formally, if $A[x-1] \geq A[x]$ and $A[x] \leq A[x+1]$. For example, there are six local minima in the following array:

9	7	7	2	1	3	7	5	4	7	3	3	4	8	6	9
	▲			▲				▲		▲	▲			▲	

We can obviously find a local minimum in $O(n)$ time by scanning through the array. Describe and analyze an algorithm that finds a local minimum in $O(\log n)$ time. *[Hint: With the given boundary conditions, the array* ***must*** *have at least one local minimum. Why?]*

33. Suppose you are given a sorted array of n distinct numbers that has been *rotated* k steps, for some ***unknown*** integer k between 1 and $n-1$. That is, you are given an array $A[1..n]$ such that some prefix $A[1..k]$ is sorted in increasing order, the corresponding suffix $A[k+1..n]$ is sorted in increasing order, and $A[n] < A[1]$.

    For example, you might be given the following 16-element array (where $k = 10$):

9	13	16	18	19	23	28	31	37	42	1	3	4	5	7	8

    (a) Describe and analyze an algorithm to compute the unknown integer k.

    (b) Describe and analyze an algorithm to determine if the given array contains a given number x.

34. At the end of the second act of the action blockbuster *Fast and Impossible XIII¾: The Last Guardians of Expendable Justice Reloaded*,the villainous Dr. Metaphor hypnotizes the entire Hero League/Force/Squad, arranges them in a long line at the edge of a cliff, and instructs each hero to shoot the closest taller heroes to their left and right, at a prearranged signal.

    Suppose we are given the heights of all n heroes, in order from left to right, in an array $Ht[1..n]$. (To avoid salary arguments, the producers insisted that no two heroes have the same height.) Then we can compute the Left and Right targets of each hero in $O(n^2)$ time using the following brute-force algorithm.

```
WhoTargetsWhom(Ht[1..n]):
  for j ← 1 to n
      ⟨⟨Find the left target L[j] for hero j⟩⟩
      L[j] ← None
      for i ← 1 to j − 1
          if Ht[i] > Ht[j]
              L[j] ← i
      ⟨⟨Find the right target R[j] for hero j⟩⟩
      R[j] ← None
      for k ← n down to j + 1
          if Ht[k] > Ht[j]
              R[j] ← k
  return L[1..n], R[1..n]
```

(a) Describe a divide-and-conquer algorithm that computes the output of WhoTargetsWhom in $O(n \log n)$ time.

(b) Prove that at least $\lfloor n/2 \rfloor$ of the n heroes are targets. That is, prove that the output arrays $R[0..n-1]$ and $L[0..n-1]$ contain at least $\lfloor n/2 \rfloor$ distinct values (other than None).

(c) Alas, Dr. Metaphor's diabolical plan is successful. At the prearranged signal, all the heroes simultaneously shoot their targets, and all targets fall over the cliff, apparently dead. Metaphor repeats his dastardly experiment over and over; after each massacre, he forces the remaining heroes to choose new targets, following the same algorithm, and then shoot their targets at the next signal. Eventually, only the shortest member of the Hero Crew/Alliance/Posse is left alive.[20]

Describe and analyze an algorithm to compute the number of rounds before Dr. Metaphor's deadly process finally ends. For full credit, your algorithm should run in $O(n)$ time.

35. You are a contestant on the hit game show "Beat Your Neighbors!" You are presented with an $m \times n$ grid of boxes, each containing a unique number. It costs \$100 to open a box. Your goal is to find a box whose number is larger than its neighbors in the grid (above, below, left, and right). If you spend less money than any of your opponents, you win a week-long trip for two to Las Vegas and a year's supply of Rice-A-Roni™, to which you are hopelessly addicted.

(a) Suppose $m = 1$. Describe an algorithm that finds a number that is bigger than either of its neighbors. How many boxes does your algorithm open in the worst case?

♥(b) Suppose $m = n$. Describe an algorithm that finds a number that is bigger than any of its neighbors. How many boxes does your algorithm open in the worst case?

♣♥(c) Prove that your solution to part (b) is optimal up to a constant factor.

36. (a) Let $n = 2^\ell - 1$ for some positive integer ℓ. Suppose someone claims to hold an unsorted array $A[1..n]$ of *distinct* ℓ-bit strings; thus, exactly one ℓ-bit string does *not* appear in A. Suppose further that the ***only*** way we can access A is by calling the function FetchBit(i, j), which returns the jth bit of the string $A[i]$ in $O(1)$ time. Describe an algorithm to find the missing string in A using only $O(n)$ calls to FetchBit.

[20] In the thrilling final act, Retcon the Squirrel, the last surviving member of the Hero Team/Group/Society, saves everyone by traveling back in time and retroactively replacing the other $n-1$ heroes with lifelike balloon sculptures. So, yeah, basically it's *Avengers: Endgame*.

♥(b) Now suppose $n = 2^\ell - k$ for some positive integers k and ℓ, and again we are given an array $A[1..n]$ of *distinct* ℓ-bit strings. Describe an algorithm to find the k strings that are missing from A using only $O(n \log k)$ calls to FetchBit.

Trees

37. For this problem, a *subtree* of a binary tree means any connected subgraph. A binary tree is *complete* if every internal node has two children, and every leaf has exactly the same depth. Describe and analyze a recursive algorithm to compute the *largest complete subtree* of a given binary tree. Your algorithm should return both the root and the depth of this subtree. See Figure 1.26 for an example.

Figure 1.26. The largest complete subtree of this binary tree has depth 3.

38. Let T be a binary tree with n vertices. Deleting any vertex v splits T into at most three subtrees, containing the left child of v (if any), the right child of v (if any), and the parent of v (if any). We call v a ***central*** vertex if each of these smaller trees has at most $n/2$ vertices. See Figure 1.27 for an example.

    Describe and analyze an algorithm to find a central vertex in an arbitrary given binary tree. *[Hint: First prove that every tree has a central vertex.]*

Figure 1.27. Deleting a central vertex in a 34-node binary tree, leaving subtrees with 14, 7, and 12 nodes.

39. (a) Professor George O'Jungle has a 27-node binary tree, in which every node is labeled with a unique letter of the Roman alphabet or the character **&**. Preorder and postorder traversals of the tree visit the nodes in the following order:

    - Preorder: `I Q J H L E M V O T S B R G Y Z K C A & F P N U D W X`
    - Postorder: `H E M L J V Q S G Y R Z B T C P U D N F W & X A K O I`

    Draw George's binary tree.

    (b) Recall that a binary tree is *full* if every non-leaf node has exactly two children.

    i. Describe and analyze a recursive algorithm to reconstruct an arbitrary *full* binary tree, given its preorder and postorder node sequences as input.

    ii. Prove that there is no algorithm to reconstruct an *arbitrary* binary tree from its preorder and postorder node sequences.

    (c) Describe and analyze a recursive algorithm to reconstruct an *arbitrary* binary tree, given its preorder and *inorder* node sequences as input.

    (d) Describe and analyze a recursive algorithm to reconstruct an arbitrary binary *search* tree, given only its preorder node sequence.

    ♥(e) Describe and analyze a recursive algorithm to reconstruct an arbitrary binary *search* tree, given only its preorder node sequence, ***in $O(n)$ time***.

    In parts (b)–(e), assume that all keys are distinct and that the input is consistent with at least one binary tree.

40. Suppose we have n points scattered inside a two-dimensional box. A ***kd-tree***[21] recursively subdivides the points as follows. If the box contains no points in its interior, we are done. Otherwise, we split the box into two smaller boxes with a *vertical* line, through a median point inside the box (*not* on its boundary), partitioning the points as evenly as possible. Then we recursively build a kd-tree for the points in each of the two smaller boxes, *after rotating them 90 degrees*. Thus, we alternate between splitting vertically and splitting horizontally at each level of recursion. The final empty boxes are called *cells*.

[21]The term "kd-tree" (pronounced "kay *dee* tree") was originally an abbreviation for "k-dimensional tree", but modern usage ignores this etymology, in part because nobody in their right mind would *ever* use the letter k to denote dimension instead of the *obviously* superior d. Etymological consistency would require calling the data structure in this problem a "2d-tree" (or perhaps a "2-d tree"), but the standard nomenclature is now "two-dimensional kd-tree". See also: B-tree (maybe), alpha shape, beta skeleton, epsilon net, Potomac River, Mississippi River, Lake Michigan, Lake Tahoe, Manhattan Island, La Brea Tar Pits, Sahara Desert, Mount Kilimanjaro, South Vietnam, East Timor, the Milky Way Galaxy, the City of Townsville, and self-driving automobiles.

Figure 1.28. A kd-tree for 15 points. The dashed line crosses the four shaded cells.

(a) How many cells are there, as a function of n? Prove your answer is correct.

(b) In the worst case, *exactly* how many cells can a horizontal line cross, as a function of n? Prove your answer is correct. Assume that $n = 2^k - 1$ for some integer k. *[Hint: There is more than one function f such that $f(16) = 4$.]*

(c) Suppose we are given n points stored in a kd-tree. Describe and analyze an algorithm that counts the number of points above a horizontal line (such as the dashed line in the figure) as quickly as possible. *[Hint: Use part (b).]*

(d) Describe an analyze an efficient algorithm that counts, given a kd-tree containing n points, the number of points that lie inside a rectangle R with horizontal and vertical sides. *[Hint: Use part (c).]*

♥41. Bob Ratenbur, a new student in CS 225, is trying to write code to perform preorder, inorder, and postorder traversals of binary trees. Bob sort-of understands the basic idea behind the traversal algorithms, but whenever he actually tries to implement them, he keeps mixing up the recursive calls. Five minutes before the deadline, Bob frantically submits code with the following structure:

```
PreOrder(v):
  if v = Null
      return
  else
      print label(v)
      [redacted]Order(left(v))
      [redacted]Order(right(v))
```

```
InOrder(v):
  if v = Null
      return
  else
      [redacted]Order(left(v))
      print label(v)
      [redacted]Order(right(v))
```

```
PostOrder(v):
  if v = Null
      return
  else
      [redacted]Order(left(v))
      [redacted]Order(right(v))
      print label(v)
```

Each [redacted] in this pseudocode hides one of the prefixes Pre, In, or Post. Moreover, each of the following function calls appears exactly once in Bob's submitted code:

PreOrder($left(v)$)	PreOrder($right(v)$)
InOrder($left(v)$)	InOrder($right(v)$)
PostOrder($left(v)$)	PostOrder($right(v)$)

Thus, there are precisely 36 possibilities for Bob's code. Unfortunately, Bob accidentally deleted his source code after submitting the executable, so neither you nor he knows which functions were called where.

Now suppose you are given the output of Bob's traversal algorithms, executed on some ***unknown*** binary tree T. Bob's output has been helpfully parsed into three arrays $Pre[1..n]$, $In[1..n]$, and $Post[1..n]$. You may assume that these traversal sequences are consistent with exactly one binary tree T; in particular, the vertex labels of the unknown tree T are distinct, and every internal node in T has exactly two children.

(a) Describe an algorithm to reconstruct the unknown tree T from the given traversal sequences.

(b) Describe an algorithm that either reconstructs Bob's code from the given traversal sequences, or correctly reports that the traversal sequences are consistent with more than one set of algorithms.

For example, given the input

$$Pre[1..n] = [\texttt{H A E C B I F G D}]$$
$$In[1..n] = [\texttt{A H D C E I F B G}]$$
$$Post[1..n] = [\texttt{A E I B F C D G H}]$$

your first algorithm should return the following tree:

and your second algorithm should reconstruct the following code:

```
PreOrder(v):
  if v = Null
      return
  else
      print label(v)
      PreOrder(left(v))
      PostOrder(right(v))
```

```
InOrder(v):
  if v = Null
      return
  else
      PostOrder(left(v))
      print label(v)
      PreOrder(right(v))
```

```
PostOrder(v):
  if v = Null
      return
  else
      InOrder(left(v))
      InOrder(right(v))
      print label(v)
```

♥42. Let T be a binary tree whose nodes store distinct numerical values. Recall that T is a ***binary search tree*** if and only if either (1) T is empty, or (2) T satisfies the following recursive conditions:

- The left subtree of T is a binary search tree.
- All values in the left subtree are smaller than the value at the root.
- The right subtree of T is a binary search tree.
- All values in the right subtree are larger than the value at the root.

Consider the following pair of operations on binary trees:

- ***Rotate*** an arbitrary node upward.[22]

- ***Swap*** the left and right subtrees of an arbitrary node.

In both of these operations, some, all, or none of the subtrees A, B, and C could be empty.

(a) Describe an algorithm to transform an *arbitrary* n-node binary tree with distinct node values into a binary search tree, using at most $O(n^2)$ rotations and swaps. Figure 1.29 shows a sequence of eight operations that transforms a five-node binary tree into a binary search tree.

Figure 1.29. "Sorting" a binary tree: rotate 2, rotate 2, swap 3, rotate 3, rotate 4, swap 3, rotate 2, swap 4.

Your algorithm is not allowed to directly modify parent or child pointers, create new nodes, or delete old nodes; the *only* way to modify the tree is through rotations and swaps.

On the other hand, you may *compute* anything you like for free, as long as that computation does not modify the tree; the running time of your algorithm is *defined* to be the number of rotations and swaps that it performs.

♥(b) Describe an algorithm to transform an arbitrary n-node binary tree into a binary search tree, using at most $O(n \log n)$ rotations and swaps.

[22]Rotations preserve the inorder sequence of nodes in a binary tree. Partly for this reason, rotations are used to maintain several types of balanced binary search trees, including AVL trees, red-black trees, splay trees, scapegoat trees, and treaps. See http://algorithms.wtf for lecture notes on most of these data structures.

(c) Prove that any n-node binary *search* tree can be transformed into any other binary *search* tree with the same node values, using only $O(n)$ rotations (and *no* swaps).

♥(d) **Open problem:** Either describe an algorithm to transform an arbitrary n-node binary tree into a binary search tree using only $O(n)$ rotations and swaps, or prove that no such algorithm is possible. *[Hint: I don't think it's possible.]*

Where, however, the ambiguity cannot be cleared up, either by the rule of faith or by the context, there is nothing to hinder us to point the sentence according to any method we choose of those that suggest themselves.

— Augustine of Hippo, *De doctrina Christiana* (397CE)
Translated by Marcus Dods (1892)

I dropped my dinner, and ran back to the laboratory. There, in my excitement, I tasted the contents of every beaker and evaporating dish on the table. Luckily for me, none contained any corrosive or poisonous liquid.

— Constantine Fahlberg on his discovery of saccharin,
Scientific American (1886)

The greatest challenge to any thinker is stating the problem in a way that will allow a solution.

— attributed to Bertrand Russell

When you come to a fork in the road, take it.

— Yogi Berra (giving directions to his house)

2

Backtracking

This chapter describes another important recursive strategy called ***backtracking***. A backtracking algorithm tries to construct a solution to a computational problem incrementally, one small piece at a time. Whenever the algorithm needs to decide between multiple alternatives to the next component of the solution, it recursively evaluates ***every*** alternative and then chooses the best one.

2.1 N Queens

The prototypical backtracking problem is the classical ***n Queens Problem***, first proposed by German chess enthusiast Max Bezzel in 1848 (under his pseudonym "Schachfreund") for the standard 8×8 board and by François-Joseph Eustache Lionnet in 1869 for the more general $n \times n$ board. The problem is to place n queens on an $n \times n$ chessboard, so that no two queens are attacking each other.

For readers not familiar with the rules of chess, this means that no two queens are in the same row, the same column, or the same diagonal.

Figure 2.1. Gauss's first solution to the 8 queens problem, represented by the array $[5, 7, 1, 4, 2, 8, 6, 3]$

In a letter written to his friend Heinrich Schumacher in 1850, the eminent mathematician Carl Friedrich Gauss wrote that one could easily confirm Franz Nauck's claim that the Eight Queens problem has 92 solutions by trial and error in a few hours. ("*Schwer ist es übrigens nicht, durch ein methodisches Tatonniren sich diese Gewissheit zu verschaffen, wenn man 1 oder ein paar Stunden daran wenden will.*") His description *Tatonniren* comes from the French *tâtonner*, meaning to feel, grope, or fumble around blindly, as if in the dark.

Gauss's letter described the following recursive strategy for solving the n-queens problem; the same strategy was described in 1882 by the French recreational mathematician Édouard Lucas, who attributed the method to Emmanuel Laquière. We place queens on the board one row at a time, starting with the top row. To place the rth queen, we methodically try all n squares in row r from left to right in a simple for loop. If a particular square is attacked by an earlier queen, we ignore that square; otherwise, we tentatively place a queen on that square and *recursively* grope for consistent placements of the queens in later rows.

Figure 2.2 shows the resulting algorithm, which recursively enumerates *all* complete n-queens solutions that are consistent with a given partial solution. Following Gauss, we represent the positions of the queens using an array $Q[1..n]$, where $Q[i]$ indicates which square in row i contains a queen. When PlaceQueens is called, the input parameter r is the index of the first empty row, and the prefix $Q[1..r-1]$ contains the positions of the first $r-1$ queens. In particular, to compute all n-queens solutions with no restrictions, we would call PlaceQueens$(Q[1..n], 1)$. The outer for-loop considers all possible placements of a queen on row r; the inner for-loop checks whether a candidate placement of row r is consistent with the queens that are already on the first $r-1$ rows.

The execution of PlaceQueens can be illustrated using a ***recursion tree***. Each node in this tree corresponds to a recursive subproblem, and thus to a legal partial solution; in particular, the root corresponds to the empty board

```
PlaceQueens(Q[1..n], r):
  if r = n + 1
      print Q[1..n]
  else
      for j ← 1 to n
          legal ← True
          for i ← 1 to r − 1
              if (Q[i] = j) or (Q[i] = j + r − i) or (Q[i] = j − r + i)
                  legal ← False
          if legal
              Q[r] ← j
              PlaceQueens(Q[1..n], r + 1)        ⟨⟨Recursion!⟩⟩
```

Figure 2.2. Gauss and Laquière's backtracking algorithm for the n queens problem.

(with $r = 0$). Edges in the recursion tree correspond to recursive calls. Leaves correspond to partial solutions that cannot be further extended, either because there is already a queen on every row, or because every position in the next empty row is attacked by an existing queen. The backtracking search for complete solutions is equivalent to a depth-first search of this tree.

Figure 2.3. The complete recursion tree of Gauss and Laquière's algorithm for the 4 queens problem.

2.2 Game Trees

Consider the following simple two-player game[1] played on an $n \times n$ square grid with a border of squares; let's call the players Horace Fahlberg-Remsen and Vera Rebaudi.[2] Each player has n tokens that they move across the board from one side to the other. Horace's tokens start in the left border, one in each row, and move ***hor***izontally to the right; symmetrically, Vera's tokens start in the top border, one in each column, and move ***ver***tically downward. The players alternate turns. In each of his turns, Horace either *moves* one of his tokens one step to the right into an empty square, or *jumps* one of his tokens over exactly one of Vera's tokens into an empty square two steps to the right. If no legal moves or jumps are available, Horace simply passes. Similarly, Vera either moves or jumps one of her tokens downward in each of her turns, unless no moves or jumps are possible. The first player to move all their tokens off the edge of the board wins. (It's not hard to prove that as long as there are tokens on the board, at least one player can legally move, and therefore someone eventually wins.)

Figure 2.4. Vera wins the 3×3 fake-sugar-packet game.

[1]I don't know what this game is called, or even if I'm reporting the rules correctly; I learned it (or something like it) from Lenny Pitt, who recommended playing it with fake-sugar packets at restaurants.

[2]Constantin Fahlberg and Ira Remsen synthesized saccharin for the first time in 1878, while Fahlberg was a postdoc in Remsen's lab investigating coal tar derivatives. In 1900, Ovidio Rebaudi published the first chemical analysis of *ka'a he'ê*, a medicinal plant cultivated by the Guaraní for more than 1500 years, now more commonly known as *Stevia rebaudiana*.

Unless you've seen this game before[3], you probably don't have any idea how to play it well. Nevertheless, there is a relatively simple backtracking algorithm that can play this game—or any two-player game without randomness or hidden information that ends after a finite number of moves—*perfectly*. That is, if we drop you into the middle of a game, and it is *possible* to win against another perfect player, the algorithm will tell you how to win.

A ***state*** of the game consists of the locations of all the pieces and the identity of the current player. These states can be connected into a *game tree*, which has an edge from state x to state y if and only if the current player in state x can legally move to state y. The root of the game tree is the initial position of the game, and every path from the root to a leaf is a complete game.

Figure 2.5. The first two levels of the fake-sugar-packet game tree.

To navigate through this game tree, we recursively define a game state to be ***good*** or ***bad*** as follows:

- A game state is *good* if either the current player has already won, or if the current player can move to a bad state for the opposing player.
- A game state is *bad* if either the current player has already lost, or if every available move leads to a good state for the opposing player.

Equivalently, a non-leaf node in the game tree is good if it has at least one bad child, and a non-leaf node is bad if all its children are good. By induction, any player that finds the game in a good state on their turn can win the game, even if their opponent plays perfectly; on the other hand, starting from a bad state, a player can win only if their opponent makes a mistake. This recursive definition was proposed by Ernst Zermelo in 1913.[4]

[3] If you have, please tell me where!

[4] In fact, Zermelo considered the more subtle class of games that have a finite number of states, but that allow infinite sequences of moves. (Zermelo defined infinite play to be a draw.)

This recursive definition immediately suggests the following recursive backtracking algorithm to determine whether a given game state is good or bad. At its core, this algorithm is just a depth-first search of the game tree; equivalently, the game tree is the recursion tree of the algorithm! A simple modification of this backtracking algorithm finds a good move (or even all possible good moves) if the input is a good game state.

```
PlayAnyGame(X, player):
  if player has already won in state X
      return Good
  if player has already lost in state X
      return Bad
  for all legal moves X ⇝ Y
      if PlayAnyGame(Y, ¬player) = Bad
          return Good          ⟨⟨X ⇝ Y is a good move⟩⟩
  return Bad                   ⟨⟨There are no good moves⟩⟩
```

All game-playing programs are ultimately based on this simple backtracking strategy. However, since most games have an enormous number of states, it is not possible to traverse the entire game tree in practice. Instead, game programs employ other heuristics[5] to *prune* the game tree, by ignoring states that are obviously (or "obviously") good or bad, or at least better or worse than other states, and/or by cutting off the tree at a certain depth (or *ply*) and using a more efficient heuristic to evaluate the leaves.

2.3 Subset Sum

Let's consider a more complicated problem, called SubsetSum: Given a set X of positive integers and *target* integer T, is there a subset of elements in X that add up to T? Notice that there can be more than one such subset. For example, if $X = \{8, 6, 7, 5, 3, 10, 9\}$ and $T = 15$, the answer is True, because the subsets $\{8, 7\}$ and $\{7, 5, 3\}$ and $\{6, 9\}$ and $\{5, 10\}$ all sum to 15. On the other hand, if $X = \{11, 6, 5, 1, 7, 13, 12\}$ and $T = 15$, the answer is False.

There are two trivial cases. If the target value T is zero, then we can immediately return True, because the empty set is a subset of *every* set X, and the elements of the empty set add up to zero.[6] On the other hand, if $T < 0$, or if $T \neq 0$ but the set X is empty, then we can immediately return False.

For the general case, consider an arbitrary element $x \in X$. (We've already handled the case where X is empty.) There is a subset of X that sums to T if and only if one of the following statements is true:

[5] A heuristic is an algorithm that doesn't work. (Except in practice. Sometimes. Maybe.)

[6] . . . because what else could they add up to?

- There is a subset of X that *includes* x and whose sum is T.
- There is a subset of X that *excludes* x and whose sum is T.

In the first case, there must be a subset of $X \setminus \{x\}$ that sums to $T - x$; in the second case, there must be a subset of $X \setminus \{x\}$ that sums to T. So we can solve SubsetSum(X, T) by reducing it to two simpler instances: SubsetSum$(X \setminus \{x\}, T - x)$ and SubsetSum$(X \setminus \{x\}, T)$. The resulting recursive algorithm is shown below.

```
⟨⟨Does any subset of X sum to T?⟩⟩
SubsetSum(X, T):
  if T = 0
      return True
  else if T < 0 or X = ∅
      return False
  else
      x ← any element of X
      with ← SubsetSum(X \ {x}, T − x)    ⟨⟨Recurse!⟩⟩
      wout ← SubsetSum(X \ {x}, T)        ⟨⟨Recurse!⟩⟩
      return (with ∨ wout)
```

Correctness

Proving this algorithm correct is a straightforward exercise in induction. If $T = 0$, then the elements of the empty subset sum to T, so True is the correct output. Otherwise, if T is negative or the set X is empty, then no subset of X sums to T, so False is the correct output. Otherwise, if there is a subset that sums to T, then either it contains $X[n]$ or it doesn't, and the Recursion Fairy correctly checks for each of those possibilities. Done.

Analysis

In order to analyze the algorithm, we have to describe a few implementation details more precisely. To begin, let's assume that the input set X is given as an array $X[1..n]$.

The previous recursive algorithm allows us to choose *any* element $x \in X$ in the main recursive case. Purely for the sake of efficiency, it is helpful to choose an element x such that the remaining subset $X \setminus \{x\}$ has a concise description, which can be computed quickly, so that setting up the recursive calls requires minimal overhead. Specifically, we will let x be the last element $X[n]$; then the subset $X \setminus \{x\}$ is stored in the prefix $X[1..n-1]$. Passing a complete *copy* of this prefix to the recursive calls would take too long—we need $\Theta(n)$ time just to make the copy—so instead, we push only two values: a reference to the array (or its starting address) and the length of the prefix. (Alternatively, we

could avoid passing a reference to X to *every* recursive call by making X a global variable.)

```
⟨⟨Does any subset of X[1..i] sum to T?⟩⟩
SubsetSum(X, i, T):
  if T = 0
      return True
  else if T < 0 or i = 0
      return False
  else
      with ← SubsetSum(X, i − 1, T − X[i])    ⟨⟨Recurse!⟩⟩
      wout ← SubsetSum(X, i − 1, T)           ⟨⟨Recurse!⟩⟩
      return (with ∨ wout)
```

With these implementation choices, the running time $T(n)$ of our algorithm satisfies the recurrence $T(n) \leq 2T(n-1) + O(1)$. The solution $T(n) = O(2^n)$ follows easily using either recursion trees or the even simpler "Oh yeah, we already solved this recurrence for the Tower of Hanoi" method. In the worst case—for example, when T is larger than the sum of all elements of X—the recursion tree for this algorithm is a complete binary tree with depth n, and the algorithm considers all 2^n subsets of X.

Variants

With only minor changes, we can solve several variants of SubsetSum. For example, Figure 2.6 shows an algorithm that actually *constructs* a subset of X that sums to T, if one exists, or returns the error value None if no such subset exists; this algorithm uses exactly the same recursive strategy as our earlier decision algorithms. This algorithm also runs in $O(2^n)$ time; the analysis is simplest if we assume a set data structure that allows us to insert a single element in $O(1)$ time (for example, a linked list), but in fact the running time is still $O(2^n)$ even if insertion requires $O(n)$ time (for example, a *sorted* linked list). Similar variants allow us to count subsets that sum to a particular value, or choose the *best* subset (according to some other criterion) that sums to a particular value.

Most other problems that are solved by backtracking have this property: the same recursive strategy can be used to solve many different variants of the same problem. For example, it is easy to modify the recursive strategy described in the previous section, which determines whether a given game position is good or bad, to instead return a good move, or a list of all good moves. For this reason, when we *design* backtracking algorithms, we should aim for the simplest possible variant of the problem, computing a number or even a single boolean instead of more complex information or structure.

```
⟨⟨Return a subset of X[1..i] that sums to T⟩⟩
⟨⟨or NONE if no such subset exists⟩⟩
CONSTRUCTSUBSET(X, i, T):
  if T = 0
      return ∅
  if T < 0 or n = 0
      return NONE
  Y ← CONSTRUCTSUBSET(X, i − 1, T)
  if Y ≠ NONE
      return Y
  Y ← CONSTRUCTSUBSET(X, i − 1, T − X[i])
  if Y ≠ NONE
      return Y ∪ {X[i]}
  return NONE
```

Figure 2.6. A recursive backtracking algorithm for the construction version of SUBSETSUM.

2.4 The General Pattern

Backtracking algorithms are commonly used to make a *sequence of decisions*, with the goal of building a recursively defined structure satisfying certain constraints. Often (but not always) this goal structure is itself a sequence. For example:

- In the n-queens problem, the goal is a sequence of queen positions, one in each row, such that no two queens attack each other. For each row, the algorithm *decides* where to place the queen.
- In the game tree problem, the goal is a sequence of legal moves, such that each move is as good as possible for the player making it. For each game state, the algorithm *decides* the best possible next move.
- In the SUBSETSUM problem, the goal is a sequence of input elements that have a particular sum. For each input element, the algorithm *decides* whether to include it in the output sequence or not.

(Hang on, why is the goal of *subset* sum finding a *sequence*? That was a deliberate design decision. We imposed a convenient ordering on the input set—by representing it using an array, as opposed to some other more amorphous data structure—that we can exploit in our recursive algorithm.)

In each recursive call to the backtracking algorithm, we need to make ***exactly one*** decision, and our choice must be consistent with all previous decisions. Thus, each recursive call requires not only the portion of the input data we have not yet processed, but also a suitable summary of the decisions we have already made. For the sake of efficiency, the summary of past decisions should be as small as possible. For example:

- For the n-queens problem, we must pass in not only the number of empty rows, but the positions of all previously placed queens. Here, unfortunately, we must remember our past decisions in complete detail.
- For the game tree problem, we only need to pass in the current state of the game, including the identity of the next player. We don't need to remember *anything* about our past decisions, because who wins from a given game state does not depend on the moves that created that state.[7]
- For the SubsetSum problem, we need to pass in both the remaining available integers and the remaining target value, which is the original target value minus the *sum* of the previously chosen elements. Precisely which elements were previously chosen is unimportant.

When we design new recursive backtracking algorithms, we must figure out *in advance* what information we will need about past decisions *in the middle of the algorithm*. If this information is nontrivial, our recursive algorithm might need to solve a more general problem than the one we were originally asked to solve. (We've seen this kind of generalization before: To find the *median* of an unsorted array in linear time, we derived an algorithm to select the kth smallest element for *arbitrary* k.)

Finally, once we've figured out what recursive problem we *really* need to solve, we solve that problem by ***recursive brute force***: Try *all* possibilities for the next decision that are consistent with past decisions, and let the Recursion Fairy worry about the rest. No being clever here. No skipping "obviously" stupid choices. Try everything. You can make the algorithm faster later.

2.5 Text Segmentation (*Interpunctio Verborum*)

Suppose you are given a string of letters representing text in some foreign language, but without any spaces or punctuation, and you want to break this string into its individual constituent words. For example, you might be given the following passage from Cicero's famous oration in defense of Lucius Licinius Murena in 62BCE, in the standard *scriptio continua* of classical Latin:[8]

[7] Many games *appear* to violate this independence condition. For example, the standard rules of both chess and checkers allow a player to declare a draw if the same arrangement of pieces occurs three times, and the Chinese rules for go simply forbid repeating any earlier arrangement of stones. Thus, for these games, a game state formally includes not only the current positions of the pieces but the entire history of previous moves.

[8] In·fact·most·classical·Latin·manuscripts·separated·words·with·small·dots·called·*interpuncts*. Interpunctuation all but disappeared by the 3rd century in favor of *scriptio continua*. Empty spaces between words were introduced by Irish monks in the 8th century and slowly spread across Europe over the next several centuries. *Scriptio continua* survives in early 21st-century English in the form of URLs and hashtags. #octotherps4lyfe

PRIMVSDIGNITASINTAMTENVISCIENTIANONPOTEST
ESSERESENIMSVNTPARVAEPROPEINSINGVLISLITTERIS
ATQVEINTERPVNCTIONIBUSVERBORVMOCCVPATAE

A fluent Latin reader would parse this string (in modern orthography) as *Primus dignitas in tam tenui scientia non potest esse; res enim sunt parvae, prope in singulis litteris atque interpunctionibus verborum occupatae*.[9] Text segmentation is not only a problem in classical Latin and Greek, but in several modern languages and scripts including Balinese, Burmese, Chinese, Japanese, Javanese, Khmer, Lao, Thai, Tibetan, and Vietnamese. Similar problems arise in segmenting unpunctuated English text into sentences,[10] segmenting text into lines for typesetting, speech and handwriting recognition, curve simplification, and several types of time-series analysis. For purposes of illustration, I'll stick to segmenting sequences of letters in the modern English alphabet into modern English words.

Of course, some strings can be segmented in several different ways; for example, `BOTHEARTHANDSATURNSPIN` can be decomposed into English words as either `BOTH·EARTH·AND·SATURN·SPIN` or `BOT·HEART·HANDS·AT·URNS·PIN`, among many other possibilities. For now, let's consider an extremely simple segmentation problem: Given a string of characters, can it be segmented into English words *at all*?

To make the problem concrete (and language-agnostic), let's assume we have access to a subroutine IsWord(w) that takes a string w as input, and returns True if w is a "word", or False if w is not a "word". For example, if we are trying to decompose the input string into palindromes, then a "word" is a synonym for "palindrome", and therefore IsWord(`ROTATOR`) = True but IsWord(`PALINDROME`) = False.

Just like the SubsetSum problem, the *input* structure is a sequence, this time containing letters instead of numbers, so it is natural to consider a decision process that consumes the input characters in order from left to right. Similarly, the *output* structure is a sequence of words, so it is natural to consider a process that produces the output words in order from left to right. Thus, jumping into the middle of the segmentation process, we might imagine the following picture:

[9] Loosely translated: "First of all, dignity in such paltry knowledge is impossible; this is trivial stuff, mostly concerned with individual letters and the placement of points between words." Cicero was openly mocking the legal expertise of his friend(!) and noted jurist Servius Sulpicius Rufus, who had accused Murena of bribery, after Murena defeated Rufus in election for consul. Murena was acquitted, thanks in part to Cicero's acerbic defense, although he was almost certainly guilty. #librapondo #nunquamestfidelis

[10] St. Augustine's *De doctrina Christiana* devotes an entire chapter to removing ambiguity from Latin scripture by adding punctuation.

Here the black bar separates our past decisions—splitting the first 17 letters into four words—from the portion of the input string that we have not yet processed.

The next stage in our imagined process is to ***decide*** where the next word in the output sequence ends. For this specific example, there are four possibilities for the next output word—HE, HEAR, HEART, and HEARTH. We have *no idea* which of these choices, if any, is consistent with a complete segmentation of the input string. We could be "smart" at this point and try to *figure out* which choices are good, but that would require *thinking*! Instead, let's "stupidly" try every possibility by brute force, and let the Recursion Fairy do all the real work.

- First *tentatively* accept HE as the next word, and let the Recursion Fairy make the rest of the decisions.

BLUE	**STEM**	**UNIT**	**ROBOT**	**HE**	ARTHANDSATURNSPIN

- Then *tentatively* accept HEAR as the next word, and let the Recursion Fairy make the rest of the decisions.

BLUE	**STEM**	**UNIT**	**ROBOT**	**HEAR**	THANDSATURNSPIN

- Then *tentatively* accept HEART as the next word, and let the Recursion Fairy make the rest of the decisions.

BLUE	**STEM**	**UNIT**	**ROBOT**	**HEART**	HANDSATURNSPIN

- Finally, *tentatively* accept HEARTH as the next word, and let the Recursion Fairy make the rest of the decisions.

BLUE	**STEM**	**UNIT**	**ROBOT**	**HEARTH**	ANDSATURNSPIN

As long as the Recursion Fairy reports success at least once, we report success. On the other hand, if the Recursion Fairy *never* reports success—in particular, if the set of possible next words is empty—then we report failure.

None of our past decisions affect which choices are available now; all that matters is the suffix of characters that we have not yet processed. In particular, several different sequences of past decisions could lead us to the same suffix, but they all leave us with exactly the same set of choices for that suffix.

BLUE	**STEM**	**UNIT**	**ROBOT**	HEARTHANDSATURNSPIN
BLUEST	**EMU**	**NITRO**	**BOT**	HEARTHANDSATURNSPIN

Thus, we can simplify our picture of the recursive process by discarding *everything* left of the black bar:

HEARTHANDSATURNSPIN

We are now left with a simple and natural backtracking strategy: *Select the first output word, and recursively segment the rest of the input string.*

To get a complete recursive algorithm, we need a base case. Our recursive strategy breaks down when we reach the end of the input string, because there is no next word. Fortunately, the empty string has a unique segmentation into zero words!

Putting all the pieces together, we arrive at the following simple recursive algorithm:

```
SPLITTABLE(A[1..n]):
  if n = 0
      return TRUE
  for i ← 1 to n
      if ISWORD(A[1..i])
          if SPLITTABLE(A[i+1..n])
              return TRUE
  return FALSE
```

Index Formulation

In practice, passing arrays as input parameters is rather slow; we should really find a more compact way to describe our recursive subproblems. *For purposes of designing the algorithm*, it's incredibly useful to treat the original input array as a global variable, and then reformulate the problem and the algorithm in terms of array indices instead of explicit subarrays.

For our string segmentation problem, the argument of any recursive call is always a ***suffix*** $A[i..n]$ of the original input array. So if we treat the input array $A[1..n]$ as a global variable, we can reformulate our recursive problem as follows:

Given an index i, find a segmentation of the suffix $A[i..n]$.

To describe our algorithm, we need two boolean functions:

- For any indices i and j, let $\textsc{IsWord}(i, j) = \textsc{True}$ if and only if the substring $A[i..j]$ is a word. (We're assuming this function is given to us.)
- For any index i, let $\mathit{Splittable}(i) = \textsc{True}$ if and only if the suffix $A[i..n]$ can be split into words. (This is the function we need to implement.)

For example, $\textsc{IsWord}(1, n) = \textsc{True}$ if and only if the entire input string is a single word, and $\mathit{Splittable}(1) = \textsc{True}$ if and only if the entire input string can be segmented. Our earlier recursive strategy gives us the following recurrence:

$$\mathit{Splittable}(i) = \begin{cases} \textsc{True} & \text{if } i > n \\ \displaystyle\bigvee_{j=i}^{n} \big(\textsc{IsWord}(i, j) \;\wedge\; \mathit{Splittable}(j+1)\big) & \text{otherwise} \end{cases}$$

This is *exactly* the same algorithm as we saw earlier; the only thing we've changed is the notation. The similarity is even more apparent if we rewrite the recurrence in pseudocode:

```
⟨⟨Is the suffix A[i..n] Splittable?⟩⟩
SPLITTABLE(i):
  if i > n
      return TRUE
  for j ← i to n
      if ISWORD(i, j)
          if SPLITTABLE(j + 1)
              return TRUE
  return FALSE
```

Although it may look like a trivial notational difference, using index notation instead of array notation is an important habit, not only to speed up backtracking algorithms in practice, but for developing dynamic programming algorithms, which we discuss in the next chapter.

♥Analysis

It should come as no surprise that most backtracking algorithms have exponential worst-case running times. Analyzing the precise running times of many of these algorithms requires techniques that are beyond the scope of this book. Fortunately, most of the backtracking algorithms we will encounter *in this book* are only intermediate results on the way to more efficient algorithms, which means their exact worst-case running time is not actually important. (First make it work; then make it fast.)

But just for fun, let's analyze the running time of our recursive algorithm SPLITTABLE. Because we don't know what ISWORD is doing, we can't know how long each call to ISWORD takes, so we're forced to analyze the running time in terms of the number of calls to ISWORD.[11] SPLITTABLE calls ISWORD on every prefix of the input string, and *possibly* calls itself recursively on every suffix of the output string. Thus, the "running time" of SPLITTABLE obeys the scary-looking recurrence

$$T(n) \leq \sum_{i=0}^{n-1} T(i) + O(n)$$

This really isn't as bad as it looks, especially once you've seen the trick.

First, we replace the $O(n)$ term with an explicit expression αn, for some unknown (and ultimately unimportant) constant α. Second, we conservatively

[11] In fact, as long as ISWORD runs in *polynomial* time, SPLITTABLE runs in $O(2^n)$ time.

assume that the algorithm actually makes every possible recursive call.[12] Then we can transform the "full history" recurrence into a "limited history" recurrence by subtracting the recurrence for $T(n-1)$, as follows:

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} T(i) + \alpha n \\ T(n-1) &= \sum_{i=0}^{n-2} T(i) + \alpha(n-1) \\ \Longrightarrow T(n) - T(n-1) &= T(n-1) + \alpha \end{aligned}$$

This final recurrence simplifies to $T(n) = 2T(n-1) + \alpha$. At this point, we can confidently guess (or derive via recursion trees, or remember from our Tower of Hanoi analysis) that $T(n) = O(2^n)$; indeed, this upper bound is not hard to prove by induction from the original full-history recurrence.

Moreover, this analysis is tight. There are exactly 2^{n-1} possible ways to segment a string of length n—each input character either ends a word or doesn't, except the last input character, which always ends the last word. In the worst case, our SPLITTABLE algorithm explores each of these 2^{n-1} possibilities.

Variants

Now that we have the basic recursion pattern in hand, we can use it to solve many different variants of the segmentation problem, just as we did for the SUBSETSUM problem. Here I'll describe just one example; more variations are considered in the exercises. As usual, the original input to our problem is an array $A[1..n]$.

If a string can be segmented in more than one sequence of words, we may want to find the *best* segmentation according to some criterion; conversely, if the input string cannot be segmented into words, we may want to compute the best segmentation we can find, rather than merely reporting failure. To meet both of these goals, suppose we have access to a second function SCORE that takes a string as input and returns a numerical value. For example, we might assign higher scores to longer or more common words, lower scores to shorter or more obscure words, slightly negative scores for minor spelling errors, and more negative scores to obvious non-words. Our goal is to find a segmentation that maximizes the sum of the scores of the segments.

[12] This assumption is wildly conservative for English *word* segmentation, since most strings of letters are not English words, but *not* for the similar problem of segmenting sequences of English *words* into grammatically correct English *sentences*. Consider, for example, a sequence of n copies of the word "buffalo", or n copies of the work "police", or n copies of the word "can", for any positive integer n. (At the Moulin Rouge, dances that are preservable in metal cylinders by other dances have the opportunity to fire dances that happen in prison restroom trash receptacles.)

For any index i, let $MaxScore(i)$ denote the maximum score of any segmentation of the suffix $A[i\,..\,n]$; we need to compute $MaxScore(1)$. This function satisfies the following recurrence:

$$MaxScore(i) = \begin{cases} 0 & \text{if } i > n \\ \max\limits_{i \le j \le n} \big(\textsc{Score}(A[i\,..\,j]) \;+\; MaxScore(j+1)\big) & \text{otherwise} \end{cases}$$

This is essentially the same recurrence as the one we developed for *Splittable*; the only difference is that the boolean operations $\vee$ and $\wedge$ have been replaced by the numerical operations max and $+$.

2.6 Longest Increasing Subsequence

For any sequence S, a ***subsequence*** of S is another sequence obtained from S by deleting zero or more elements, without changing the order of the remaining elements; the elements of the subsequence need not be contiguous in S. For example, when you drive down a major street in any city, you drive through a *sequence* of intersections with traffic lights, but you only have to stop at a *subsequence* of those intersections, where the traffic lights are red. If you're very lucky, you never stop at all: the empty sequence is a subsequence of S. On the other hand, if you're very unlucky, you may have to stop at every intersection: S is a subsequence of itself.

As another example, the strings BENT, ACKACK, SQUARING, and SUBSEQUENT are all subsequences of the string SUBSEQUENCEBACKTRACKING, as are the empty string and the entire string SUBSEQUENCEBACKTRACKING, but the strings QUEUE and EQUUS and TALLYHO are not. A subsequence whose elements are contiguous in the original sequence is called a ***substring***; for example, MASHER and LAUGHTER are both subsequences of MANSLAUGHTER, but only LAUGHTER is a substring.

Now suppose we are given a sequence of *integers*, and we need to find the longest subsequence whose elements are in increasing order. More concretely, the input is an integer array $A[1\,..\,n]$, and we need to compute the longest possible sequence of indices $1 \le i_1 < i_2 < \cdots < i_\ell \le n$ such that $A[i_k] < A[i_{k+1}]$ for all k.

One natural approach to building this ***longest increasing subsequence*** is to *decide*, for each index j in order from 1 to n, whether or not to include $A[j]$ in the subsequence. Jumping into the middle of this decision sequence, we might imagine the following picture:

3	**1**	**4**	1	**5**	9	2	**6**	5	3	**5**?	8	9	7	9	3	2	3	8	4	6	2	6

As in our earlier text segmentation examples, the black bar separates our past decisions from the portion of the input we have not yet processed. Numbers we

have already decided to include are highlighted and bold; numbers we have already decided to exclude are grayed out. (Notice that the numbers we've decided to include are increasing!) Our algorithm must decide whether or not to include the number immediately after the black bar.

In this example, we definitely *cannot* include 5, because then the selected numbers would no longer be in increasing order. So let's skip ahead to the next decision:

3	**1**	**4**	1	**5**	9	2	**6**	5	3	5	‖	**8**?	9	7	9	3	2	3	8	4	6	2	6

Now we *can* include 8, but it's not obvious whether we *should*. Rather than trying to be "smart", our backtracking algorithm will use simple brute force.

- First *tentatively* include the 8, and let the Recursion Fairy make the rest of the decisions.
- Then *tentatively* exclude the 8, and let the Recursion Fairy make the rest of the decisions.

Whichever choice leads to a longer increasing subsequence is the right one. (This is precisely the same recursion pattern we used to solve SubsetSum.)

Now for the key question: *What do we need to remember about our past decisions?* We can only include $A[j]$ if the resulting subsequence is in increasing order. If we assume (inductively!) that the numbers previously selected from $A[1..j-1]$ are in increasing order, then we can include $A[j]$ if and only if $A[j]$ is larger than the last number selected from $A[1..j-1]$. Thus, the only information we need about the past is ***the last number selected so far***. We can now revise our pictures by erasing everything we don't need:

6	‖	**5**?	8	9	7	9	3	2	3	8	4	6	2	6
6	‖	**8**?	9	7	9	3	2	3	8	4	6	2	6

So the problem our recursive strategy is *actually* solving is the following:

> Given an integer *prev* and an array $A[1..n]$, find the longest increasing subsequence of A in which every element is larger than *prev*.

As usual, our recursive strategy requires a base case. Our current strategy breaks down when we get to the end of the array, because there is no "next number" to consider. But an empty array has exactly one subsequence, namely, the *empty* sequence. Vacuously, every element in the empty sequence is larger than whatever value you want, and every pair of elements in the empty sequence appears in increasing order. Thus, the longest increasing subsequence of the empty array has length 0.

Here's the resulting recursive algorithm:

```
LISbigger(prev, A[1..n]):
  if n = 0
      return 0
  else if A[1] ≤ prev
      return LISbigger(prev, A[2..n])
  else
      skip ← LISbigger(prev, A[2..n])
      take ← LISbigger(A[1], A[2..n]) + 1
      return max{skip, take}
```

Okay, but remember that passing arrays around on the call stack is expensive; let's try to rephrase everything in terms of array indices, assuming that the array $A[1..n]$ is a global variable. The integer *prev* is typically an array element $A[i]$, and the remaining array is always a suffix $A[j..n]$ of the original input array. So we can reformulate our recursive *problem* as follows:

> Given two indices i and j, where $i < j$, find the longest increasing subsequence of $A[j..n]$ in which every element is larger than $A[i]$.

Let $LISbigger(i, j)$ denote the *length* of the longest increasing subsequence of $A[j..n]$ in which every element is larger than $A[i]$. Our recursive strategy gives us the following recurrence:

$$LISbigger(i,j) = \begin{cases} 0 & \text{if } j > n \\ LISbigger(i, j+1) & \text{if } A[i] \geq A[j] \\ \max \left\{ \begin{array}{c} LISbigger(i, j+1) \\ 1 + LISbigger(j, j+1) \end{array} \right\} & \text{otherwise} \end{cases}$$

Alternatively, if you prefer pseudocode:

```
LISbigger(i, j):
  if j > n
      return 0
  else if A[i] ≥ A[j]
      return LISbigger(i, j + 1)
  else
      skip ← LISbigger(i, j + 1)
      take ← LISbigger(j, j + 1) + 1
      return max{skip, take}
```

Finally, we need to connect our recursive strategy to the original problem: Finding the longest increasing subsequence of an array *with no other constraints*. The simplest approach is to add an artificial sentinel value $-\infty$ to the beginning of the array.

```
LIS(A[1..n]):
  A[0] ← −∞
  return LISBIGGER(0, 1)
```

The running time of LISBIGGER satisfies the Hanoi recurrence $T(n) \leq 2T(n-1)+O(1)$, which as usual implies that $T(n) = O(2^n)$. We really shouldn't be surprised by this running time; in the worst case, the algorithm examines each of the 2^n subsequences of the input array.

2.7 Longest Increasing Subsequence, Take 2

This is not the only backtracking strategy we can use to find longest increasing subsequences. Instead of considering the *input* sequence one element at a time, we could try to construct the *output* sequence one element at a time. That is, instead of asking "Is $A[i]$ the next element of the output sequence?", we could ask directly, "Where is the next element of the output sequence, if any?"

Jumping into the middle of this strategy, we might be faced with the following picture. Suppose we just decided to include the 6 just left of the black bar in our output sequence, and we need to decide which element to the right of the bar to include next.

Of course, we can only include numbers on the right that are greater than 6; otherwise, our output sequence would not be increasing.

But we have no idea *which* of those larger numbers is the best choice, and trying to cleverly *figure out* the best choice is too much work, and it's only going to get us into trouble anyway. Instead, we enumerate all possibilities by brute force, and let the Recursion Fairy evaluate each one.

The subset of numbers we can consider as the next element depends *only* on the last number we decided to include. Thus, we can simplify our picture of the decision process by discarding everything to the left of the bar *except* the last number we decided to include.

6	5	3	5	8	9	7	9	3	2	3	8	4	6	2	6

The remaining sequence of numbers is just a suffix of the original input array. Thus, if we think of the input array $A[1..n]$ as a global variable, we can formally express our recursive problem in terms of indices as follows:

> Given an index i, find the longest increasing subsequence of $A[i..n]$ that begins with $A[i]$.

Let $LISfirst(i)$ denote the length of the longest increasing subsequence of $A[i..n]$ that begins with $A[i]$. We can now formulate our recursive backtracking strategy as the following recursive definition:

$$LISfirst(i) = 1 + \max\left\{LISfirst(j) \;\middle|\; j > i \text{ and } A[j] > A[i]\right\}$$

Because we are dealing with sets of natural numbers, we define $\max \varnothing = 0$. Then we automatically have $LISfirst(i) = 1$ if $A[j] \leq A[i]$ for all $j > i$; in particular, $LISfirst(n) = 1$. These are the base cases for our recurrence.

We can also express this recursive definition in pseudocode as follows:

```
LISFIRST(i):
  best ← 0
  for j ← i + 1 to n
      if A[j] > A[i]
          best ← max{best, LISFIRST(j)}
  return 1 + best
```

Finally, we need to reconnect this recursive algorithm to our original problem—finding the longest increasing subsequence without knowing its first element. One natural approach that works is to try all possible first elements by brute force. Equivalently, we can add a sentinel element $-\infty$ to the beginning of the array, find the longest increasing subsequence that starts with the sentinel, and finally ignore the sentinel.

```
LIS(A[1..n]):
  best ← 0
  for i ← 1 to n
      best ← max{best, LISFIRST(i)}
  return best
```

```
LIS(A[1..n]):
  A[0] ← −∞
  return LISFIRST(0) − 1
```

2.8 Optimal Binary Search Trees

Our final example combines recursive backtracking with the divide-and-conquer strategy. Recall that the running time for a successful search in a binary search tree is proportional to the number of ancestors of the target node.[13] As a result, the worst-case search time is proportional to the depth of the tree. Thus, to minimize the worst-case search time, the height of the tree should be as small as possible; by this metric, the ideal tree is perfectly balanced.

In many applications of binary search trees, however, it is more important to minimize the total cost of several searches rather than the worst-case cost of a single search. If x is a more frequent search target than y, we can save time by building a tree where the depth of x is smaller than the depth of y, even if that means increasing the overall depth of the tree. A perfectly balanced tree is *not* the best choice if some items are significantly more popular than others. In fact, a totally unbalanced tree with depth $\Omega(n)$ might actually be the best choice!

This situation suggests the following problem. Suppose we are given a sorted array of ***keys*** $A[1..n]$ and an array of corresponding ***access frequencies*** $f[1..n]$. Our task is to build the binary search tree that minimizes the *total* search time, assuming that there will be exactly $f[i]$ searches for each key $A[i]$.

Before we think about how to solve this problem, we should first come up with a good recursive definition of the function we are trying to optimize! Suppose we are also given a binary search tree T with n nodes. Let $v_1, v_2, \ldots, v_n$ be the nodes of T, indexed in sorted order, so that each node v_i stores the corresponding key $A[i]$. Then ignoring constant factors, the total cost of performing all the binary searches is given by the following expression:

$$Cost(T, f[1..n]) := \sum_{i=1}^{n} f[i] \cdot \#\text{ancestors of } v_i \text{ in } T \qquad (*)$$

Now suppose v_r is the root of T; by definition, v_r is an ancestor of every node in T. If $i < r$, then all ancestors of v_i except the root are in the left subtree of T. Similarly, if $i > r$, then all ancestors of v_i except the root are in the right subtree of T. Thus, we can partition the cost function into three parts as follows:

$$\begin{aligned} Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] \; &+ \sum_{i=1}^{r-1} f[i] \cdot \#\text{ancestors of } v_i \text{ in } \mathit{left}(T) \\ &+ \sum_{i=r+1}^{n} f[i] \cdot \#\text{ancestors of } v_i \text{ in } \mathit{right}(T) \end{aligned}$$

The second and third summations look exactly like our original definition $(*)$

[13] An *ancestor* of a node v is either the node itself or an ancestor of the parent of v. A *proper* ancestor of v is either the parent of v or a proper ancestor of the parent of v.

for $Cost(T, f[1..n])$. Simple substitution now gives us a recurrence for $Cost$:

$$Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] \;+\; Cost(left(T), f[1..r-1]) \;+\; Cost(right(T), f[r+1..n])$$

The base case for this recurrence is, as usual, $n = 0$; the cost of performing no searches in the empty tree is zero.

Now our task is to compute the tree T_{opt} that minimizes this cost function. Suppose we somehow magically knew that the root of T_{opt} is v_r. Then the recursive definition of $Cost(T, f)$ immediately implies that the left subtree $left(T_{\text{opt}})$ must be the optimal search tree for the keys $A[1..r-1]$ and access frequencies $f[1..r-1]$. Similarly, the right subtree $right(T_{\text{opt}})$ must be the optimal search tree for the keys $A[r+1..n]$ and access frequencies $f[r+1..n]$. **Once we choose the correct key to store at the root, the Recursion Fairy will construct the rest of the optimal tree.**

More generally, let $OptCost(i, k)$ denote the total cost of the optimal search tree for the interval of frequencies $f[i..k]$. This function obeys the following recurrence.

$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ \displaystyle\sum_{j=i}^{k} f[i] + \min_{i \le r \le k} \left\{ \begin{array}{l} OptCost(i, r-1) \\ \quad + OptCost(r+1, k) \end{array} \right\} & \text{otherwise} \end{cases}$$

The base case correctly indicates that the minimum possible cost to perform zero searches into the empty set is zero! Our original problem is to compute $OptCost(1, n)$.

This recursive definition can be translated mechanically into a recursive backtracking algorithm to compute $OptCost(1, n)$. Not surprisingly, the running time of this algorithm is exponential. In the next chapter, we'll see how to reduce the running time to polynomial, so there's not much point in computing the precise running time. . .

♥Analysis

. . . unless you're into that sort of thing. Just for the fun of it, let's figure out how slow this backtracking algorithm actually is. The running time satisfies the recurrence

$$T(n) = \sum_{k=1}^{n} \big(T(k-1) + T(n-k)\big) \;+\; O(n).$$

The $O(n)$ term comes from computing the total number of searches $\sum_{i=1}^{n} f[i]$. Yeah, that's one ugly recurrence, but we can solve it using exactly the same

subtraction trick we used before. We replace the $O(\,)$ notation with an explicit constant, regroup and collect identical terms, subtract the recurrence for $T(n-1)$ to get rid of the summation, and then regroup again.

$$\begin{aligned}
T(n) &= 2\sum_{k=0}^{n-1} T(k) + \alpha n \\
T(n-1) &= 2\sum_{k=0}^{n-2} T(k) + \alpha(n-1) \\
T(n) - T(n-1) &= 2T(n-1) + \alpha \\
T(n) &= 3T(n-1) + \alpha
\end{aligned}$$

Hey, that doesn't look so bad after all. The recursion tree method immediately gives us the solution $\boldsymbol{T(n) = O(3^n)}$ (or we can just guess and confirm by induction).

This analysis implies that our recursive algorithm does *not* examine all possible binary search trees! The number of binary search trees with n vertices satisfies the recurrence

$$N(n) = \sum_{r=1}^{n-1} \big(N(r-1) \cdot N(n-r)\big),$$

which has the closed-form solution $N(n) = \Theta(4^n/\sqrt{n})$. (No, that's not obvious.) Our algorithm saves considerable time by searching *independently* for the optimal left and right subtrees for each root. A full enumeration of binary search trees would consider all possible *pairs* of left and right subtrees; hence the product in the recurrence for $N(n)$.

Exercises

1. Describe recursive algorithms for the following generalizations of the SubsetSum problem:

   (a) Given an array $X[1..n]$ of positive integers and an integer T, compute the *number* of subsets of X whose elements sum to T.

   (b) Given two arrays $X[1..n]$ and $W[1..n]$ of positive integers and an integer T, where each $W[i]$ denotes the *weight* of the corresponding element $X[i]$, compute the *maximum weight* subset of X whose elements sum to T. If no subset of X sums to T, your algorithm should return $-\infty$.

2. Describe recursive algorithms for the following variants of the text segmentation problem. Assume that you have a subroutine IsWord that takes an

array of characters as input and returns TRUE if and only if that string is a "word".

(a) Given an array $A[1..n]$ of characters, compute the number of partitions of A into words. For example, given the string `ARTISTOIL`, your algorithm should return 2, for the partitions `ARTIST·OIL` and `ART·IS·TOIL`.

(b) Given two arrays $A[1..n]$ and $B[1..n]$ of characters, decide whether A and B can be partitioned into words at the same indices. For example, the strings `BOTHEARTHANDSATURNSPIN` and `PINSTARTRAPSANDRAGSLAP` can be partitioned into words at the same indices as follows:

```
BOT·HEART·HAND·SAT·URNS·PIN
PIN·START·RAPS·AND·RAGS·LAP
```

(c) Given two arrays $A[1..n]$ and $B[1..n]$ of characters, compute the number of different ways that A and B can be partitioned into words at the same indices.

3. An *addition chain* for an integer n is an increasing sequence of integers that starts with 1 and ends with n, such that each entry after the first is the sum of two earlier entries. More formally, the integer sequence $x_0 < x_1 < x_2 < \cdots < x_\ell$ is an addition chain for n if and only if

- $x_0 = 1$,
- $x_\ell = n$, and
- for every index $k > 0$, there are indices $i \le j < k$ such that $x_k = x_i + x_j$.

The ℓength of an addition chain is the number of elements minus 1; we don't bother to count the first entry. For example, $\langle 1, 2, 3, 5, 10, 20, 23, 46, 92, 184, 187, 374\rangle$ is an addition chain for 374 of length 11.

(a) Describe a recursive backtracking algorithm to compute a minimum-length addition chain for a given positive integer n. ***Don't*** analyze or optimize your algorithm's running time, except to satisfy your own curiosity. A correct algorithm whose running time is exponential in n is sufficient for full credit. *[Hint: This problem is a lot more like n Queens than text segmentation.]*

♥(b) Describe a recursive backtracking algorithm to compute a minimum-length addition chain for a given positive integer n *in time that is sub-exponential in n. [Hint: You may find the results of certain Egyptian rope-fasteners, Indus-River prosodists, and Russian peasants helpful.]*

4. (a) Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A *common subsequence* of A and B is both a subsequence of A and a subsequence of B. Give a simple recursive definition for the function $lcs(A, B)$, which gives the length of the *longest* common subsequence of A and B.

(b) Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A *common supersequence* of A and B is another sequence that contains both A and B as subsequences. Give a simple recursive definition for the function $scs(A, B)$, which gives the length of the *shortest* common supersequence of A and B.

(c) Call a sequence $X[1..n]$ of numbers ***bitonic*** if there is an index i with $1 < i < n$, such that the prefix $X[1..i]$ is increasing and the suffix $X[i..n]$ is decreasing. Give a simple recursive definition for the function $lbs(A)$, which gives the length of the longest bitonic subsequence of an arbitrary array A of integers.

(d) Call a sequence $X[1..n]$ *oscillating* if $X[i] < X[i+1]$ for all even i, and $X[i] > X[i+1]$ for all odd i. Give a simple recursive definition for the function $los(A)$, which gives the length of the longest oscillating subsequence of an arbitrary array A of integers.

(e) Give a simple recursive definition for the function $sos(A)$, which gives the length of the shortest oscillating supersequence of an arbitrary array A of integers.

(f) Call a sequence $X[1..n]$ *convex* if $2 \cdot X[i] < X[i-1] + X[i+1]$ for all i. Give a simple recursive definition for the function $lxs(A)$, which gives the length of the longest convex subsequence of an arbitrary array A of integers.

5. For each of the following problems, the input consists of two arrays $X[1..k]$ and $Y[1..n]$ where $k \leq n$.

(a) Describe a recursive backtracking algorithm to determine whether X is a subsequence of Y. For example, the string PPAP is a subsequence of the string PENPINEAPPLEAPPLEPEN.

(b) Describe a recursive backtracking algorithm to find the smallest number of symbols that can be removed from Y so that X is no longer a subsequence. Equivalently, your algorithm should find the longest subsequence of Y that is *not* a supersequence of X. For example, after removing removing two symbols from the string PENPINEAPPLEAPPLEPEN, the string PPAP is no longer a subsequence.

♥(c) Describe a recursive backtracking algorithm to determine whether X occurs as two *disjoint* subsequences of Y. For example, the string PPAP appears as two disjoint subsequences in the string PENPINEAPPLEAPPLEPEN.

Don't analyze the running times of your algorithms, except to satisfy your own curiosity. All three algorithms run in exponential time; we'll improve that later, so the precise running time isn't particularly important.

6. This problem asks you to design backtracking algorithms to find the cost of an optimal binary search tree that satisfies additional balance constraints. Your input consists of a sorted array $A[1..n]$ of search keys and an array $f[1..n]$ of frequency counts, where $f[i]$ is the number of searches for $A[i]$. This is exactly the same cost function as described in Section 2.8. But now your task is to compute an optimal tree that satisfies some additional constraints.

   (a) ***AVL trees*** were the earliest self-balancing balanced binary search trees, first described in 1962 by Georgy Adelson-Velsky and Evgenii Landis. An AVL tree is a binary search tree where for every node v, the height of the left subtree of v and the height of the right subtree of v differ by at most one.

   Describe a recursive backtracking algorithm to construct an optimal AVL tree for a given set of search keys and frequencies.

   (b) ***Symmetric binary B-trees*** are another self-balancing binary trees, first described by Rudolf Bayer in 1972; these are better known by the name ***red-black trees***, after a somewhat simpler reformulation by Leo Guibas and Bob Sedgwick in 1978. A red-black tree is a binary search tree with the following additional constraints:

   - Every node is either red or black.
   - Every red node has a black parent.
   - Every root-to-leaf path contains the same number of black nodes.

   Describe a recursive backtracking algorithm to construct an optimal red-black tree for a given set of search keys and frequencies.

   (c) ***AA trees*** were proposed by proposed by Arne Andersson in 1993 and slightly simplified (and named) by Mark Allen Weiss in 2000. AA trees are also known as *left-leaning red-black trees*, after a symmetric reformulation (with different rebalancing algorithms) by Bob Sedgewick in 2006. An AA tree is a red-black tree with one additional constraint:

   - No left child is red.[14]

   Describe a recursive backtracking algorithm to construct an optimal AA tree for a given set of search keys and frequencies.

   Don't analyze the running times of your algorithms, except to satisfy your own curiosity. All three algorithms run in exponential time; we'll improve that later, so the precise running times aren't particularly important.

For more backtracking exercises, see the next chapter!

[14] Sedgwick's reformulation requires that no *right* child is red. Whatever. Andersson and Sedgwick are strangely silent on which end of the egg to eat first.

Potes enim videre in hac margine, qualiter hoc operati fuimus, scilicet quod iunximus primum numerum cum secundo, videlicet 1 cum 2; et secundum cum tercio; et tercium cum quarto; et quartum cum quinto, et sic deinceps....

[You can see in the margin here how we have worked this; clearly, we combined the first number with the second, namely 1 with 2, and the second with the third, and the third with the fourth, and the fourth with the fifth, and so forth....]

— Leonardo Pisano, *Liber Abaci* (1202)

Those who cannot remember the past are condemned to repeat it.

— Jorge Agustín Nicolás Ruiz de Santayana y Borrás, *The Life of Reason, Book I: Introduction and Reason in Common Sense* (1905)

You know what a learning experience is?
A learning experience is one of those things that says,
"You know that thing you just did? Don't do that."

— Douglas Adams, *The Salmon of Doubt* (2002)

3

Dynamic Programming

3.1 Mātrāvṛtta

One of the earliest examples of recursion arose in India more than 2000 years ago, in the study of poetic meter, or prosody. Classical Sanskrit poetry distinguishes between two types of syllables (*akṣara*): *light* (*laghu*) and *heavy* (*guru*). In one class of meters, variously called *mātrāvṛtta* or *mātrāchandas*, each line of poetry consists of a fixed number of "beats" (*mātrā*), where each light syllable lasts one beat and each heavy syllable lasts two beats. The formal study of *mātrā-vṛtta* dates back to the *Chandaḥśāstra*, written by the scholar Piṅgala between 600BCE and 200BCE. Piṅgala observed that there are exactly five 4-beat meters: ——, —••, •—•, ••—, and ••••. (Here each "—" represents a long syllable and each "•" represents a short syllable.)[1]

[1]In Morse code, a "dah" lasts three times as long as a "dit", but each "dit" or "dah" is followed by a pause with the same duration as a "dit". Thus, each "dit-pause" is a *laghu akṣara*, each

Although Piṅgala's text *hints* at a systematic rule for counting meters with a given number of beats,[2] it took about a millennium for that rule to be stated explicitly. In the 7th century CE, another Indian scholar named Virahāṇka wrote a commentary on Piṅgala's work, in which he observed that the number of meters with n beats is the sum of the number of meters with $(n-2)$ beats and the number of meters with $(n-1)$ beats. In more modern notation, Virahāṇka's observation implies a recurrence for the total number $M(n)$ of n-beat meters:

$$M(n) = M(n-2) + M(n-1)$$

It is not hard to see that $M(0) = 1$ (there is only one empty meter) and $M(1) = 1$ (the only one-beat meter consists of a single short syllable).

The same recurrence reappeared in Europe about 500 years after Virahāṇka, in Leonardo of Pisa's 1202 treatise *Liber Abaci*, one of the most influential early European works on "algorism". In full compliance with Stigler's Law of Eponymy,[3] the modern *Fibonacci numbers* are defined using Virahāṇka's recurrence, but with different base cases:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

In particular, we have $M(n) = F_{n+1}$ for all n.

Backtracking Can Be Slow

The recursive definition of Fibonacci numbers immediately gives us a recursive algorithm for computing them. Here is the same algorithm written in pseudocode:

"dah-pause" is a *guru akṣara*, and there are exactly five letters (M, D, R, U, and H) whose codes last four *mātrā*.

[2]The *Chandaḥśāstra* contains *two* systematic rules for listing all meters with a given number of *syllables*, which correspond roughly to writing numbers in binary from left to right (like Greeks) or from right to left (like Egyptians). The same text includes a recursive algorithm to compute 2^n (the number of meters with n syllables) by repeated squaring, and (arguably) a recursive algorithm to compute binomial coefficients (the number of meters with k short syllables and n syllables overall).

[3]*"No scientific discovery is named after its original discoverer."* In his 1980 paper that gives the law its name, the statistician Stephen Stigler jokingly claimed that this law was first proposed by sociologist Robert K. Merton. However, similar statements were previously made by Vladimir Arnol'd in the 1970's ("Discoveries are rarely attributed to the correct person."), Carl Boyer in 1968 ("Clio, the muse of history, often is fickle in attaching names to theorems!"), Alfred North Whitehead in 1917 ("Everything of importance has been said before by someone who did not discover it."), and even Stephen's father George Stigler in 1966 ("If we should ever encounter a case where a theory is named for the correct man, it will be noted."). We will see *many* other examples of Stigler's law in this book.

```
RecFibo(n):
  if n = 0
      return 0
  else if n = 1
      return 1
  else
      return RecFibo(n − 1) + RecFibo(n − 2)
```

Unfortunately, this naive recursive algorithm is horribly slow. Except for the recursive calls, the entire algorithm requires only a constant number of steps: one comparison and possibly one addition. Let $T(n)$ denote the number of recursive calls to RecFibo; this function satisfies the recurrence

$$T(0) = 1, \quad T(1) = 1, \quad T(n) = T(n-1) + T(n-2) + 1,$$

which looks an awful lot like the recurrence for Fibonacci numbers themselves! Writing out the first several values of $T(n)$ suggests the closed-form solution $\boldsymbol{T(n) = 2F_{n+1} - 1}$, which we can verify by induction (hint, hint). So computing F_n using this algorithm takes about twice as long as just counting to F_n. Methods beyond the scope of this book[4] imply that $F_n = \Theta(\phi^n)$, where $\phi = (\sqrt{5}+1)/2 \approx 1.61803$ is the so-called *golden ratio*. In short, the running time of this recursive algorithm is exponential in n.

We can actually see this exponential growth directly as follows. Think of the recursion tree for RecFibo as a binary tree of additions, with only 0s and 1s at the leaves. Since the eventual output is F_n, exactly F_n of the leaves must have value 1; these leaves represent the calls to RecRibo(1). An easy inductive argument (hint, hint) implies that RecFibo(0) is called exactly F_{n-1} times. (If we just want an asymptotic bound, it's enough to observe that the number of calls to RecFibo(0) is at most the number of calls to RecFibo(1).) Thus, the recursion tree has exactly $F_n + F_{n-1} = F_{n+1} = O(F_n)$ leaves, and therefore, because it's a full binary tree, $2F_{n+1} - 1 = O(F_n)$ nodes altogether.

Memo(r)ization: Remember Everything

The obvious reason for the recursive algorithm's lack of speed is that it computes the same Fibonacci numbers over and over and over. A single call to RecFibo(n) results in one recursive call to RecFibo($n-1$), two recursive calls to RecFibo($n-2$), three recursive calls to RecFibo($n-3$), five recursive calls to RecFibo($n-4$), and in general F_{k-1} recursive calls to RecFibo($n-k$) for any integer $0 \le k < n$. Each call is recomputing some Fibonacci number from scratch.

We can speed up our recursive algorithm considerably by writing down the results of our recursive calls and looking them up again if we need them later.

[4] See http://algorithms.wtf for notes on solving backtracking recurrences.

Figure 3.1. The recursion tree for computing F_7; arrows represent recursive calls.

This optimization technique, now known as *memoization* (yes, without an R), is usually credited to Donald Michie in 1967, but essentially the same technique was proposed in 1959 by Arthur Samuel.[5]

```
MemFibo(n):
  if n = 0
      return 0
  else if n = 1
      return 1
  else
      if F[n] is undefined
          F[n] ← MemFibo(n − 1) + MemFibo(n − 2)
      return F[n]
```

Memoization clearly decreases the running time of the algorithm, but by how much? If we actually trace through the recursive calls made by MemFibo, we find that the array $F[\,]$ is filled from the bottom up: first $F[2]$, then $F[3]$, and so on, up to $F[n]$. This pattern can be verified by induction: Each entry $F[i]$ is filled only after its predecessor $F[i-1]$. If we ignore the time spent in recursive calls, it requires only constant time to evaluate the recurrence for each Fibonacci number F_i. But by design, the recurrence for F_i is evaluated only once for each index i. We conclude that MemFibo performs only $O(n)$ additions, an *exponential* improvement over the naïve recursive algorithm!

[5]Michie proposed that programming languages should support an abstraction he called a "memo function", consisting of both a standard function ("rule") and a dictionary ("rote"), instead of separately supporting arrays and functions. Whenever a memo function computes a function value for the first time, it "memorises" (yes, with an R) that value into its dictionary. Michie was inspired by Samuel's use of "rote learning" to speed up the recursive evaluation of checkers game trees; Michie describes his more general proposal as "enabling the programmer to 'Samuelize' any functions he pleases." (As far as I can tell, Michie never used the term "memoisation" himself.) Memoization was used even earlier by Claude Shannon's maze-solving robot "Theseus", which he designed and constructed in 1950.

Figure 3.2. The recursion tree for F_7 trimmed by memoization. Downward green arrows indicate writing into the memoization array; upward red arrows indicate reading from the memoization array.

Dynamic Programming: Fill Deliberately

Once we see how the array $F[\,]$ is filled, we can replace the memoized recurrence with a simple for-loop that *intentionally* fills the array in that order, instead of relying on a more complicated recursive algorithm to do it for us accidentally.

```
IterFibo(n):
  F[0] ← 0
  F[1] ← 1
  for i ← 2 to n
      F[i] ← F[i−1] + F[i−2]
  return F[n]
```

Now the time analysis is immediate: IterFibo clearly uses $\boldsymbol{O(n)}$ ***additions*** and stores $O(n)$ *integers*.

This is our first explicit ***dynamic programming*** algorithm. The dynamic programming paradigm was formalized and popularized by Richard Bellman in the mid-1950s, while working at the RAND Corporation, although he was far from the first to use the technique. In particular, this iterative algorithm for Fibonacci numbers was already proposed by Virahāṇka and later Sanskrit prosodists in the 12th century, and again by Fibonacci at the turn of the 13th century![6]

[6]More general dynamic programming techniques were independently deployed several times in the late 1930s and early 1940s. For example, Pierre Massé used dynamic programming algorithms to optimize the operation of hydroelectric dams in France during the Vichy regime. John von Neumann and Oskar Morgenstern developed dynamic programming algorithms to determine the winner of any two-player game with perfect information (for example, checkers). Alan Turing and his cohorts used similar methods as part of their code-breaking efforts at

Many years after the fact, Bellman claimed that he deliberately chose the name "dynamic programming" to hide the mathematical character of his work from his military bosses, who were actively hostile toward anything resembling mathematical research.[7] The word "programming" does not refer to writing code, but rather to the older sense of *planning* or *scheduling*, typically by filling in a table. For example, sports programs and theater programs are schedules of important events (with ads); television programming involves filling each available time slot with a show (and ads); degree programs are schedules of classes to be taken (with ads). The Air Force funded Bellman and others to develop methods for constructing training and logistics schedules, or as they called them, "programs". The word "dynamic" was not only a reference to the multistage, time-varying processes that Bellman and his colleagues were attempting to optimize, but also a marketing buzzword that would resonate with the Futuristic Can-Do Zeitgeist™ of post-WW II America.[8] Thanks in part to Bellman's proselytizing, dynamic programming is now a standard tool for multistage planning in economics, robotics, control theory, and several other disciplines.

Don't Remember Everything After All

In many dynamic programming algorithms, it is not necessary to retain *all* intermediate results through the entire computation. For example, we can significantly reduce the space requirements of our algorithm IterFibo by maintaining only the two newest elements of the array:

Bletchley Park. Both Massé's work and von Neumann and Mergenstern's work were first published in 1944, six years before Bellman coined the phrase "dynamic programming". The details of Turing's "Banburismus" were kept secret until the mid-1980s.

[7]Charles Erwin Wilson became Secretary of Defense in January 1953, after a dozen years as the president of General Motors. "Engine Charlie" reorganized the Department of Defense and significantly decreased its budget in his first year in office, with the explicit goal of running the Department much more like an industrial corporation. Bellman described Wilson in his 1984 autobiography as follows:

> We had a very interesting gentleman in Washington named Wilson. He was secretary of Defense, and he actually had a pathological fear and hatred of the word "*research*". I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term "*research*" in his presence. You can imagine how he felt, then, about the term "*mathematical*". . . . I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose?

However, Bellman's first published use of the term "dynamic programming" already appeared in 1952, several months before Wilson took office, so this story is at least *slightly* embellished.

[8]. . . and just possibly a riff on the iconic brand name "Dynamic-Tension" for Charles Atlas's famous series of exercises, which Charles Roman coined in 1928. Hero of the Beach!

```
ITERFIBO2(n):
  prev ← 1
  curr ← 0
  for i ← 1 to n
      next ← curr + prev
      prev ← curr
      curr ← next
  return curr
```

(This algorithm uses the non-standard but consistent base case $F_{-1} = 1$ so that ITERFIBO2(0) returns the correct value 0.) Although saving space can be absolutely crucial in practice, we won't focus on space issues in this book.

♥3.2 Aside: Even Faster Fibonacci Numbers

Although the previous algorithm is simple and attractive, it is *not* the fastest algorithm to compute Fibonacci numbers. We can derive a faster algorithm by exploiting the following matrix reformulation of the Fibonacci recurrence:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} y \\ x+y \end{bmatrix}$$

In other words, multiplying a two-dimensional vector by the matrix $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right]$ has exactly the same effect as one iteration of the inner loop of ITERFIBO2. It follows that multiplying by the matrix n times is the same as iterating the loop n times:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} F_{n-1} \\ F_n \end{bmatrix}.$$

So if we want the nth Fibonacci number, we only need to compute the nth power of the matrix $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right]$. If we use repeated squaring,[9] computing the nth power of something requires only $O(\log n)$ multiplications. Here, because "something" is a 2×2 matrix, that means $O(\log n)$ 2×2 matrix multiplications, each of which reduces to a constant number of integer multiplications and additions. Thus, we can compute F_n in only ***$O(\log n)$ integer arithmetic operations***.

We can achieve the same speedup using the identity $F_n = F_m F_{n-m-1} + F_{m+1} F_{n-m}$, which holds (by induction!) for all integers m and n. In particular, this identity implies the following mutual recurrence for pairs of adjacent Fibonacci numbers, first proposed by Édouard Lucas in 1898:

$$F_{2n-1} = F_{n-1}^2 + F_n^2$$
$$F_{2n} = F_n(F_{n-1} + F_{n+1}) = F_n(2F_{n-1} + F_n)$$

[9] as suggested by Piṅgala for powers of 2 elsewhere in *Chandaḥśāstra*

(We can also derive this mutual recurrence directly from the matrix-squaring algorithm.) These recurrences translate directly into the following algorithm:

```
⟨⟨Compute the pair F_{n-1}, F_n⟩⟩
FastRecFibo(n):
  if n = 1
      return 0, 1
  m ← ⌊n/2⌋
  hprv, hcur ← FastRecFibo(m)      ⟨⟨F_{m-1}, F_m⟩⟩
  prev ← hprv^2 + hcur^2           ⟨⟨F_{2m-1}⟩⟩
  curr ← hcur · (2 · hprv + hcur)  ⟨⟨F_{2m}⟩⟩
  next ← prev + curr               ⟨⟨F_{2m+1}⟩⟩
  if n is even
      return prev, curr
  else
      return curr, next
```

Our standard recursion tree technique implies that this algorithm performs only $O(\log n)$ integer arithmetic operations.

This is an exponential speedup over the standard iterative algorithm, which was already an exponential speedup over our original recursive algorithm. Right?

Whoa! Not so fast!

Well, not exactly. Fibonacci numbers grow exponentially fast. The nth Fibonacci number is approximately $n \log_{10} \phi \approx n/5$ decimal digits long, or $n \log_2 \phi \approx 2n/3$ bits. So we can't possibly compute F_n in logarithmic *time* — we need $\Omega(n)$ time just to write down the answer!

The way out of this apparent paradox is to observe that ***we can't perform arbitrary-precision arithmetic in constant time***. Let $M(n)$ denote the time required to multiply two n-digit numbers. The running time of FastRecFibo satisfies the recurrence $T(n) = T(\lfloor n/2 \rfloor) + M(n)$, which solves to $T(n) = \boldsymbol{O(M(n))}$ via recursion trees. The fastest integer multiplication algorithm known (as of 2019) runs in $O(n \log n)$ time, so that is also the running time of the fastest algorithm known (as of 2019) to compute Fibonacci numbers.

Is this algorithm slower than our "linear-time" iterative algorithms? Actually, no—addition isn't free, either! Adding two n-digit numbers requires $O(n)$ time, so the iterative algorithms IterFibo and IterFibo2 actually run in $\boldsymbol{O(n^2)}$ ***time***. (Do you see why?) So FastRecFibo is significantly faster than the iterative algorithms, just not *exponentially* faster.

In the original recursive algorithm, the extra cost of arbitrary-precision arithmetic is overwhelmed by the huge number of recursive calls. The correct

recurrence is $T(n) = T(n-1) + T(n-2) + O(n)$, which still has the solution $T(n) = O(\phi^n)$.

3.3 Interpunctio Verborum Redux

For our next dynamic programming algorithm, let's consider the text segmentation problem from the previous chapter. We are given a string $A[1..n]$ and a subroutine IsWord that determines whether a given string is a word (whatever that means), and we want to know whether A can be partitioned into a sequence of words.

We solved this problem by defining a function $Splittable(i)$ that returns True if and only if the suffix $A[i..n]$ can be partitioned into a sequence of words. We need to compute $Splittable(1)$. This function satisfies the recurrence

$$Splittable(i) = \begin{cases} \text{True} & \text{if } i > n \\ \displaystyle\bigvee_{j=i}^{n} \big(IsWord(i,j) \wedge Splittable(j+1)\big) & \text{otherwise} \end{cases}$$

where $IsWord(i,j)$ is shorthand for $IsWord(A[i..j])$. This recurrence translates directly into a recursive backtracking algorithm that calls the IsWord subroutine $O(2^n)$ times in the worst case.

But for any fixed string $A[1..n]$, there are only n different ways to call the recursive function $Splittable(i)$—one for each value of i between 1 and $n+1$—and only $O(n^2)$ different ways to call IsWord(i,j)—one for each pair (i,j) such that $1 \le i \le j \le n$. Why are we spending exponential time computing only a polynomial amount of stuff?

Each recursive subproblem is specified by an integer between 1 and $n+1$, so we can memoize the function *Splittable* into an array $SplitTable[1..n+1]$. Each subproblem $Splittable(i)$ depends only on results of subproblems $Splittable(j)$ where $j > i$, so the memoized recursive algorithm fills the array in *decreasing* index order. If we fill the array in this order deliberately, we obtain the dynamic programming algorithm shown in Figure 3.3. The algorithm makes $O(n^2)$ calls to IsWord, an exponential improvement over our earlier backtracking algorithm.

3.4 The Pattern: Smart Recursion

In a nutshell, dynamic programming is *recursion without repetition*. Dynamic programming algorithms store the solutions of intermediate subproblems, often *but not always* in some kind of array or table. Many algorithms students

```
FastSplittable(A[1..n]):
  SplitTable[n + 1] ← True
  for i ← n down to 1
      SplitTable[i] ← False
      for j ← i to n
          if IsWord(i, j) and SplitTable[j + 1]
              SplitTable[i] ← True
  return SplitTable[1]
```

Figure 3.3. Interpunctio verborum velox

(and instructors, and textbooks) make the mistake of focusing on the table—because tables are easy and familiar—instead of the *much* more important (and difficult) task of finding a correct recurrence. As long as we memoize the correct recurrence, an explicit table isn't really necessary, but if the recurrence is incorrect, we are well and truly hosed.

Dynamic programming is *not* about filling in tables. It's about smart recursion!

Dynamic programming algorithms are best developed in two distinct stages.

1. **Formulate the problem recursively.** Write down a recursive formula or algorithm for the whole problem in terms of the answers to smaller subproblems. This is the hard part. A complete recursive formulation has two parts:

   (a) **Specification.** Describe the problem that you want to solve recursively, in coherent and precise English—not *how* to solve that problem, but *what* problem you're trying to solve. Without this specification, it is impossible, even in principle, to determine whether your solution is correct.

   (b) **Solution.** Give a clear recursive formula or algorithm for the whole problem in terms of the answers to smaller instances of *exactly* the same problem.

2. **Build solutions to your recurrence from the bottom up.** Write an algorithm that starts with the base cases of your recurrence and works its way up to the final solution, by considering intermediate subproblems in the correct order. This stage can be broken down into several smaller, relatively mechanical steps:

   (a) **Identify the subproblems.** What are all the different ways your recursive algorithm can call itself, starting with some initial input? For example, the argument to RecFibo is always an integer between 0 and n.

(b) **Choose a memoization data structure.** Find a data structure that can store the solution to *every* subproblem you identified in step (a). This is usually *but not always* a multidimensional array.

(c) **Identify dependencies.** Except for the base cases, every subproblem depends on other subproblems—which ones? Draw a picture of your data structure, pick a generic element, and draw arrows from each of the other elements it depends on. Then formalize your picture.

(d) **Find a good evaluation order.** Order the subproblems so that each one comes *after* the subproblems it depends on. You should consider the base cases first, then the subproblems that depends only on base cases, and so on, eventually building up to the original top-level problem. The dependencies you identified in the previous step define a partial order over the subproblems; you need to find a linear extension of that partial order. ***Be careful!***

(e) **Analyze space and running time.** The number of distinct subproblems determines the space complexity of your memoized algorithm. To compute the total running time, add up the running times of all possible subproblems, *assuming deeper recursive calls are already memoized*. You can actually do this immediately after step (a).

(f) **Write down the algorithm.** You know what order to consider the subproblems, and you know how to solve each subproblem. So do that! If your data structure is an array, this usually means writing a few nested for-loops around your original recurrence, and replacing the recursive calls with array look-ups.

Of course, you have to prove that each of these steps is correct. If your recurrence is wrong, or if you try to build up answers in the wrong order, your algorithm won't work!

3.5 Warning: Greed is Stupid

If we're incredibly lucky, we can bypass all the recurrences and tables and so forth, and solve the problem using a *greedy* algorithm. Like a backtracking algorithm, a greedy algorithm constructs a solution through a series of decisions, but it makes those decisions directly, *without* solving at any recursive subproblems. While this approach seems very natural, it almost never works; optimization problems that can be solved correctly by a greedy algorithm are quite rare. Nevertheless, for many problems that should be solved by backtracking or dynamic programming, many students' first intuition is to apply a greedy strategy.

For example, a greedy algorithm for the text segmentation problem might find the shortest (or, if you prefer, longest) prefix of the input string that is

a word, accept that prefix as the first word in the segmentation, and then recursively segment the remaining suffix of the input string. Similarly, a greedy algorithm for the longest increasing subsequence problem might look for the smallest element of the input array, accept that element as the start of the target subsequence, and then recursively look for the longest increasing subsequence to the right of that element. If these sound like stupid hacks to you, pat yourself on the back; these aren't even *close* to correct solutions.

Everyone should tattoo the following sentence on the back of their hands, right under all the rules about logarithms and big-Oh notation:

Greedy algorithms never work!
Use dynamic programming instead!

What, never?

No, never!

What, *never*?

Well. . . hardly ever.[10]

Because the greedy approach is so incredibly tempting, but so rarely correct, I strongly advocate the following policy in any algorithms course, even (or perhaps *especially*) for courses that do not normally ask for proofs of correctness.[11]

You will not receive *any* credit for *any* greedy algorithm, on *any* homework or exam, even if the algorithm is correct, without a *formal* proof of correctness.

Moreover, the vast majority of problems for which students are tempted to submit a greedy algorithm are actually best solved using dynamic programming. So I always offer the following advice to my algorithms students.

Whenever you write—or even *think*—the word "greeDY", your subconscious is telling you to use DYnamic programming.

Even for problems that *can* be correctly solved by greedy algorithms, it's usually more productive to develop a backtracking or dynamic programming algorithm first. First make it work, then make it fast. We will see techniques for proving greedy algorithms correct in the next chapter.

[10] They hardly ever ever work! Then give three cheers, and one cheer more, for the rigorous Captain of the *Pinafore*! Then give three cheers, and one cheer more, for the Captain of the *Pinafore*!

[11] Introducing this policy in my own algorithms courses significantly improved students' grades, because it significantly reduced the frequency of incorrect greedy algorithms.

3.6 Longest Increasing Subsequence

Another problem we considered in the previous chapter was computing the length of the longest increasing subsequence of a given array $A[1..n]$ of numbers. We developed two different recursive backtracking algorithms for this problem. Both algorithms run in $O(2^n)$ time in the worst case; both algorithms can be sped up significantly via dynamic programming.

First Recurrence: Is This Next?

Our first backtracking algorithm evaluated the function $LISbigger(i, j)$, which we defined as the length of the longest increasing subsequence of $A[j..n]$ in which every element is larger than $A[i]$. We derived the following recurrence for this function:

$$LISbigger(i,j) = \begin{cases} 0 & \text{if } j > n \\ LISbigger(i, j+1) & \text{if } A[i] \geq A[j] \\ \max \left\{ \begin{array}{c} LISbigger(i, j+1) \\ 1 + LISbigger(j, j+1) \end{array} \right\} & \text{otherwise} \end{cases}$$

To solve the original problem, we can add a sentinel value $A[0] = -\infty$ to the array and compute $LISbigger(0, 1)$.

Each recursive subproblem is identified by two indices i and j, so there are only $O(n^2)$ distinct recursive subproblems to consider. We can memoize the results of these subproblems into a two-dimensional array $\boldsymbol{LISbigger[0..n, 1..n]}$.[12] Moreover, each subproblem can be solved in $O(1)$ time, not counting recursive calls, so we should expect the final dynamic programming algorithm to run in $\boldsymbol{O(n^2)}$ ***time***.

The order in which the memoized recursive algorithm fills this array is not immediately clear; all we can tell from the recurrence is that each entry $LISbigger[i, j]$ is filled in *after* the entries $LISbigger[i, j+1]$ and $LISbigger[j, j+1]$ in the next column, as indicated on the left in Figure 3.4.

Fortunately, this partial information is enough to give us a *valid* evaluation order. If we fill the table one column at a time, from right to left, then whenever we reach an entry in the table, the entries it depends on are already available. This may not be the order that the recursive algorithm would use, but it works, so we'll go with it. The right figure in Figure 3.4 illustrates this evaluation order, with a double arrow indicating the outer loop and single arrows indicating the

[12] In fact, we only need half of this array, because we always have $i < j$. But even if we cared about constant factors in this book (we don't), this would be the wrong time to worry about them. First make it work; then make it better.

Figure 3.4. Subproblem dependencies for longest increasing subsequence, and a valid evaluation order

inner loop. In this case, the single arrows are bidirectional, because the order that we use to fill each column doesn't matter.

And we're done! Pseudocode for our dynamic programming algorithm is shown below; as expected, our algorithm clearly runs in $O(n^2)$ time. If necessary, we can reduce the space bound from $O(n^2)$ to $O(n)$ by maintaining only the two most recent columns of the table, $LISbigger[\cdot, j]$ and $LISbigger[\cdot, j+1]$.[13]

```
FastLIS(A[1..n]):
  A[0] ← −∞                          ⟨⟨Add a sentinel⟩⟩
  for i ← 0 to n                     ⟨⟨Base cases⟩⟩
      LISbigger[i, n+1] ← 0
  for j ← n down to 1
      for i ← 0 to j−1               ⟨⟨...or whatever⟩⟩
          keep ← 1 + LISbigger[j, j+1]
          skip ← LISbigger[i, j+1]
          if A[i] ≥ A[j]
              LISbigger[i, j] ← skip
          else
              LISbigger[i, j] ← max{keep, skip}
  return LISbigger[0, 1]
```

Second Recurrence: What's Next?

Our second backtracking algorithm evaluated the function $LISfirst(i)$, which we defined as the length of the longest increasing subsequence of $A[i..n]$ that begins with $A[i]$. We derived the following recurrence for this function:

$$LISfirst(i) = 1 + \max\left\{LISfirst(j) \;\middle|\; j > i \text{ and } A[j] > A[i]\right\}$$

Here, we assume that $\max\varnothing = 0$, so that the base cases like $LISfirst(n) = 1$ fall out of the recurrence automatically. To solve the original problem, we can add a sentinel value $A[0] = -\infty$ to the array and compute $LISfirst(0) - 1$.

In this case, recursive subproblems are indicated by a single index i, so we can memoize the recurrence into a one-dimensional array $LISfirst[1..n]$. Each

[13] See, I told you not to worry about constant factors yet!

entry $LISfirst[i]$ depends only on entries $LISfirst[j]$ with $j > i$, so we can fill the array in decreasing index order. To compute each $LISfirst[i]$, we need to consider $LISfirst[j]$ for *all* indices $j > i$, but we don't need to consider those indices j in any particular order. The resulting dynamic programming algorithm runs in $\boldsymbol{O(n^2)}$ ***time*** and uses $O(n)$ space.

```
FastLIS2(A[1..n]):
  A[0] = −∞                          ⟨⟨Add a sentinel⟩⟩
  for i ← n downto 0
      LISfirst[i] ← 1
      for j ← i + 1 to n             ⟨⟨...or whatever⟩⟩
          if A[j] > A[i] and 1 + LISfirst[j] > LISfirst[i]
              LISfirst[i] ← 1 + LISfirst[j]
  return LISfirst[0] − 1             ⟨⟨Don't count the sentinel⟩⟩
```

3.7 Edit Distance

The ***edit distance*** between two strings is the minimum number of letter insertions, letter deletions, and letter substitutions required to transform one string into the other. For example, the edit distance between FOOD and MONEY is at most four:

$$\underline{\text{F}}\text{OOD} \to \text{MO}\underline{\text{O}}\text{D} \to \text{MON}\underset{\wedge}{\ }\text{D} \to \text{MONE}\underline{\text{D}} \to \text{MONEY}$$

This distance function was independently proposed by Vladimir Levenshtein in 1965 (working on coding theory), Taras Vintsyuk in 1968 (working on speech recognition), and Stanislaw Ulam in 1972 (working with biological sequences). For this reason, edit distance is sometimes called ***Levenshtein distance*** or ***Ulam distance*** (but strangely, never "Vintsyuk distance").

We can visualize this editing process by aligning the strings one above the other, with a gap in the first word for each insertion and a gap in the second word for each deletion. Columns with two *different* characters correspond to substitutions. In this representation, the number of editing steps is just the number of columns that do *not* contain the same character twice.

```
F O O   D
M O N E Y
```

It's fairly obvious that we can't transform FOOD into MONEY in three steps, so the edit distance between FOOD and MONEY is exactly four. Unfortunately, it's not so easy in general to tell when a sequence of edits is as short as possible. For example, the following alignment shows that the distance between the strings ALGORITHM and ALTRUISTIC is *at most* 6. Is that the best we can do?

```
A L G O R   I   T H M
A L   T R U I S T I C
```

Recursive Structure

To develop a dynamic programming algorithm to compute edit distance, we first need to formulate the problem recursively. Our alignment representation for edit sequences has a crucial "optimal substructure" property. Suppose we have the gap representation for the shortest edit sequence for two strings. **If we remove the last column, the remaining columns must represent the shortest edit sequence for the remaining prefixes.** We can easily prove this observation by contradiction: If the prefixes had a shorter edit sequence, gluing the last column back on would gives us a shorter edit sequence for the original strings. So once we figure out what should happen in the last column, the Recursion Fairy can figure out the rest of the optimal gap representation.

Said differently, the alignment we are looking for represents a sequence of editing operations, ordered (for no particular reason) from right to left. Solving the edit distance problem requires making a sequence of decisions, one for each column in the output alignment. In the middle of this sequence of decisions, we have already aligned a suffix of one string with a suffix of the other.

ALGOR	I		T	H	M
ALTRU	I	S	T	I	C

Because the cost of an alignment is just the number of mismatched columns, our remaining decisions don't depend on the editing operations we've already chosen; they only depend on the prefixes we haven't aligned yet.

ALGOR
ALTRU

Thus, for any two input strings $A[1..m]$ and $B[1..n]$, we can formulate the edit distance problem recursively as follows: For any indices i and j, let $\boldsymbol{Edit(i,j)}$ denote the edit distance between the prefixes $A[1..i]$ and $B[1..j]$. We need to compute $Edit(m,n)$.

Recurrence

When i and j are both positive, there are exactly three possibilities for the last column in the optimal alignment of $A[1..i]$ and $B[1..j]$:

- **Insertion:** The last entry in the top row is empty. In this case, the edit distance is equal to $Edit(i, j-1)+1$. The $+1$ is the cost of the final insertion,

and the recursive expression gives the minimum cost for the remaining alignment.

ALGOR	
ALTR	U

- **Deletion:** The last entry in the bottom row is empty. In this case, the edit distance is equal to $Edit(i-1, j)+1$. The $+1$ is the cost of the final deletion, and the recursive expression gives the minimum cost for the remaining alignment.

ALGO	R
ALTRU	

- **Substitution:** Both rows have characters in the last column. If these two characters are different, then the edit distance is equal to $Edit(i-1, j-1)+1$. If these two characters are equal, the substitution is free, so the edit distance is $Edit(i-1, j-1)$.

ALGO	R
ALTR	U

ALGO	R
ALT	R

This generic case analysis breaks down if either $i = 0$ or $j = 0$, but those boundary cases are easy to handle directly.

- Transforming the empty string into a string of length j requires j insertions, so $Edit(0, j) = j$.
- Transforming a string of length i into the empty string requires i deletions, so $Edit(i, 0) = i$.

As a sanity check, both of these base cases correctly indicate that the edit distance between the empty string and the empty string is zero!

We conclude that the *Edit* function satisfies the following recurrence:

$$Edit(i, j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \left\{ \begin{array}{c} Edit(i, j-1)+1 \\ Edit(i-1, j)+1 \\ Edit(i-1, j-1)+[A[i] \neq B[j]] \end{array} \right\} & \text{otherwise} \end{cases}$$

Dynamic Programming

Now that we have a recurrence, we can transform it into a dynamic programming algorithm following our usual mechanical recipe.

- **Subproblems:** Each recursive subproblem is identified by two indices $0 \leq i \leq m$ and $0 \leq j \leq n$.

- **Memoization structure:** So we can memoize all possible values of $Edit(i, j)$ in a two-dimensional array $Edit[0..m, 0..n]$.
- **Dependencies:** Each entry $Edit[i, j]$ depends only on its three neighboring entries $Edit[i-1, j]$, $Edit[i, j-1]$, and $Edit[i-1, j-1]$.
- **Evaluation order:** If we fill this array in standard row-major order—row by row from top down, each row from left to right—then whenever we reach an entry in the array, all the entries it depends on are already available. (This isn't the *only* evaluation order we could use, but it works, so let's go with it.)

- **Space and time:** The memoization structure uses $O(mn)$ space. We can compute each entry $Edit[i, j]$ in $O(1)$ time once we know its predecessors, so the overall algorithm runs in $\boldsymbol{O(mn)}$ ***time.***

Here is the resulting dynamic programming algorithm:

```
EditDistance(A[1..m], B[1..n]):
  for j ← 0 to n
      Edit[0, j] ← j

  for i ← 1 to m
      Edit[i, 0] ← i
      for j ← 1 to n
          ins ← Edit[i, j − 1] + 1
          del ← Edit[i − 1, j] + 1
          if A[i] = B[j]
              rep ← Edit[i − 1, j − 1]
          else
              rep ← Edit[i − 1, j − 1] + 1
          Edit[i, j] ← min {ins, del, rep}

  return Edit[m, n]
```

This algorithm is most commonly attributed to Robert Wagner and Michael Fischer, who described the algorithm in 1974. However, in full compliance with Stigler's Law of Eponymy, either identical or more general algorithms were independently discovered by Taras Vintsyuk in 1968, V. M. Velichko and N. G. Zagoruyko in 1970, David Sankoff in 1972, Peter Sellers in 1974, and

almost certainly several others.[14] Interestingly, *none* of these authors cite either Levenshtein or Ulam!

The memoization table for the input strings ALGORITHM and ALTRUISTIC is shown below. Bold numbers indicate places where characters in the two strings are equal. The edit distance between ALGORITHM and ALTRUISTIC is indeed six!

		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	**0**	1	2	3	4	5	6	7	8
L	2	1	**0**	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	**4**	5	6
R	4	3	2	2	2	**2**	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	**3**	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	**4**	5	6
I	9	8	7	7	7	7	**6**	5	5	6
C	10	9	8	8	8	8	7	6	6	6

The arrows in this table indicate which predecessor(s) actually define each entry. Each direction of arrow corresponds to a different edit operation: horizontal=deletion, vertical=insertion, and diagonal=substitution. Bold red diagonal arrows indicate "free" substitutions of a letter for itself. Any path of arrows from the top left corner to the bottom right corner of this table represents an optimal edit sequence between the two strings. The example memoization array contains exactly three directed paths from the top left corner to the bottom right corner, each indicating a different sequence of six edits transforming ALGORITHM into ALTRUISTIC, as shown on the next page.

[14]This algorithm is sometimes also *incorrectly* attributed to Saul Needleman and Christian Wunsch in 1970. "The Needleman-Wunsch algorithm" more commonly refers to the standard dynamic programming algorithm for computing the longest common subsequence of two strings (or equivalently, the edit distance where only insertions and deletions are permitted) in $O(mn)$ time, but that attribution is *also* incorrect! In fact, Needleman and Wunsch's algorithm computes (weighted) longest common subsequences (possibly with gap costs) in $O(m^2n^2)$ time, using a different recurrence. Sankoff explicitly describes his $O(mn)$-time algorithm as an improvement of Needleman and Wunsch's algorithm.

```
A L G O R I   T H M
A L T R U I S T I C

A L G O R   I   T H M
A L   T R U I S T I C

A L G O R   I   T H M
A L T   R U I S T I C
```

Our EditDistance algorithm does not actually compute or store any arrows in the table, but the arrow(s) leading into any entry in the table can be reconstructed on the fly in $O(1)$ time from the numerical values. Thus, once we've filled in the table, we can reconstruct the shortest edit sequence in $O(n+m)$ additional time.

3.8 Subset Sum

Recall that the *Subset Sum* problem asks whether any subset of a given array $X[1..n]$ of positive integers sums to a given integer T. In the previous chapter, we developed a recursive Subset Sum algorithm that can be reformulated as follows. Fix the original input array $X[1..n]$ and define the boolean function

$$SS(i,t) = \textsc{True} \text{ if and only if some subset of } X[i..n] \text{ sums to } t.$$

We need to compute $SS(1,T)$. This function satisfies the following recurrence:

$$SS(i,t) = \begin{cases} \textsc{True} & \text{if } t = 0 \\ \textsc{False} & \text{if } t < 0 \text{ or } i > n \\ SS(i+1,t) \vee SS(i+1,t-X[i]) & \text{otherwise} \end{cases}$$

We can transform this recurrence into a dynamic programming algorithm following the usual boilerplate.

- **Subproblems:** Each subproblem is described by an integer i such that $1 \le i \le n+1$, and an integer $t \le T$. However, subproblems with $t < 0$ are trivial, so it seems rather silly to memoize them.[15] Indeed, we can modify the recurrence so that those subproblems never arise:

$$SS(i,t) = \begin{cases} \textsc{True} & \text{if } t = 0 \\ \textsc{False} & \text{if } i > n \\ SS(i+1,t) & \text{if } t < X[i] \\ SS(i+1,t) \vee SS(i+1,t-X[i]) & \text{otherwise} \end{cases}$$

[15] Yes, I'm breaking my own rule against premature optimization.

- **Data structure:** We can memoize our recurrence into a two-dimensional array $S[1..n+1, 0..T]$, where $S[i,t]$ stores the value of $SS(i,t)$.
- **Evaluation order:** Each entry $S[i,t]$ depends on at most two other entries, both of the form $SS[i+1,\cdot]$. So we can fill the array by considering rows from bottom to top in the outer loop, and considering the elements in each row in arbitrary order in the inner loop.
- **Space and time:** The memoization structure uses $O(nT)$ space. If $S[i+1,t]$ and $S[i+1,t-X[i]]$ are already known, we can compute $S[i,t]$ in constant time, so the algorithm runs in $\boldsymbol{O(nT)}$ ***time***.

Here is the resulting dynamic programming algorithm:

```
FastSubsetSum(X[1..n], T):
  S[n+1, 0] ← True
  for t ← 1 to T
      S[n+1, t] ← False

  for i ← n downto 1
      S[i, 0] = True
      for t ← 1 to X[i] − 1
          S[i, t] ← S[i+1, t]        ⟨⟨Avoid the case t < 0⟩⟩
      for t ← X[i] to T
          S[i, t] ← S[i+1, t] ∨ S[i+1, t − X[i]]

  return S[1, T]
```

The worst-case running time $O(nT)$ for this algorithm is a significant improvement over the $O(2^n)$-time recursive backtracking algorithm when T is small.[16] However, if the target sum T is significantly larger than 2^n, this iterative algorithm is actually slower than the naïve recursive algorithm, because it's wasting time solving subproblems that the recursive algorithm never considers. Dynamic programming isn't *always* an improvement![17]

3.9 Optimal Binary Search Trees

The final problem we considered in the previous chapter was the optimal binary search tree problem. The input is a sorted array $A[1..n]$ of search keys and an array $f[1..n]$ of frequency counts, where $f[i]$ is the number of times we will

[16] Even though the subset sum problem is NP-hard, this time bound does *not* imply that P=NP, because T is not necessarily bounded by a polynomial function of the input size.

[17] In the 1967 research memorandum(!) where he proposed memo functions, Donald Michie wrote, "To tabulate values of a function which will not be needed is a waste of space, and to recompute the same values more than once is a waste of time." But in fact, tabulating values of a function that will not be needed is also a waste of *time*!

search for $A[i]$. Our task is to construct a binary search tree for that set such that the total cost of all the searches is as small as possible.

Fix the frequency array f, and let $OptCost(i,k)$ denote the total search time in the optimal search tree for the subarray $A[i..k]$. We derived the following recurrence for the function *OptCost*:

$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ \displaystyle\sum_{j=i}^{k} f[j] + \min_{i \le r \le k} \left\{ \begin{array}{l} OptCost(i, r-1) \\ \quad + OptCost(r+1, k) \end{array} \right\} & \text{otherwise} \end{cases}$$

You can probably guess what we're going to do with this recurrence eventually, but let's rid of that ugly summation first.

For any pair of indices $i \le k$, let $F(i,k)$ denote the total frequency count for all the keys in the interval $A[i..k]$:

$$F(i,k) := \sum_{j=i}^{k} f[j]$$

This function satisfies the following simple recurrence:

$$F(i,k) = \begin{cases} f[i] & \text{if } i = k \\ F(i,k-1) + f[k] & \text{otherwise} \end{cases}$$

We can compute all possible values of $F(i,k)$ in $O(n^2)$ time using—you guessed it!—dynamic programming! The usual mechanical steps give us the following dynamic programming algorithm:

```
InitF(f[1..n]):
  for i ← 1 to n
      F[i, i−1] ← 0
      for k ← i to n
          F[i, k] ← F[i, k−1] + f[k]
```

We will use this short algorithm as an initialization subroutine. This initialization allows us to simplify the original *OptCost* recurrence as follows:

$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ \displaystyle F[i,k] + \min_{i \le r \le k} \left\{ \begin{array}{l} OptCost(i, r-1) \\ \quad + OptCost(r+1, k) \end{array} \right\} & \text{otherwise} \end{cases}$$

Now let's turn the crank.

- **Subproblems:** Each recursive subproblem is specified by two integers i and k, such that $1 \le i \le n+1$ and $0 \le k \le n$.
- **Memoization:** We can store all possible values of *OptCost* in a two-dimensional array $OptCost[1..n+1, 0..n]$. (Only the entries $OptCost[i, j]$ with $j \ge i-1$ will actually be used, but whatever.)
- **Dependencies:** Each entry $OptCost[i, k]$ depends on the entries $OptCost[i, j-1]$ and $OptCost[j+1, k]$, for all j such that $i \le j \le k$. In other words, each table entry depends on all entries either directly to the left or directly below.

The following subroutine fills the entry $OptCost[i, k]$, assuming all the entries it depends on have already been computed.

```
ComputeOptCost(i, k):
  OptCost[i, k] ← ∞
  for r ← i to k
      tmp ← OptCost[i, r − 1] + OptCost[r + 1, k]
      if OptCost[i, k] > tmp
          OptCost[i, k] ← tmp
  OptCost[i, k] ← OptCost[i, k] + F[i, k]
```

- **Evaluation order:** There are at least three different orders that can be used to fill the array. The first one that occurs to most students is to scan through the table one diagonal at a time, starting with the trivial base cases $OptCost[i, i-1]$ and working toward the final answer $OptCost[1, n]$, like so:

```
OptimalBST(f[1..n]):
  InitF(f[1..n])
  for i ← 1 to n + 1
      OptCost[i, i − 1] ← 0
  for d ← 0 to n − 1
      for i ← 1 to n − d        ⟨⟨...or whatever⟩⟩
          ComputeOptCost(i, i + d)
  return OptCost[1, n]
```

We could also traverse the array row by row from the bottom up, traversing each row from left to right, or column by column from left to right, traversing each columns from the bottom up.

```
OptimalBST2(f[1..n]):
  InitF(f[1..n])
  for i ← n+1 downto 1
      OptCost[i, i−1] ← 0
      for j ← i to n
          ComputeOptCost(i, j)
  return OptCost[1, n]
```

```
OptimalBST3(f[1..n]):
  InitF(f[1..n])
  for j ← 0 to n+1
      OptCost[j+1, j] ← 0
      for i ← j downto 1
          ComputeOptCost(i, j)
  return OptCost[1, n]
```

As before, we can illustrate these evaluation orders using a double-lined arrow to indicate the outer loop and single-lined arrows to indicate the inner loop. The bidirectional arrows in the first evaluation order indicate that the order of the inner loops doesn't matter.

- **Time and space:** The memoization structure uses $O(n^2)$ space. No matter which evaluation order we choose, we need $O(n)$ time to compute each entry $OptCost[i,k]$, so our overall algorithm runs in $\boldsymbol{O(n^3)}$ ***time***.

As usual, we could have predicted the final space and time bounds directly from the original recurrence:

$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ F[i,k] + \min\limits_{i \le r \le k} \left\{ \begin{array}{l} OptCost(i, r-1) \\ \quad + OptCost(r+1, k) \end{array} \right\} & \text{otherwise} \end{cases}$$

The *OptCost* function has two arguments, each of which can take on roughly n different values, so we probably need a data structure of size $O(n^2)$. On the other hand, there are *three* variables in the body of the recurrence (i, k, and r), each of which can take roughly n different values, so it should take $O(n^3)$ time to compute everything.

3.10 Dynamic Programming on Trees

So far, all of our dynamic programming examples use multidimensional arrays to store the results of recursive subproblems. However, as the next example shows, this is not always the most appropriate data structure to use.

An ***independent set*** in a graph is a subset of the vertices with no edges between them. Finding the largest independent set in an arbitrary graph is extremely hard; in fact, this is one of the canonical NP-hard problems we will

study in Chapter 12. But in some special classes of graphs, we can find largest independent sets quickly. In particular, when the input graph is a *tree* with n vertices, we can actually compute the largest independent set in $O(n)$ time.

Suppose we are given a tree T. Without loss of generality, suppose T is a rooted tree; that is, there is a special node in T called the *root*, and all edges are implicitly directed away from this vertex. (If T is an unrooted tree—a connected acyclic undirected graph—we can choose an arbitrary vertex as the root.) We call vertex w a *descendant* of vertex v if the unique path from w to the root includes v; equivalently, the descendants of v are v itself and the descendants of the children of v. The *subtree rooted at* v consists of all the descendants of v and the edges between them.

For any node v in T, let $MIS(v)$ denote the size of the largest independent set in the subtree rooted at v. Any independent set in this subtree that excludes v itself is the union of independent sets in the subtrees rooted at the children of v. On the other hand, any independent set that *includes* v necessarily excludes all of v's children, and therefore includes independent sets in the subtrees rooted at v's grandchildren. Thus, the function MIS obeys the following recurrence, where the nonstandard notation $w \downarrow v$ means "w is a child of v":

$$MIS(v) = \max\left\{\sum_{w\downarrow v} MIS(w),\ 1 + \sum_{w\downarrow v}\sum_{x\downarrow w} MIS(x)\right\}$$

We need to compute $MIS(r)$, where r is the root of T.

Figure 3.5. Computing the maximum independent set in a tree

What data structure should we use to memoize this recurrence? The most natural choice is ***the tree T itself!*** Specifically, for each vertex v in T, we store the result of $MIS(v)$ in a new field $v.MIS$. (In principle, we *could* use an array instead, but then we'd need pointers back and forth between each node and its corresponding array entry, so why bother?)

What's a good order to consider the subproblems? The subproblem associated with any node v depends on the subproblems associated with the children and grandchildren of v. So we can visit the nodes in any order we like, provided that every vertex is visited before its parent; in particular, we can use a standard ***post-order*** traversal.

What's the running time of the algorithm? The non-recursive time associated with each node v is proportional to the number of children and grandchildren

of v; this number can be very different from one vertex to the next. But we can turn the analysis around: Each vertex contributes a constant amount of time to its parent and its grandparent! Because each vertex has at most one parent and at most one grandparent, the algorithm runs in ***O(n) time***.

Here is the resulting dynamic programming algorithm. Yes, it's still recursive, because that's the most natural way to implement a post-order tree traversal.

```
TreeMIS(v):
  skipv ← 0
  for each child w of v
      skipv ← skipv + TreeMIS(w)
  keepv ← 1
  for each grandchild x of v
      keepv ← keepv + x.MIS
  v.MIS ← max{keepv, skipv}
  return v.MIS
```

We can derive an even simpler linear-time algorithm by defining two separate functions over the nodes of T:

- Let $MISyes(v)$ denote the size of the largest independent set of the subtree rooted at v that *includes* v.
- Let $MISno(v)$ denote the size of the largest independent set of the subtree rooted at v that *excludes* v.

Again, we need to compute $\max\{MISyes(r), MISno(r)\}$, where r is the root of T. The first two functions satisfy the following mutual recurrence:

$$MISyes(v) = 1 + \sum_{w \downarrow v} MISno(w)$$

$$MISno(v) = \sum_{w \downarrow v} \max\{MISyes(w), MISno(w)\}$$

Again, we can memoize these functions into the tree itself, by defining two new fields for each vertex. A straightforward post-order tree traversal evaluates both functions at every node in ***O(n) time***. The following algorithm not only memoizes both function values at v, it also returns the larger of those two values.

```
TreeMIS2(v):
  v.MISno ← 0
  v.MISyes ← 1
  for each child w of v
      v.MISno ← v.MISno + TreeMIS2(w)
      v.MISyes ← v.MISyes + w.MISno
  return max{v.MISyes, v.MISno}
```

In the second line of the inner loop, we are using the value $w.MISno$ that was memoized by the recursive call in the previous line.

Exercises

For all of the following exercises—and more generally when developing ***any*** new dynamic programming algorithm—I strongly recommend following the steps outlined in Section 3.4. In particular, don't even *start* thinking about tables or for-loops until you have a complete recursive solution, including a clear English specification of the recursive subproblems you are actually solving.[18] **First make it work, then make it fast.**

Sequences/Arrays

1. In a previous life, you worked as a cashier in the lost Antarctican colony of Nadiria, spending the better part of your day giving change to your customers. Because paper is a very rare and valuable resource in Antarctica, cashiers were required by law to use the fewest bills possible whenever they gave change. Thanks to the numerological predilections of one of its founders, the currency of Nadiria, called Dream-Dollars, was available in the following denominations: \$1, \$4, \$7, \$13, \$28, \$52, \$91, and \$365.[19]

♠(a) The greedy change algorithm repeatedly takes the largest bill that does not exceed the target amount. For example, to make \$122 using the greedy algorithm, we first take a \$91 bill, then a \$28 bill, and finally three \$1 bills. Give an example where this greedy algorithm uses more Dream-Dollar bills than the minimum possible. *[Hint: It may be easier to write a small program than to work this out by hand.]*

(b) Describe and analyze a recursive algorithm that computes, given an integer k, the minimum number of bills needed to make k Dream-Dollars. (Don't worry about making your algorithm fast; just make sure it's correct.)

(c) Describe a dynamic programming algorithm that computes, given an integer k, the minimum number of bills needed to make k Dream-Dollars. (This one needs to be fast.)

[18] In my algorithms classes, any dynamic programming solution that does *not* include an English specification of the underlying recursive subproblems automatically gets a score of *zero*, even if the solution is otherwise perfect. Introducing this policy significantly improved students' grades, because it significantly reduced the number of times they submitted incorrect (or incoherent) dynamic programming algorithms.

[19] For more details on the history and culture of Nadiria, including images of the various denominations of Dream-Dollars, see http://moneyart.biz/dd/.

2. Describe efficient algorithms for the following variants of the text segmentation problem. Assume that you have a subroutine IsWord that takes an array of characters as input and returns True if and only if that string is a "word". Analyze your algorithms by bounding the number of calls to IsWord.

   (a) Given an array $A[1..n]$ of characters, compute the number of partitions of A into words. For example, given the string `ARTISTOIL`, your algorithm should return 2, for the partitions `ARTIST·OIL` and `ART·IS·TOIL`.

   (b) Given two arrays $A[1..n]$ and $B[1..n]$ of characters, decide whether A and B can be partitioned into words at the same indices. For example, the strings `BOTHEARTHANDSATURNSPIN` and `PINSTARTRAPSANDRAGSLAP` can be partitioned into words at the same indices as follows:

   `BOT·HEART·HAND·SAT·URNS·PIN`
   `PIN·START·RAPS·AND·RAGS·LAP`

   (c) Given two arrays $A[1..n]$ and $B[1..n]$ of characters, compute the number of different ways that A and B can be partitioned into words at the same indices.

3. Suppose you are given an array $A[1..n]$ of numbers, which may be positive, negative, or zero, and which are ***not*** necessarily integers.

   (a) Describe and analyze an algorithm that finds the largest sum of elements in a contiguous subarray $A[i..j]$.

   (b) Describe and analyze an algorithm that finds the largest *product* of elements in a contiguous subarray $A[i..j]$.

   For example, given the array $[-6, 12, -7, 0, 14, -7, 5]$ as input, your first algorithm should return 19, and your second algorithm should return 504.

   Given the one-element array $[-374]$ as input, your first algorithm should return 0, and your second algorithm should return 1. (The empty interval is still an interval!) For the sake of analysis, assume that comparing, adding, or multiplying any pair of numbers takes $O(1)$ time.

   [Hint: Part (a) has been a standard computer science interview question since at least the mid-1980s. You can find many correct solutions on the web; the problem even has its own Wikipedia page! But at least in 2016, a significant fraction of the solutions I found on the web for part (b) were either slower than necessary or actually incorrect.]

4. This exercise explores variants of the maximum-subarray problem (Problem 3). In all cases, your input consists of an array $A[1..n]$ of real numbers (which could be positive, negative, or zero) and possibly an additional integer $X \geq 0$.

   (a) *Wrapping around:* Suppose A is a *circular* array. In this setting, a "contiguous subarray" can be either an interval $A[i..j]$ or a suffix followed by a prefix $A[i..n] \cdot A[1..j]$. Describe and analyze an algorithm that finds a contiguous subarray of A with the largest sum.

   (b) *Long subarrays only:* Describe and analyze an algorithm that finds a contiguous subarray of A *of length at least* X that has the largest sum. (Assume $X \leq n$.)

   (c) *Short subarrays only:* Describe and analyze an algorithm that finds a contiguous subarray of A *of length at most* X that has the largest sum.

   (d) *The Price Is Right:* Describe and analyze an algorithm that finds a contiguous subarray of A with the largest sum *less than or equal to* X.

   (e) Describe a faster algorithm for Problem 4(d) when every number in the array A is non-negative.

5. This exercise asks you to develop efficient algorithms to find optimal *subsequences* of various kinds. A subsequence is anything obtained from a sequence by extracting a subset of elements, but keeping them in the same order; the elements of the subsequence need not be contiguous in the original sequence. For example, the strings `C`, `DAMN`, `YAIOAI`, and `DYNAMICPROGRAMMING` are all subsequences of the string `DYNAMICPROGRAMMING`.

   [Hint: Exactly one of these problems can be solved in $O(n)$ time using a greedy algorithm.]

   (a) Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A ***common subsequence*** of A and B is another sequence that is a subsequence of both A and B. Describe an efficient algorithm to compute the length of the *longest* common subsequence of A and B.

   (b) Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A ***common supersequence*** of A and B is another sequence that contains both A and B as subsequences. Describe an efficient algorithm to compute the length of the *shortest* common supersequence of A and B.

   (c) Call a sequence $X[1..n]$ of numbers ***bitonic*** if there is an index i with $1 < i < n$, such that the prefix $X[1..i]$ is increasing and the suffix $X[i..n]$ is decreasing. Describe an efficient algorithm to compute the length of the longest bitonic subsequence of an arbitrary array A of integers.

(d) Call a sequence $X[1..n]$ of numbers ***oscillating*** if $X[i] < X[i+1]$ for all even i, and $X[i] > X[i+1]$ for all odd i. Describe an efficient algorithm to compute the length of the longest oscillating subsequence of an arbitrary array A of integers.

(e) Describe an efficient algorithm to compute the length of the shortest oscillating supersequence of an arbitrary array A of integers.

(f) Call a sequence $X[1..n]$ of numbers ***convex*** if $2 \cdot X[i] < X[i-1] + X[i+1]$ for all i. Describe an efficient algorithm to compute the length of the longest convex subsequence of an arbitrary array A of integers.

(g) Call a sequence $X[1..n]$ of numbers ***weakly increasing*** if each element is larger than the average of the two previous elements; that is, $2 \cdot X[i] > X[i-1] + X[i-2]$ for all $i > 2$. Describe an efficient algorithm to compute the length of the longest weakly increasing subsequence of an arbitrary array A of integers.

(h) Call a sequence $X[1..n]$ of numbers ***double-increasing*** if $X[i] > X[i-2]$ for all $i > 2$. (In other words, a double-increasing sequence is a perfect shuffle of two increasing sequences.) Describe an efficient algorithm to compute the length of the longest double-increasing subsequence of an arbitrary array A of integers.

(i) Recall that a sequence $X[1..n]$ of numbers is *increasing* if $X[i] < X[i+1]$ for all i. Describe an efficient algorithm to compute the length of the *longest common increasing subsequence* of two given arrays of integers. For example, $\langle 1, 4, 5, 6, 7, 9 \rangle$ is the longest common increasing subsequence of the sequences $\langle 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3 \rangle$ and $\langle 1, 4, 1, 4, 2, 1, 3, 5, 6, 2, 3, 7, 3, 0, 9, 5 \rangle$.

6. A *shuffle* of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the string BANANAANANAS is a shuffle of the strings BANANA and ANANAS in several different ways.

$$\text{BANANA}_{\text{ANANAS}} \qquad \text{BAN}_{\text{ANA}}\text{ANA}_{\text{NAS}} \qquad \text{B}_{\text{AN}}\text{AN}_{\text{A}}\text{A}_{\text{NA}}\text{NA}_{\text{S}}$$

Similarly, the strings PRODGYRNAMAMMIINCG and DYPRONGARMAMMICING are both shuffles of DYNAMIC and PROGRAMMING:

$$\text{PRO}^{\text{D}}\text{G}^{\text{Y}}\text{R}^{\text{NAM}}\text{A}_{\text{MMI}}\text{I}_{\text{N}}\text{C}_{\text{G}} \qquad \text{DY}_{\text{PRO}}\text{N}_{\text{G}}\text{A}_{\text{R}}\text{M}_{\text{AMM}}\text{I}^{\text{IC}}\text{ING}$$

(a) Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B.

(b) A ***smooth*** shuffle of X and Y is a shuffle of X and Y that never uses more than two consecutive symbols of either string. For example,

- PRDOYGNARAMMMIICNG is a smooth shuffle of the strings DYNAMIC and PROGRAMMING.
- DYPRNOGRAAMMMICING is a shuffle of DYNAMIC and PROGRAMMING, but it is not a smooth shuffle (because of the substrings OGR and ING).
- XXXXXXXXXXXXXXXXXXXXXXX is a smooth shuffle of the strings XXXXXXX and XXXXXXXXXXX.
- There is no smooth shuffle of the strings XXXX and XXXXXXXXXXXX.

Describe and analyze an algorithm to decide, given three strings X, Y, and Z, whether Z is a smooth shuffle of X and Y.

7. For each of the following problems, the input consists of two arrays $X[1..k]$ and $Y[1..n]$ where $k \leq n$.

   (a) Describe and analyze an algorithm to decide whether X is a subsequence of Y. For example, the string PPAP is a subsequence of the string PENPINEAPPLEAPPLEPEN.

   (b) Describe and analyze an algorithm to find the smallest number of symbols that can be removed from Y so that X is no longer a subsequence. Equivalently, your algorithm should find the longest subsequence of Y that is *not* a supersequence of X. For example, after removing removing two symbols from the string PENPINEAPPLEAPPLEPEN, the string PPAP is no longer a subsequence.

   ♥(c) Describe and analyze an algorithm to determine whether X occurs as two *disjoint* subsequences of Y. For example, the string PPAP appears as two disjoint subsequences in the string PENPINEAPPLEAPPLEPEN.

   (d) Suppose the input also includes a third array $C[1..n]$ of numbers, which may be positive, negative, or zero, where $C[i]$ is the *cost* of $Y[i]$. Describe and analyze an algorithm to compute the minimum-cost occurrence of X as a subsequence of Y. That is, we want to find an array $I[1..k]$ such that $I[j] < I[j+1]$ and $X[I[j]] = Y[j]$ for every index j, and the total cost $\sum_{j=1}^{k} C[j]$ is as small as possible.

   (e) Describe and analyze an algorithm to compute the total number of (possibly overlapping) occurrences of X as a subsequence of Y. For purposes of analysis, assume that we can add two arbitrary integers in $O(1)$ time. For example, the string PPAP appears exactly 23 times as a subsequence of the string PENPINEAPPLEAPPLEPEN. If all characters in X and Y are equal, your algorithm should return $\binom{n}{k}$.

   (f) What is the running time of your algorithm for part (d) if adding two ℓ-bit integers requires $O(\ell)$ time?

8. Describe and analyze an efficient algorithm to find the length of the longest contiguous substring that appears both forward and backward in an input string $T[1..n]$. The forward and backward substrings must not overlap. Here are several examples:
   - Given the input string ALGORITHM, your algorithm should return 0.
   - Given the input string RECURSION, your algorithm should return 1, for the substring R.
   - Given the input string REDIVIDE, your algorithm should return 3, for the substring EDI. (The forward and backward substrings must not overlap!)
   - Given the input string DYNAMICPROGRAMMINGMANYTIMES, your algorithm should return 4, for the substring YNAM. (In particular, it should *not* return 6, for the subsequence YNAMIR).

9. A palindrome is any string that is exactly the same as its reversal, like I, or DEED, or RACECAR, or AMANAPLANACATACANALPANAMA.

   (a) Describe and analyze an algorithm to find the length of the *longest subsequence* of a given string that is also a palindrome.

   For example, the longest palindrome subsequence of the string **M**A**H**D**Y**NA**M**ICP**RO**G**R**AMZLET**M**ESHOW**Y**OUT**H**E**M** is MHYMRORMYHM; thus, given that string as input, your algorithm should return 11.

   (b) Describe and analyze an algorithm to find the length of the *shortest supersequence* of a given string that is also a palindrome. For example, the shortest palindrome supersequence of TWENTYONE is **TWENT**O**YO**T**NE**WT, so given the string TWENTYONE as input, your algorithm should return 13.

   (c) Any string can be decomposed into a sequence of palindromes. For example, the string BUBBASEESABANANA ("Bubba sees a banana.") can be broken into palindromes in the following ways (and 65 others):

   BUB • BASEESAB • ANANA

   B • U • BB • ASEESA • B • ANANA

   BUB • B • A • SEES • ABA • N • ANA

   B • U • BB • A • S • EE • S • A • B • A • NAN • A

   B • U • B • B • A • S • E • E • S • A • B • A • N • A • N • A

   Describe and analyze an efficient algorithm to find the smallest number of palindromes that make up a given input string. For example, given the input string BUBBASEESABANANA, your algorithm should return 3.

   (d) Describe and analyze an efficient algorithm to find the largest integer k such that a given string can be split into palindromes of length at least k. For example:

- Given the string PALINDROME, your algorithm should return 1.
- Given the string BUBBASEESABANANA, your algorithm should return 3, for the partition BUB • BASEESAB • ANANA.
- Given a string of n identical symbols, your algorithm should return n.

(e) Describe and analyze an efficient algorithm to find the number of different ways that a given string can be decomposed into palindromes. For example:

- Given the string PALINDROME, your algorithm should return 1.
- Given the string BUBBASEESABANANA, your algorithm should return 70.
- Given a string of n identical symbols, your algorithm should return 2^{n-1}.

♥(f) A ***metapalindrome*** is a decomposition of a string into a sequence of palindromes, such that the sequence of palindrome lengths is itself a palindrome. For example:

BOB • S • MAM • ASEESA • UKU • L • ELE

is a metapalindrome for the string BOBSMAMASEESAUKULELE, whose length sequence is the palindrome $(3, 1, 3, 6, 3, 1, 3)$. Describe and analyze an efficient algorithm to find the length of the shortest metapalindrome for a given string. For example, given the input string BOBSMAMASEESAUKULELE, your algorithm should return 11.

10. Suppose you are given an array $A[1..n]$ of positive integers. An *increasing back-and-forth subsequence* is an sequence of indices $I[1..\ell]$ with the following properties:

- $1 \leq I[j] \leq n$ for all j.
- $A[I[j]] < A[I[j+1]]$ for all $j < \ell$.
- If $I[j]$ is even, then $I[j+1] > I[j]$.
- If $I[j]$ is odd, then $I[j+1] < I[j]$.

Less formally, suppose we are given an array of n squares, each containing a positive integer. Suppose we place a token on one of the squares, and then repeatedly move the token left (if it's on an odd-indexed square) or right (if it's on an even-indexed square), always moving from a smaller number to a larger number. Then the sequence of token positions is an increasing back-and-forth subsequence.

Describe an algorithm to compute the length of the longest increasing back-and-forth subsequence of a given array of n integers. For example, given the input array

1	1	8	7	5	6	3	6	4	4	8	3	9	1	2	2	3	9	4	0
1<	2>	3<	4>	5<	6>	7<	8>	9<	10>	11<	12>	13<	14>	15<	16>	17<	18>	19<	20>

your algorithm should return the integer 9, which is the length of the following increasing back-and-forth subsequence:

0	1	2	3	4	6	7	8	9
20>	1<	15<	18>	10>	6>	4>	3<	13<

11. Suppose we want to typeset a paragraph of text onto a piece of paper (or if you insist, a computer screen). The text consists of a sequence of n words, where the ith word has length $\ell[i]$. We want to break the paragraph into several lines of total length exactly L. For example, according to TEX, the program used to typeset these notes, *the paragraph you are reading right now* is approximately 11.94794 cm ≈ 4.7055 inches wide.

    Depending on how the paragraph is broken into lines of text, we must insert different amounts of white space between the words. The paragraph should be fully justified, meaning that the first character on each line starts at the left margin, and *except for the last line*, the last character on each line ends at the right margin. There must be at least one unit of white space between any two words on the same line. See *the paragraph you are reading right now*? Just like that.

    Define the *slop* of a paragraph layout as the sum over all lines, *except the last*, of the cube of the amount of extra white-space in each line, not counting the one unit of required space between each adjacent pair of words. Specifically, if a line contains words i through j, then the slop of that line is defined to be $\left(L - j + i - \sum_{k=i}^{j} \ell[k]\right)^3$. Describe a dynamic programming algorithm to print the paragraph with minimum slop.

12. You and your eight-year-old nephew Elmo decide to play a simple card game. At the beginning of the game, the cards are dealt face up in a long row. Each card is worth a different number of points. After all the cards are dealt, you and Elmo take turns removing either the leftmost or rightmost card from the row, until all the cards are gone. At each turn, you can decide which of the two cards to take. The winner of the game is the player that has collected the most points when the game ends.

    Having never taken an algorithms class, Elmo follows the obvious greedy strategy—when it's his turn, Elmo *always* takes the card with the higher point value. Your task is to find a strategy that will beat Elmo whenever possible. (It might seem mean to beat up on a little kid like this, but Elmo absolutely *hates* it when grown-ups let him win.)

    (a) Prove that you should not also use the greedy strategy. That is, show that there is a game that you can win, but only if you do *not* follow the same greedy strategy as Elmo.

(b) Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against Elmo.

♣(c) When Elmo was four, he used an even simpler strategy—on his turn, he always chose his next card uniformly at random. That is, if there was more than one card left on his turn, he would take the leftmost card with probability 1/2, and the rightmost card with probability 1/2. Describe an algorithm to determine, given the initial sequence of cards, the maximum *expected* number of points you can collect playing against four-year-old-Elmo.

(d) Five years later, thirteen-year-old Elmo has become a *much* stronger player. Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against a *perfect* opponent.

13. It's almost time to show off your flippin' sweet dancing skills! Tomorrow is the big dance contest you've been training for your entire life, except for that summer you spent with your uncle in Alaska hunting wolverines. You've obtained an advance copy of the list of n songs that the judges will play during the contest, in chronological order. Yessssssssss!

You know all the songs, all the judges, and your own dancing ability extremely well. For each integer k, you know that if you dance to the kth song on the schedule, you will be awarded exactly $Score[k]$ points, but then you will be physically unable to dance for the next $Wait[k]$ songs (that is, you cannot dance to songs $k+1$ through $k+Wait[k]$). The dancer with the highest total score at the end of the night wins the contest, so you want your total score to be as high as possible.

Describe and analyze an efficient algorithm to compute the maximum total score you can achieve. The input to your sweet algorithm is the pair of arrays $Score[1..n]$ and $Wait[1..n]$.

14. The new swap-puzzle game *Candy Swap Saga XIII* involves n cute animals numbered from 1 to n. Each animal holds one of three types of candy: circus peanuts, Heath bars, and Cioccolateria Gardini chocolate truffles. You also have a candy in your hand; at the start of the game, you have a circus peanut.

To earn points, you visit each of the animals in order from 1 to n. For each animal, you can either keep the candy in your hand or exchange it with the candy the animal is holding.

- If you swap your candy for another candy of the *same* type, you earn one point.

- If you swap your candy for a candy of a *different* type, you lose one point. (Yes, your score can be negative.)
- If you visit an animal and decide not to swap candy, your score does not change.

You *must* visit the animals in order, and once you visit an animal, you can never visit it again.

Describe and analyze an efficient algorithm to compute your maximum possible score. Your input is an array $C[1..n]$, where $C[i]$ is the type of candy that the ith animal is holding.

15. Lenny Rutenbar, the founding dean of the new Maksymilian R. Levchin College of Computer Science, has commissioned a series of snow ramps on the south slope of the Orchard Downs sledding hill[20] and challenged Bill Kudeki, head of the Department of Electrical and Computer Engineering, to a sledding contest. Bill and Lenny will both sled down the hill, each trying to maximize their air time. The winner gets to expand their department/college into both Siebel Center and the new ECE Building; the loser has to move their entire department/college in the Boneyard culvert next to Loomis Lab.

    Whenever Lenny or Bill reaches a ramp *while on the ground*, they can either use that ramp to jump through the air, possibly flying over one or more ramps, or sled past that ramp and stay on the ground. Obviously, if someone flies over a ramp, they cannot use that ramp to extend their jump.

    (a) Suppose you are given a pair of arrays $Ramp[1..n]$ and $Length[1..n]$, where $Ramp[i]$ is the distance from the top of the hill to the ith ramp, and $Length[i]$ is the distance that any sledder who takes the ith ramp will travel through the air. Describe and analyze an algorithm to determine the maximum total distance that Lenny or Bill can spend in the air.

    (b) The university lawyers heard about Lenny and Bill's little bet and immediately objected. To protect the university from either lawsuits or sky-rocketing insurance rates, they impose an upper bound on the number of jumps that either sledder can take. Describe and analyze an algorithm to determine the maximum total distance that Lenny or Bill can spend in the air *with at most k jumps*, given the original arrays $Ramp[1..n]$ and $Length[1..n]$ and the integer k as input.

    ♥(c) When the lawyers realized that imposing their restriction didn't immediately shut down the contest, they added a new restriction: No ramp can be used more than once! Disgusted by the legal interference, Lenny and Bill give up on their bet and decide to cooperate to put on a good show

[20] The north slope is faster, but too short for an interesting contest.

for the spectators. Describe and analyze an algorithm to determine the maximum total distance that Lenny and Bill can spend in the air, each taking at most k jumps (so at most $2k$ jumps total), and with each ramp used at most once.

16. Farmers Boggis, Bunce, and Bean have set up an obstacle course for Mr. Fox. The course consists of a long row of booths, each with a number painted on the front with bright red paint. Formally, Mr. Fox is given an array $A[1..n]$, where $A[i]$ is the number painted on the front of the ith booth. Each number $A[i]$ could be positive, negative, or zero. Everyone agrees with the following rules:

    - At each booth, Mr. Fox ***must*** say either "Ring!" or "Ding!".
    - If Mr. Fox says "Ring!" at the ith booth, he earns a reward of $A[i]$ chickens. (If $A[i] < 0$, Mr. Fox pays a penalty of $-A[i]$ chickens.)
    - If Mr. Fox says "Ding!" at the ith booth, he pays a penalty of $A[i]$ chickens. (If $A[i] < 0$, Mr. Fox earns a reward of $-A[i]$ chickens.)
    - Mr. Fox is forbidden to say the same word more than three times in a row. For example, if he says "Ring!" at booths 6, 7, and 8, then he must say "Ding!" at booth 9.
    - All accounts will be settled at the end, after Mr. Fox visits every booth and the umpire calls "Hot box!" Mr. Fox does not actually have to carry chickens through the obstacle course.
    - Finally, if Mr. Fox violates any of the rules, or if he ends the obstacle course owing the farmers chickens, the farmers will shoot him.

    Describe and analyze an algorithm to compute, the largest number of chickens that Mr. Fox can earn by running the obstacle course, given the array $A[1..n]$ of numbers as input. *[Hint: Watch out for the burning pine cone!]*

17. ***Dance Dance Revolution*** is a dance video game, first introduced in Japan by Konami in 1998. Players stand on a platform marked with four arrows, pointing forward, back, left, and right, arranged in a cross pattern. During play, the game plays a song and scrolls a sequence of n arrows (🡐, 🡑, 🡓, or 🡒) from the bottom to the top of the screen. At the precise moment each arrow reaches the top of the screen, the player must step on the corresponding arrow on the dance platform. (The arrows are timed so that you'll step with the beat of the song.)

    You are playing a variant of this game called "Vogue Vogue Revolution", where the goal is to play perfectly but move as little as possible. When an arrow reaches the top of the screen, if one of your feet is already on the

correct arrow, you are awarded one style point for maintaining your current pose. If neither foot is on the right arrow, you must move one (and *only* one) foot from its current location to the correct arrow on the platform. If you ever step on the wrong arrow, or fail to step on the correct arrow, or move more than one foot at a time, or move either foot when you are already standing on the correct arrow, all your style points are taken away and you lose the game.

How should you move your feet to maximize your total number of style points? For purposes of this problem, assume you always start with your left foot on ← and your right foot on →, and that you've memorized the entire sequence of arrows. For example, if the sequence is ↑↑↓↓←→←→, you can earn 5 style points by moving your feet as shown below:

(a) ***Prove*** that for *any* sequence of n arrows, it is possible to earn at least $n/4-1$ style points.

(b) Describe an efficient algorithm to find the maximum number of style points you can earn during a given VVR routine. The input to your algorithm is an array $Arrow[1..n]$ containing the sequence of arrows.

18. Consider the following solitaire form of Scrabble. We begin with a fixed, finite sequence of tiles; each tile has both a letter and a numerical value. At the start of the game, we draw the first seven tiles from the sequence and put them into our hand. In each turn, we form an English word from some or all of the tiles in our hand, place those tiles on the table, and receive the total value of those tiles as points. (If no English word can be formed from the tiles in our hand, the game immediately ends.) Then we repeatedly draw the next tile from the start of the sequence until either (a) we have seven tiles in our hand, or (b) the sequence is empty. (Sorry, no double/triple word/letter scores, bingos, blanks, or passing.) Our goal is to obtain as many points as possible.

For example, consider the following sequence of 20 tiles:

I_2	N_2	X_8	A_1	N_2	A_1	D_3	U_5	D_3	I_2	D_3	K_8	U_5	B_4	L_2	A_1	K_8	H_5	A_1	N_2

Given this sequence of tiles at the beginning of the game, we can earn 68 points as follows:

- We initially draw I_2 N_2 X_8 A_1 N_2 A_1 D_3.
- Play the word N_2 A_1 I_2 A_1 D_3 for 9 points, leaving N_2 X_8 in hand.

- Draw the next five tiles $\boxed{\mathtt{U}_5}$ $\boxed{\mathtt{D}_3}$ $\boxed{\mathtt{I}_2}$ $\boxed{\mathtt{D}_3}$ $\boxed{\mathtt{K}_8}$.
- Play the word $\boxed{\mathtt{U}_5}$ $\boxed{\mathtt{N}_2}$ $\boxed{\mathtt{D}_3}$ $\boxed{\mathtt{I}_2}$ $\boxed{\mathtt{D}_3}$ for 15 points, leaving $\boxed{\mathtt{K}_8}$ $\boxed{\mathtt{X}_8}$ in hand.
- Draw the next five tiles $\boxed{\mathtt{U}_5}$ $\boxed{\mathtt{B}_4}$ $\boxed{\mathtt{L}_2}$ $\boxed{\mathtt{A}_1}$ $\boxed{\mathtt{K}_8}$.
- Play the word $\boxed{\mathtt{B}_4}$ $\boxed{\mathtt{U}_5}$ $\boxed{\mathtt{L}_2}$ $\boxed{\mathtt{K}_8}$ for 19 points, leaving $\boxed{\mathtt{K}_8}$ $\boxed{\mathtt{X}_8}$ $\boxed{\mathtt{A}_1}$ in hand.
- Draw the last three tiles $\boxed{\mathtt{H}_5}$ $\boxed{\mathtt{A}_1}$ $\boxed{\mathtt{N}_2}$.
- Play the word $\boxed{\mathtt{A}_1}$ $\boxed{\mathtt{N}_2}$ $\boxed{\mathtt{K}_8}$ $\boxed{\mathtt{H}_5}$ for 16 points, leaving $\boxed{\mathtt{X}_8}$ $\boxed{\mathtt{A}_1}$ in hand.
- Play the word $\boxed{\mathtt{A}_1}$ $\boxed{\mathtt{X}_8}$ for 9 points, emptying our hand and ending the game.

(a) Suppose the sequence of tiles is represented by two arrays $Letter[1..n]$, containing a sequence of letters between A and Z, and $Value[\mathtt{A}..\mathtt{Z}]$, where $Value[\ell]$ is the value of any tile with letter ℓ. Design and analyze an efficient algorithm to compute the maximum number of points that can be earned from the given sequence of tiles.

(b) Now suppose two tiles with the same letter might have different values. Now the tile sequence is represented by two arrays $Letter[1..n]$ and $Value[1..n]$, where $Value[i]$ is the value of the ith tile. Design and analyze an efficient algorithm to compute the maximum number of points that can be earned from the given sequence of tiles.

In both problems, the output is a single number: the maximum possible score. Assume (because it's true) that you can find all English words that can be made from any set of at most seven tiles, along with the point values of those words, in $O(1)$ time.

19. Suppose we are given a set L of n line segments in the plane, where each segment has one endpoint on the line $y = 0$ and one endpoint on the line $y = 1$, and all $2n$ endpoints are distinct.

(a) Describe and analyze an algorithm to compute the largest subset of L in which no pair of segments intersects.

(b) Describe and analyze an algorithm to compute the largest subset of L in which ***every*** pair of segments intersects.

Now suppose we are given a set L of n line segments in the plane, where both endpoints of each segment lie on the unit circle $x^2 + y^2 = 1$, and all $2n$ endpoints are distinct.

(c) Describe and analyze an algorithm to compute the largest subset of L in which no pair of segments intersects.

(d) Describe and analyze an algorithm to compute the largest subset of L in which ***every*** pair of segments intersects.

20. Let P be a set of n points evenly distributed on the unit circle, and let S be a set of m line segments with endpoints in P. The endpoints of the m segments are *not* necessarily distinct; n could be significantly smaller than $2m$.

    (a) Describe an algorithm to find the size of the largest subset of segments in S such that every pair is disjoint. Two segments are disjoint if they do not intersect even at their endpoints.

    (b) Describe an algorithm to find the size of the largest subset of segments in S such that every pair is interior-disjoint. Two segments are interior-disjoint if their intersection is either empty or an endpoint of both segments.

    (c) Describe an algorithm to find the size of the largest subset of segments in S such that every pair intersects.

    (d) Describe an algorithm to find the size of the largest subset of segments in S such that every pair crosses. Two segments cross if they intersect but not at their endpoints.

    For full credit, all four algorithms should run in $O(mn)$ time.

21. You are driving a bus along a highway, full of rowdy, hyper, thirsty students and a soda fountain machine. Each minute that a student is on your bus, that student drinks one ounce of soda. Your goal is to drop the students off quickly, so that the total amount of soda consumed by all students is as small as possible.

    You know how many students will get off of the bus at each exit. Your bus begins somewhere along the highway (probably not at either end) and moves at a constant speed of 37.4 miles per hour. You must drive the bus along the highway; however, you may drive forward to one exit then backward to an exit in the opposite direction, switching as often as you like. (You can stop the bus, drop off students, and turn around instantaneously.)

    Describe an efficient algorithm to drop the students off so that they drink as little soda as possible. Your input consists of the bus route (a list of the exits, together with the travel time between successive exits), the number of students you will drop off at each exit, and the current location of your bus (which you may assume is an exit).

22. Let's define a *summary* of two strings A and B to be a concatenation of substrings of the following form:

    - ▲SNA indicates a substring SNA of only the first string A.
    - ♦FOO indicates a common substring FOO of both strings.
    - ▼BAR indicates a substring BAR of only the second string B.

A summary is *valid* if we can recover the original strings A and B by concatenating the appropriate substrings of the summary in order and discarding the delimiters ▲, ♦, and ▼. Each regular character has length 1, and each delimiter ▲, ♦, or ▼ has some fixed non-negative length Δ. The *length* of a summary is the sum of the lengths of its symbols.

For example, each of the following strings is a valid summary of the strings KITTEN and KNITTING:

- ♦K▼N♦ITT▲E▼I♦N▼G has length $9 + 7\Delta$.
- ♦K▼N♦ITT▲EN▼ING has length $10 + 5\Delta$.
- ♦K▲ITTEN▼NITTING has length $13 + 3\Delta$.
- ▲KITTEN▼KNITTING has length $14 + 2\Delta$.

Describe and analyze an algorithm that computes the length of the shortest summary of two given strings $A[1..m]$ and $B[1..n]$. The delimiter length Δ is also part of the input to your algorithm. For example:

- Given strings KITTEN and KNITTING and $\Delta = 0$, your algorithm should return 9.
- Given strings KITTEN and KNITTING and $\Delta = 1$, your algorithm should return 15.
- Given strings KITTEN and KNITTING and $\Delta = 2$, your algorithm should return 18.

23. *Vankin's Mile* is an American solitaire game played on an $n \times n$ square grid. The player starts by placing a token on any square of the grid. Then on each turn, the player moves the token either one square to the right or one square down. The game ends when player moves the token off the edge of the board. Each square of the grid has a numerical value, which could be positive, negative, or zero. The player starts with a score of zero; whenever the token lands on a square, the player adds its value to his score. The object of the game is to score as many points as possible.

For example, given the grid below, the player can score $8-6+7-3+4 = 10$ points by placing the initial token on the 8 in the second row, and then moving down, down, right, down, down. (This is *not* the best possible score for this grid of numbers.)

−1	7	−8	10	−5
−4	−9	**8**	−6	0
5	−2	**−6**	−6	7
−7	4	**7**	**−3**	−3
7	1	−6	**4**	−9

(a) Describe and analyze an efficient algorithm to compute the maximum possible score for a game of Vankin's Mile, given the $n \times n$ array of values as input.

(b) In the European version of this game, appropriately called *Vankin's Kilometer*, the player can move the token either one square down, one square right, *or one square left* in each turn. However, to prevent infinite scores, the token cannot land on the same square more than once. Describe and analyze an efficient algorithm to compute the maximum possible score for a game of Vankin's Kilometer, given the $n \times n$ array of values as input.[21]

24. Suppose you are given an $m \times n$ bitmap as an array $M[1..n, 1..n]$ of 0s and 1s. A *solid block* in M is a subarray of the form $M[i..i', j..j']$ in which all bits are equal. A solid block is square if it has the same number of rows and columns.

(a) Describe an algorithm to find the maximum area of a solid *square* block in M in $O(n^2)$ time.

(b) Describe an algorithm to find the maximum area of a solid block in M in $O(n^3)$ time.

(c) Describe an algorithm to find the maximum area of a solid block in M in $O(n^2 \log n)$ time. *[Hint: Divide and conquer.]*

♥(d) Describe an algorithm to find the maximum area of a solid block in M in $O(n^2)$ time.

25. Suppose you are given an array $M[1..n, 1..n]$ of numbers, which may be positive, negative, or zero, and which are *not* necessarily integers. Describe an algorithm to find the largest sum of elements in any rectangular subarray of the form $M[i..i', j..j']$. For full credit, your algorithm should run in $O(n^3)$ time. *[Hint: See problem 3.]*

26. Describe and analyze an algorithm that finds the maximum-area rectangular pattern that appears more than once in a given bitmap. Specifically, given a two-dimensional array $M[1..n, 1..n]$ of bits as input, your algorithm should output the area of the largest repeated rectangular pattern in M. For example, given the bitmap shown on the left in the figure below, your algorithm should return the integer 195, which is the area of the 15×13 doggo. (Although it doesn't happen in this example, the two copies of the repeated pattern might overlap.)

[21] If we also allowed *upward* movement, the resulting game (Vankin's Fathom?) would be NP-hard.

(a) For full credit, describe an algorithm that runs in $O(n^5)$ time.

♥(b) For extra credit, describe an algorithm that runs in $O(n^4)$ time.

♣♥(c) For extra extra credit, describe an algorithm that runs in $O(n^3 \operatorname{polylog} n)$ time.

27. Let P be a set of points in the plane in *convex position*. Intuitively, if a rubber band were wrapped around the points, then every point would touch the rubber band. More formally, for any point p in P, there is a line that separates p from the other points in P. Moreover, suppose the points are indexed $P[1], P[2], \ldots, P[n]$ in counterclockwise order around the "rubber band", starting with the leftmost point $P[1]$.

This problem asks you to solve a special case of the traveling salesman problem, where the salesman must visit every point in P, and the cost of moving from one point $p \in P$ to another point $q \in P$ is the Euclidean distance $|pq|$.

(a) Describe a simple algorithm to compute the shortest *cyclic* tour of P.

(b) A *simple* tour is one that never crosses itself. Prove that the shortest tour of P must be simple.

(c) Describe and analyze an efficient algorithm to compute the shortest tour of P that starts at the leftmost point $P[1]$ and ends at the rightmost point $P[r]$.

(d) Describe and analyze an efficient algorithm to compute the shortest tour of P, with no restrictions on the endpoints.

♥28. Describe and analyze an algorithm to solve the traveling salesman problem in $O(2^n \operatorname{poly}(n))$ time. Given an undirected n-vertex graph G with weighted edges, your algorithm should return the weight of the lightest cycle in G that visits every vertex exactly once, or ∞ if G has no such cycles. *[Hint: The obvious recursive backtracking algorithm takes $O(n!)$ time.]*

29. Let $W = \{w_1, w_2, \ldots, w_n\}$ be a finite set of strings over some fixed alphabet Σ. An *edit center* for W is a string $C \in \Sigma^*$ such that the maximum edit distance

from C to any string in W is as small as possible. The *edit radius* of W is the maximum edit distance from an edit center to a string in W. A set of strings may have several edit centers, but its edit radius is unique.

$$EditRadius(W) := \min_{C \in \Sigma^*} \max_{w \in W} Edit(w, C)$$

$$EditCenter(W) := \arg\min_{C \in \Sigma^*} \max_{w \in W} Edit(w, C)$$

(a) Describe and analyze an efficient algorithm to compute the edit radius of three given strings.

♣♥(b) Describe and analyze an efficient algorithm to approximate the edit radius of an arbitrary set of strings within a factor of 2. (Computing the *exact* edit radius is NP-hard unless the number of strings is fixed.)

♥30. Let $D[1..n]$ be an array of digits, each an integer between 0 and 9. A ***digital subsequence*** of D is a sequence of positive integers composed in the usual way from disjoint substrings of D. For example, the sequence $3, 4, 5, 6, 8, 9, 32, 38, 46, 64, 83, 279$ is a digital subsequence of the first several digits of π:

$$\underbrace{3}, 1, \underbrace{4}, 1, \underbrace{5}, 9, 2, \underbrace{6}, 5, 3, 5, \underbrace{8}, \underbrace{9}, 7, 9, \underbrace{3, 2}, \underbrace{3, 8}, \underbrace{4, 6}, 2, \underbrace{6, 4}, 3, 3, \underbrace{8, 3}, \underbrace{2, 7, 9}$$

The *length* of a digital subsequence is the number of integers it contains, *not* the number of digits; the preceding example has length 12. As usual, a digital subsequence is ***increasing*** if each number is larger than its predecessor.

Describe and analyze an efficient algorithm to compute the longest increasing digital subsequence of D. *[Hint: Be careful about your computational assumptions. How long does it take to compare two k-digit numbers?]*

For full credit, your algorithm should run in $O(n^4)$ time; faster algorithms are worth extra credit. The fastest algorithm I know for this problem runs in $O(n^{3/2} \log n)$ time; achieving this bound requires several tricks, both in the design of the algorithm and in its analysis, but nothing outside the scope of this class.[22]

♥31. Consider the following variant of the classical Tower of Hanoi problem. As usual, there are n disks with distinct sizes, placed on three pegs numbered 0, 1, and 2. Initially, all n disks are on peg 0, sorted by size from smallest on top to largest on bottom. Our goal is to move all the disks to peg 2. In a single step, we can move the highest disk on any peg to a different peg,

[22] With more advanced techniques, I believe the running time can be reduced to $O(n^{3/2} \log \log n)$, but I haven't worked through the details.

provided we satisfy two constraints. First, we must never place a smaller disk on top of a larger disk. Second—and this is the non-standard part—***we must never move a disk directly from peg* 0 *to peg* 2.**

Describe and analyze an algorithm to compute the exact number of moves required to move all n disks from peg 0 to peg 2, subject to the stated restrictions. For full credit, your algorithm should use only $O(\log n)$ arithmetic operations in the worst case. For the sake of analysis, assume that adding or multiplying two k-digit numbers requires $O(k)$ time. *[Hint: Matrices!]*

Splitting Sequences/Arrays

32. A ***basic arithmetic expression*** is composed of characters from the set $\{1, +, \times\}$ and parentheses. Almost every integer can be represented by more than one basic arithmetic expression. For example, all of the following basic arithmetic expression represent the integer 14:

$$1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1$$
$$((1+1)\times(1+1+1+1+1))+((1+1)\times(1+1))$$
$$(1+1)\times(1+1+1+1+1+1+1)$$
$$(1+1)\times(((1+1+1)\times(1+1))+1)$$

Describe and analyze an algorithm to compute, given an integer n as input, the minimum number of 1s in a basic arithmetic expression whose value is equal to n. The number of parentheses doesn't matter, just the number of 1s. For example, when $n = 14$, your algorithm should return 8, for the final expression above. The running time of your algorithm should be bounded by a small polynomial function of n.

33. Suppose you are given a sequence of integers separated by $+$ and $-$ signs; for example:

$$1+3-2-5+1-6+7$$

You can change the value of this expression by adding parentheses in different places. For example:

$$1+3-2-5+1-6+7=-1$$
$$(1+3-(2-5))+(1-6)+7=9$$
$$(1+(3-2))-(5+1)-(6+7)=-17$$

Describe and analyze an algorithm to compute, given a list of integers separated by $+$ and $-$ signs, the maximum possible value the expression

can take by adding parentheses. Parentheses must be used only to group additions and subtractions; in particular, do not use them to create implicit multiplication as in $1+3(-2)(-5)+1-6+7=33$.

34. Suppose you are given a sequence of integers separated by $+$ and $\times$ signs; for example:

$$1+3\times 2\times 0+1\times 6+7$$

You can change the value of this expression by adding parentheses in different places. For example:

$$(1+(3\times 2))\times 0+(1\times 6)+7=13$$
$$((1+(3\times 2\times 0)+1)\times 6)+7=19$$
$$(1+3)\times 2\times (0+1)\times (6+7)=104$$

(a) Describe and analyze an algorithm to compute the maximum possible value the given expression can take by adding parentheses, assuming all integers in the input are *positive*. *[Hint: This is easy.]*

(b) Describe and analyze an algorithm to compute the maximum possible value the given expression can take by adding parentheses, assuming all integers in the input are *non-negative*.

(c) Describe and analyze an algorithm to compute the maximum possible value the given expression can take by adding parentheses, with no restrictions on the input numbers.

Assume any arithmetic operation takes $O(1)$ time.

35. After graduating from Sham-Poobanana University, you decide to interview for a position at the Wall Street bank ***Long Live Boole***. The managing director of the bank, Eloob Egroeg, poses a 'solve-or-die' problems to each new employee, which they must solve within 24 hours. Those who fail to solve the problem are fired immediately!

Entering the bank for the first time, you notice that the employee offices are organized in a straight row, with a large T or F printed on the door of each office. Furthermore, between each adjacent pair of offices, there is a board marked by one of the symbols $\wedge, \vee$, or $\oplus$. When you ask about these arcane symbols, Eloob confirms that T and F represent the boolean values True and False, and the symbols on the boards represent the standard boolean operators And, Or, and Xor. He also explains that these letters and symbols describe whether certain combinations of employees can work together successfully. At the start of any new project, Eloob hierarchically clusters his employees by adding parentheses to the sequence of symbols, to

obtain an unambiguous boolean expression. The project is successful if this parenthesized boolean expression evaluates to T.

For example, if the bank has three employees, and the sequence of symbols on and between their doors is $T \wedge F \oplus T$, there is exactly one successful parenthesization scheme: $(T \wedge (F \oplus T))$. However, if the list of door symbols is $F \wedge T \oplus F$, there is no way to add parentheses to make the project successful.

Eloob finally poses your solve-or-die interview question: Describe an algorithm to decide whether a given sequence of symbols can be parenthesized so that the resulting boolean expression evaluates to T. Your input is an array $S[0..2n]$, where $S[i] \in \{T, F\}$ when i is even, and $S[i] \in \{\vee, \wedge, \oplus\}$ when i is odd.

36. Every year, as part of its annual meeting, the Antarctican Snail Lovers of Upper Glacierville hold a Round Table Mating Race. Several high-quality breeding snails are placed at the edge of a round table. The snails are numbered in order around the table from 1 to n. During the race, each snail wanders around the table, leaving a trail of slime behind it. The snails have been specially trained never to fall off the edge of the table or to cross a slime trail, even their own. If two snails meet, they are declared a breeding pair, removed from the table, and whisked away to a romantic hole in the ground to make little baby snails. Note that some snails may never find a mate, even if the race goes on forever.

Figure 3.6. The end of a typical Antarctican SLUG race. Snails 6 and 8 never find mates. The organizers must pay $M[3,4] + M[2,5] + M[1,7]$.

For every pair of snails, the Antarctican SLUG race organizers have posted a monetary reward, to be paid to the owners if that pair of snails meets during the Mating Race. Specifically, there is a two-dimensional

array $M[1..n, 1..n]$ posted on the wall behind the Round Table, where $M[i, j] = M[j, i]$ is the reward to be paid if snails i and j meet.

Describe and analyze an algorithm to compute the maximum total reward that the organizers could be forced to pay, given the array M as input.

37. You have mined a large slab of marble from a quarry. For simplicity, suppose the marble slab is a rectangle measuring n inches in height and m inches in width. You want to cut the slab into smaller rectangles of various sizes—some for kitchen counter tops, some for large sculpture projects, others for memorial headstones. You have a marble saw that can make either horizontal or vertical cuts across any rectangular slab. At any time, you can query the spot price $P[x, y]$ of an x-inch by y-inch marble rectangle, for any positive integers x and y. These prices depend on customer demand, and people who buy marble counter tops are weird, so don't make any assumptions about them; in particular, larger rectangles may have significantly smaller spot prices. Given the array of spot prices and the integers m and n as input, describe an algorithm to compute how to subdivide an $n \times m$ marble slab to maximize your profit.

38. This problem asks you to design efficient algorithms to construct optimal binary search trees that satisfy additional balance constraints. Your input consists of a sorted array $A[1..n]$ of search keys and an array $f[1..n]$ of frequency counts, where $f[i]$ is the number of searches for $A[i]$. This is exactly the same cost function as described in Section 3.9. But now your task is to compute an optimal tree that satisfies some additional constraints.

    (a) ***AVL trees*** were the earliest self-balancing balanced binary search trees, first described in 1962 by Georgy Adelson-Velsky and Evgenii Landis. An AVL tree is a binary search tree where for every node v, the height of the left subtree of v and the height of the right subtree of v differ by at most one.

    Describe and analyze an algorithm to construct an optimal AVL tree for a given set of search keys and frequencies.

    (b) ***Symmetric binary B-trees*** are another self-balancing binary trees, first described by Rudolf Bayer in 1972; these are better known by the name ***red-black trees***, after a somewhat simpler reformulation by Leo Guibas and Bob Sedgwick in 1978. A red-black tree is a binary search tree with the following additional constraints:

    - Every node is either red or black.
    - Every red node has a black parent.
    - Every root-to-leaf path contains the same number of black nodes.

Describe a recursive backtracking algorithm to construct an optimal red-black tree for a given set of search keys and frequencies.

(c) ***AA trees*** were proposed by proposed by Arne Andersson in 1993 and slightly simplified (and named) by Mark Allen Weiss in 2000. AA trees are also known as *left-leaning red-black trees*, after a symmetric reformulation (with different rebalancing algorithms) by Bob Sedgewick in 2006. An AA tree is a red-black tree with one additional constraint:

- No left child is red.[23]

Describe and analyze an algorithm to construct an optimal AA tree for a given set of search keys and frequencies.

39. Suppose you are given an $m \times n$ bitmap as an array $M[1..m, 1..n]$ of 0s and 1s. A *solid block* in M is a subarray of the form $M[i..i', j..i']$ in which all bits are equal. Suppose you want to decompose M into as few disjoint blocks as possible.

One natural recursive partitioning strategy is called a *guillotine subdivision*. If the entire bitmap M is a solid block, there is nothing to do. Otherwise, we cut M into two smaller bitmaps along a horizontal or vertical line, and then recursively decompose the two smaller bitmaps into solid blocks.

Any guillotine subdivision can be represented as a binary tree, where each internal node stores the position and orientation of a cut, and each leaf stores a single but 0 or 1 indicting the contents of the corresponding block. The *size* of a guillotine subdivision is the number of leaves in the corresponding binary tree (that is, the final number of solid blocks), and the *depth* of a guillotine subdivision is the depth of the corresponding binary tree.

(a) Describe and analyze an algorithm to compute a guillotine subdivision of M of minimum possible size.

(b) Show that a guillotine subdivision does *not* always yield a partition into the smallest number of solid blocks.

(c) Describe and analyze an algorithm to compute a guillotine subdivision for M with the smallest possible depth.

(d) Describe and analyze an algorithm to determine $M[i, j]$, given the tree representing a guillotine decomposition for M and two indices i and j.

[23] Sedgwick's reformulation requires that no *right* child is red. Whatever. Andersson and Sedgewick are strangely silent about whether to measure angles clockwise or counterclockwise, whether Pluto is a planet, whether "lower rank" means "better" or "worse", and whether it's better to fight a hundred duck-sized horses or a single horse-sized duck.

Figure 3.7. A guillotine subdivision with size 8 and depth 5.

(e) Define the *depth* of a pixel $M[i, j]$ in a guillotine subdivision to be the depth of the leaf that contains that pixel. Describe and analyze an algorithm to compute a guillotine subdivision for M such that the sum of the depths of the pixels as small as possible.

(f) Describe and analyze an algorithm to compute a guillotine subdivision for M such that the sum of the depths of the *black* pixels as small as possible.

♠40. Congratulations! You've been hired by the giant online bookstore DeNile ("Not just a river in Egypt!") to optimize their warehouse robots. Each book that DeNile sells has a unique ISBN (International Standard Book Number), which is just a numerical value. Each of DeNile's warehouses contains a long row of bins, each containing multiple copies of a single book. These bins are arranged in sorted order by ISBN; each bin's ISBN is printed on the front of the bin in machine-readable form. Books are retrieved from these bins by robots, which run along rails parallel to the row of bins.

DeNile does not maintain a list of which bins contain which ISBN numbers; that would be too simple! Instead, to retrieve a desired book, the robot must first find that book's bin using a binary search. Because the search requires physical motion by the robot, we can no longer assume that each step of the binary search requires $O(1)$ time. Specifically:

- The robot always starts at the "0th bin" (where the books are loaded into boxes to ship to customers).

- Moving the robot from the ith bin to the jth bin requires $\alpha|i-j|$ seconds for some constant α.
- The robot must be directly in front of a bin in order to read that bin's ISBN. Reading an ISBN requires β seconds, for some constant β.
- Reversing the robot's direction of motion (from increasing to decreasing or vice versa) requires γ additional seconds, for some constant γ.
- When the robot finds the target bin, it extracts one book from that bin and returns to "the 0th bin".

Design and analyze an algorithm to compute a binary search tree over the bins that minimizes the total time the robot spends searching for books. Your input is an array $f[1..n]$ of integers, where $f[i]$ is the number of times that the robot will be asked to retrieve a book from the ith bin, along with the time parameters α, β, and γ.

♠41. A standard method to improve the cache performance of search trees is to pack more search keys and subtrees into each node. A ***B-tree*** is a rooted tree in which each internal node stores up to B keys and pointers to up to $B+1$ children, each the root of a smaller B-tree. Specifically, each node v stores three fields:

- a positive integer $v.d \le B$,
- a *sorted* array $v.key[1..v.d]$, and
- an array $v.child[0..v.d]$ of child pointers.

In particular, the number of child pointers is always exactly one more than the number of keys.[24]

Each pointer $v.child[i]$ is either NULL or a pointer to the root of a B-tree whose keys are all larger than $v.key[i]$ and smaller than $v.key[i+1]$. In particular, all keys in the leftmost subtree $v.child[0]$ are smaller than $v.key[1]$, and all keys in the rightmost subtree $v.child[v.d]$ are larger than $v.key[v.d]$.

Intuitively, you should have the following picture in mind:

[24]Normally, B-trees are required to satisfy two additional constraints, which guarantee a worst-case search cost of $O(\log_B n)$: Every leaf must have exactly the same depth, and every node except possibly the root must contain at least $B/2$ keys. However, in this problem, we are not interested in optimizing the *worst-case* search cost, but rather the *total* cost of a sequence of searches, so we will not impose these additional constraints.

Here T_i is the subtree pointed to by $child[i]$.

The ***cost*** of searching for a key x in a B-tree is the number of nodes in the path from the root to the node containing x as one of its keys. A 1-tree is just a standard binary search tree.

Fix an arbitrary positive integer $B > 0$. (I suggest $B = 8$.) Suppose your are given a sorted array $A[1, \ldots, n]$ of search keys and a corresponding array $F[1, \ldots, n]$ of frequency counts, where $F[i]$ is the number of times that we will search for $A[i]$. Your task is to describe and analyze an efficient algorithm to find a B-tree that minimizes the total cost of searching for the given keys with the given frequencies.

(a) Describe a polynomial-time algorithm for the special case $B = 2$.

(b) Describe an algorithm for arbitrary B that runs in $O(n^{B+c})$ time for some fixed integer c.

♥(c) Describe an algorithm for arbitrary B that runs in $O(n^c)$ time for some fixed integer c that does *not* depend on B.

42. A string w of parentheses **(** and **)** and brackets **[** and **]** is ***balanced*** if it satisfies one of the following conditions:

- w is the empty string.
- $w =$ **(**x**)** for some balanced string x
- $w =$ **[**x**]** for some balanced string x
- $w = xy$ for some balanced strings x and y

For example, the string

$$w = \texttt{([()][]())[()()]()}$$

is balanced, because $w = xy$, where

$$x = \texttt{([()] [] ())} \qquad \text{and} \qquad y = \texttt{[() ()] ()}.$$

(a) Describe and analyze an algorithm to determine whether a given string of parentheses and brackets is balanced.

(b) Describe and analyze an algorithm to compute the length of a longest balanced subsequence of a given string of parentheses and brackets.

(c) Describe and analyze an algorithm to compute the length of a shortest balanced supersequence of a given string of parentheses and brackets.

(d) Describe and analyze an algorithm to compute the minimum edit distance from a given string of parentheses and brackets to a balanced string of parentheses and brackets.

♥(e) Describe and analyze an algorithm to compute the longest common balanced subsequence of two given strings of parentheses and brackets.

♥(f) Describe and analyze an algorithm to compute the longest palindromic balanced subsequence of a given string of parentheses and brackets.

♥(g) Describe and analyze an algorithm to compute the longest common palindromic balanced subsequence (whew!) of two given strings of parentheses and brackets.

For each problem, your input is an array $w[1..n]$, where $w[i] \in \{(,), [,]\}$ for every index i. (You may prefer to use different symbols instead of parentheses and brackets—for example, L, R, l, r or ◁, ▷, ◀, ▶—but please tell your grader what symbols you're using!)

♥43. Congratulations! Your research team has just been awarded a $50M multi-year project, jointly funded by DARPA, Google, and McDonald's, to produce DWIM: The first compiler to read programmers' minds! Your proposal and your numerous press releases all promise that DWIM will automatically correct errors in any given piece of code, while modifying that code as little as possible. Unfortunately, now it's time to start actually making the damn thing work.

As a warmup exercise, you decide to tackle the following necessary subproblem. Recall that the *edit distance* between two strings is the minimum number of single-character insertions, deletions, and replacements required to transform one string into the other. An *arithmetic expression* is a string w such that

- w is a string of one or more decimal digits,
- $w = (x)$ for some arithmetic expression x, or
- $w = x \diamond y$ for some arithmetic expressions x and y and some binary operator $\diamond$.

Suppose you are given a string of tokens from the alphabet $\{\#, \diamond, (,)\}$, where # represents a decimal digit and $\diamond$ represents a binary operator. Describe and analyze an algorithm to compute the minimum edit distance from the given string to an arithmetic expression.

44. Ribonucleic acid (RNA) molecules are long chains of millions of nucleotides or *bases* of four different types: adenine (A), cytosine (C), guanine (G), and uracil (U). The *sequence* of an RNA molecule is a string $b[1..n]$, where each character $b[i] \in \{A, C, G, U\}$ corresponds to a base. In addition to the chemical bonds between adjacent bases in the sequence, hydrogen bonds can form between certain pairs of bases. The set of bonded base pairs is called the *secondary structure* of the RNA molecule.

We say that two base pairs (i, j) and (i', j') with $i < j$ and $i' < j'$ ***overlap*** if $i < i' < j < j'$ or $i' < i < j' < j$. In practice, most base pairs are non-overlapping. Overlapping base pairs create so-called *pseudoknots* in the secondary structure, which are essential for some RNA functions, but are more difficult to predict.

Suppose we want to predict the best possible secondary structure for a given RNA sequence. We will adopt a drastically simplified model of secondary structure:

- Each base can bond with at most one other base.
- Only A–U pairs and C–G pairs can bond.
- Pairs of the form $(i, i+1)$ and $(i, i+2)$ cannot bond.
- Bonded base pairs cannot overlap.

The last (and least realistic) restriction allows us to visualize RNA secondary structure as a sort of fat tree, as shown below.

Figure 3.8. Example RNA secondary structure with 21 bonded base pairs, indicated by heavy red lines. Gaps are indicated by dotted curves. This structure has score $2^2 + 2^2 + 8^2 + 1^2 + 7^2 + 4^2 + 7^2 = 187$.

(a) Describe and analyze an algorithm that computes the maximum possible *number* of bonded base pairs in a secondary structure for a given RNA sequence.

(b) A *gap* in a secondary structure is a maximal substring of unpaired bases. Large gaps lead to chemical instabilities, so secondary structures with smaller gaps are more likely. To account for this preference, let's define the *score* of a secondary structure to be the sum of the *squares* of the gap lengths; see Figure 3.8. (This score function is utterly fictional; real RNA structure prediction requires *much* more complicated scoring functions.)

Describe and analyze an algorithm that computes the minimum possible score of a secondary structure for a given RNA sequence.

♣45. (a) Describe and analyze an efficient algorithm to determine, given a string w and a regular expression R, whether $w \in L(R)$.

(b) *Generalized* regular expressions allow the binary operator $\cap$ (intersection) and the unary operator $\neg$ (complement), in addition to the usual $\bullet$ (concatenation), $+$ (or), and * (Kleene closure) operators. NFA constructions and Kleene's theorem imply that any generalized regular expression E represents a regular language $L(E)$.

Describe and analyze an efficient algorithm to determine, given a string w and a generalized regular expression E, whether $w \in L(E)$.

In both problems, assume that you are actually given a parse tree for the (generalized) regular expression, not just a string.

Trees and Subtrees

46. You've just been appointed as the new organizer of the first annual mandatory holiday party at Giggle (a subsidiary of Abugida). Giggle employees are organized into a strict hierarchy—a tree with the company president at the root. The all-knowing oracles in Human Resources have assigned a real number to each employee measuring how "fun" the employee is. To keep things social, there is one restriction on the guest list: an employee cannot attend the party if their immediate supervisor is also present. On the other hand, the president of the company *must* attend the party, even though she has a negative fun rating; it's her company, after all. Give an algorithm that makes a guest list for the party that maximizes the sum of the "fun" ratings of the guests.

47. Since so few people came to last year's holiday party, the president of Giggle decides to give each employee a present instead this year. Specifically, each employee must receive on the three gifts: (1) an all-expenses-paid six-week vacation anywhere in the world, (2) an all-the-pancakes-you-can-sort breakfast for two at Jumping Jack Flash's Flapjack Stack Shack, or (3) a burning paper bag full of dog poop. Corporate regulations prohibit any employee from receiving exactly the same gift as his/her direct supervisor. Any employee who receives a better gift than his/her direct supervisor will almost certainly be fired in a fit of jealousy.

As Giggle's official party czar, it's *your* job to decide which gift each employee receives. Describe an algorithm to distribute gifts so that the minimum number of people are fired. Yes, you may send the president a flaming bag of dog poop.

More formally, you are given a rooted tree T, representing the company hierarchy, and you want to label the nodes of T with integers 1, 2, or 3, so

that every node has a different label from its parent. The *cost* of an labeling is the number of nodes with smaller labels than their parents. See Figure 3.9 for an example. Describe and analyze an algorithm to compute the minimum-cost labeling of T.

Figure 3.9. A tree labeling with cost 9. The nine bold nodes have smaller labels than their parents. This is *not* the optimal labeling for this tree.

48. After the Flaming Dog Poop Holiday Debacle, you were strongly encouraged to seek other employment, and so you left Giggle for rival company Twitbook. Unfortunately, the new president of Twitbook just decided to imitate Giggle by throwing her own holiday party, and in light of your past experience, appointed you as the official party organizer. The president demands that you invite exactly k employees, including the president herself, and everyone who is invited is required to attend. Yeah, that'll be fun.

    Just like at Giggle, employees at Twitbook are organized into a strict hierarchy: a tree with the company president at the root. The all-knowing oracles in Human Resources have assigned a real number to each employee indicating the *awkwardness* of inviting both that employee and their immediate supervisor; a negative value indicates that the employee and their supervisor actually like each other. Your goal is to choose a subset of exactly k employees to invite, so that the total awkwardness of the resulting party is as small as possible. For example, if the guest list does not include both an employee and their immediate supervisor, the total awkwardness is zero. The input to your algorithm is the tree T, the integer k, and the awkwardness of each node in T.

    (a) Describe an algorithm that computes the total awkwardness of the least awkward subset of k employees, assuming the company hierarchy is described by a *binary* tree. That is, assume that each employee directly supervises at most two others.

    ♥(b) Describe an algorithm that computes the total awkwardness of the least awkward subset of k employees, with no restrictions on the company hierarchy.

49. Suppose we need to broadcast a message to all the nodes in a rooted tree. Initially, only the root node knows the message. In a single round, any node that knows the message can forward it to at most one of its children. See Figure 3.10 for an example.

    (a) Design an algorithm to compute the minimum number of rounds required to broadcast the message to all nodes in a *binary* tree.

    ♦(b) Design an algorithm to compute the minimum number of rounds required to broadcast the message to all nodes in an *arbitrary* rooted tree. *[Hint: You may find techniques in the next chapter useful to prove your algorithm is correct, even though it's not a greedy algorithm.]*

Figure 3.10. A message being distributed through a tree in five rounds.

50. One day, Alex got tired of climbing in a gym and decided to take a very large group of climber friends outside to climb. The climbing area where they went, had a huge wide boulder, not very tall, with various marked hand and foot holds. Alex quickly determined an "allowed" set of moves that her group of friends can perform to get from one hold to another.

    The overall system of holds can be described by a rooted tree T with n vertices, where each vertex corresponds to a hold and each edge corresponds to an allowed move between holds. The climbing paths converge as they go up the boulder, leading to a unique hold at the summit, represented by the root of T.[25]

    Alex and her friends (who are all excellent climbers) decided to play a game, where as many climbers as possible are simultaneously on the boulder and each climber needs to perform a sequence of *exactly* k moves. Each climber can choose an arbitrary hold to start from, and all moves must move away from the ground. Thus, each climber traces out a path of k edges in the tree T, all directed toward the root. However, no two climbers are allowed to touch the same hold; the paths followed by different climbers cannot intersect at all.

    Describe and analyze an efficient algorithm to compute the maximum number of climbers that can play this game. More formally, you are given a rooted tree T and an integer k, and you want to find the largest possible

[25] Q: Why do computer science professors think trees have their roots at the top?
A: Because they've never been outside!

number of disjoint paths in T, where each path has length k. Do ***not*** assume that T is a binary tree. For example, given the tree T below and $k = 3$ as input, your algorithm should return the integer 8.

Figure 3.11. Seven disjoint paths of length $k = 3$. This is *not* the largest such set of paths in this tree.

51. Let T be a rooted binary tree with n vertices, and let $k \leq n$ be a positive integer. We would like to mark k vertices in T so that every vertex has a nearby marked ancestor. More formally, we define the *clustering cost* of any subset K of vertices as

$$cost(K) = \max_v cost(v, K),$$

where the maximum is taken over all vertices v in the tree, and $cost(v, K)$ is the distance from v to its nearest ancestor in K:

$$cost(v, K) = \begin{cases} 0 & \text{if } v \in K \\ \infty & \text{if } v \text{ is the root of } T \text{ and } v \notin K \\ 1 + cost(parent(v)) & \text{otherwise} \end{cases}$$

In particular, $cost(K) = \infty$ if K excludes the root of T.

Figure 3.12. A subset of five vertices in a binary tree, with clustering cost 3.

♥(a) Describe a dynamic programming algorithm to compute, given the tree T and an integer k, the minimum clustering cost of any subset of k vertices in T. For full credit, your algorithm should run in $O(n^2k^2)$ time.

(b) Describe a dynamic programming algorithm to compute, given the tree T and an integer r, the size of the smallest subset of vertices whose clustering cost is at most r. For full credit, your algorithm should run in $O(nr)$ time.

(c) Show that your solution for part (b) implies an algorithm for part (a) that runs in $O(n^2 \log n)$ time.

52. This question asks you to find efficient algorithms to compute the ***largest common rooted subtree*** of two given rooted trees. Recall that a *rooted tree* is a connected acyclic graph with a designated node called the root. A rooted subtree of a rooted tree consists of an arbitrary node and all its descendants. The precise definition of "common" depends on which pairs of rooted trees we consider isomorphic.

(a) Recall that a *binary* tree is a rooted tree in which every node has a (possibly empty) *left* subtree and a (possibly empty) *right* subtree. Two binary trees are isomorphic if and only if they are both empty, or their left subtrees are isomorphic and their right subtrees are isomorphic. Describe an algorithm to find the largest common *binary* subtree of two given *binary* trees.

Figure 3.13. Two binary trees, with their largest common (rooted) subtree emphasized.

(b) In an *ordered* rooted tree, each node has a *sequence* of children, which are the roots of ordered rooted subtrees. Two ordered rooted trees are isomorphic if they are both empty, or if their ith subtrees are isomorphic for every index i. Describe an algorithm to find the largest common ordered subtree of two ordered trees T_1 and T_2.

♦♥(c) In an *unordered* rooted tree, each node has an unordered *set* of children, which are the roots of unordered rooted subtrees. Two unordered rooted trees are isomorphic if they are both empty, or the subtrees of each root *can be ordered so that* their ith subtrees are isomorphic for every index i. Describe an algorithm to find the largest common unordered subtree of two unordered trees T_1 and T_2.

53. This question asks you to find efficient algorithms to compute optimal subtrees in *unrooted* trees—connected acyclic undirected graphs. A *subtree* of an unrooted tree is any connected subgraph.

    (a) Suppose you are given an unrooted tree T with weights on its *edges*, which may be positive, negative, or zero. Describe an algorithm to find a *path* in T with maximum total weight.

    (b) Suppose you are given an unrooted tree T with weights on its *vertices*, which may be positive, negative, or zero. Describe an algorithm to find a *subtree* of T with maximum total weight. *[This was a 2016 Google interview question.]*

    (c) Let T_1 and T_2 be arbitrary *ordered* unrooted trees, meaning that the neighbors of every node have a well-defined cyclic order. Describe an algorithm to find the largest common *ordered* subtree of T_1 and T_2.

    ♦♥(d) Let T_1 and T_2 be arbitrary *unordered* unrooted trees. Describe an algorithm to find the largest common *unordered* subtree of T_1 and T_2.

54. ***Rooted minors*** of rooted trees are a natural generalization of subsequences. A rooted minor of a rooted tree T is any tree obtained by *contracting* one or more edges. When we contract an edge $u \to v$, where u is the parent of v, the children of v become new children of u and then v is deleted. In particular, the root of T is also the root of every rooted minor of T.

Figure 3.14. A rooted tree and one of its rooted minors.

    (a) Let T be a rooted tree with labeled nodes. We say that T is *boring* if, for each node x, all children of x have the same label; children of different nodes may have different labels. Describe an algorithm to find the largest boring rooted minor of a given labeled rooted tree.

    (b) Suppose we are given a rooted tree T whose nodes are labeled with numbers. Describe an algorithm to find the largest *heap-ordered rooted minor* of T. That is, your algorithm should return the largest rooted minor M such that every node in M has a smaller label than its children in M.

(c) Suppose we are given a *binary* tree T whose nodes are labeled with numbers. Describe an algorithm to find the largest *binary-search-ordered rooted minor* of T. That is, your algorithm should return a rooted minor M such that every node in M has at most two children, and an inorder traversal of M is an increasing subsequence of an inorder traversal of T.

(d) Recall that a rooted tree is *ordered* if the children of each node have a well-defined left-to-right order. Describe an algorithm to find the largest binary-search-ordered minor of an *arbitrary* ordered tree T whose nodes are labeled with numbers. Again, the left-to-right order of nodes in M should be consistent with their order in T.

♥(e) Describe an algorithm to find the largest common *ordered* rooted minor of two *ordered* labeled rooted trees.

♦♥(f) Describe an algorithm to find the largest common *unordered* rooted minor of two *unordered* labeled rooted trees. *[Hint: Combine dynamic programming with maximum flows.]*

The point is, ladies and gentleman, greed is good. Greed works, greed is right. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit. Greed in all its forms, greed for life, money, love, knowledge has marked the upward surge in mankind. And greed—mark my words—will save not only Teldar Paper but the other malfunctioning corporation called the USA.

— Gordon Gekko [Michael Douglas], *Wall Street* (1987)

There is always an easy solution to every human problem—neat, plausible, and wrong.

— H. L. Mencken, "The Divine Afflatus", *New York Evening Mail* (November 16, 1917)

4

Greedy Algorithms

4.1 Storing Files on Tape

Suppose we have a set of n files that we want to store on magnetic tape.[1] In the future, users will want to read those files from the tape. Reading a file from tape isn't like reading a file from disk; first we have to fast-forward past all the other files, and that takes a significant amount of time. Let $L[1..n]$ be an array listing the lengths of each file; specifically, file i has length $L[i]$. If the files are stored in order from 1 to n, then the cost of accessing the kth file is

$$cost(k) = \sum_{i=1}^{k} L[i].$$

[1]Readers who are tempted to object that magnetic tape has been obsolete for decades are cordially invited to tour your nearest supercomputer facility; ask to see the tape robots. Alternatively, consider filing a sequence of books on a library bookshelf. You know, those strange brick-like objects made of dead trees and ink?

The cost reflects the fact that before we read file k we must first scan past all the earlier files on the tape. If we assume for the moment that each file is equally likely to be accessed, then the *expected* cost of searching for a random file is

$$\mathrm{E}[cost] = \sum_{k=1}^{n} \frac{cost(k)}{n} = \frac{1}{n} \sum_{k=1}^{n} \sum_{i=1}^{k} L[i].$$

If we change the order of the files on the tape, we change the cost of accessing the files; some files become more expensive to read, but others become cheaper. Different file orders are likely to result in different expected costs. Specifically, let $\pi(i)$ denote the index of the file stored at position i on the tape. Then the expected cost of the permutation π is

$$\mathrm{E}[cost(\pi)] = \frac{1}{n} \sum_{k=1}^{n} \sum_{i=1}^{k} L[\pi(i)].$$

Which order should we use if we want this expected cost to be as small as possible? The answer *seems* intuitively clear: Sort the files by increasing length. But intuition is a tricky beast. The only way to be *sure* that this order works is to ~~take off and nuke the entire site from orbit~~ actually *prove* that it works!

Lemma 4.1. $\mathrm{E}[cost(\pi)]$ *is minimized when* $L[\pi(i)] \leq L[\pi(i+1)]$ *for all* i.

Proof: Suppose $L[\pi(i)] > L[\pi(i+1)]$ for some index i. To simplify notation, let $a = \pi(i)$ and $b = \pi(i+1)$. If we swap files a and b, then the cost of accessing a increases by $L[b]$, and the cost of accessing b decreases by $L[a]$. Overall, the swap changes the expected cost by $(L[b] - L[a])/n$. But this change is an improvement, because $L[b] < L[a]$. Thus, if the files are out of order, we can decrease the expected cost by swapping some mis-ordered pair of files. □

This is our first example of a correct *greedy algorithm*. To minimize the *total* expected cost of accessing the files, we put the file that is cheapest to access first, and then recursively write everything else; no backtracking, no dynamic programming, just make the best local choice and blindly plow ahead. If we use an efficient sorting algorithm, the running time is clearly $O(n \log n)$, plus the time required to actually write the files. To show that the greedy algorithm is actually correct, we proved that the output of any other algorithm can be improved by some sort of exchange

Let's generalize this idea further. Suppose we are also given an array $F[1..n]$ of *access frequencies* for each file; file i will be accessed exactly $F[i]$ times over the lifetime of the tape. Now the *total* cost of accessing all the files on the tape is

$$\Sigma cost(\pi) = \sum_{k=1}^{n} \left(F[\pi(k)] \cdot \sum_{i=1}^{k} L[\pi(i)] \right) = \sum_{k=1}^{n} \sum_{i=1}^{k} \big(F[\pi(k)] \cdot L[\pi(i)] \big).$$

As before, reordering the files can change this total cost. So what order should we use if we want the total cost to be as small as possible? (This question is similar in spirit to the optimal binary search tree problem, but the target data structure and the cost function are both different, so the algorithm must be different, too.)

We already proved that if all the frequencies are equal, we should sort the files by increasing size. If the frequencies are all different but the file lengths $L[i]$ are all equal, then intuitively, we should sort the files by *decreasing* access frequency, with the most-accessed file first. In fact, this is not hard to prove (hint, hint) by modifying the proof of Lemma 4.1. But what if the sizes and the frequencies both vary? In this case, we should sort the files by the *ratio* L/F.

Lemma 4.2. *$\Sigma cost(\pi)$ is minimized when* $\dfrac{L[\pi(i)]}{F[\pi(i)]} \leq \dfrac{L[\pi(i+1)]}{F[\pi(i+1)]}$ *for all* i.

Proof: Suppose $L[\pi(i)]/F[\pi(i)] > L[\pi(i+1)]/F[\pi(i+i)]$ for some index i. To simplify notation, let $a = \pi(i)$ and $b = \pi(i+1)$. If we swap files a and b, then the cost of accessing a increases by $L[b]$, and the cost of accessing b decreases by $L[a]$. Overall, the swap changes the total cost by $L[b]F[a] - L[a]F[b]$. But this change is an improvement, because

$$\frac{L[a]}{F[a]} > \frac{L[b]}{F[b]} \iff L[b]F[a] - L[a]F[b] < 0.$$

Thus, if any two adjacent files are out of order, we can improve the total cost by swapping them. □

4.2 Scheduling Classes

The next example is slightly more complex. Suppose you decide to drop out of computer science and change your major to Applied Chaos. The Applied Chaos department offers all of its classes on the same day every week, called "Soberday" by the students (but interestingly, *not* by the faculty). Every class has a different start time and a different ending time: AC 101 ("Toilet Paper Landscape Architecture") starts at 10:27pm and ends at 11:51pm; AC 666 ("Immanentizing the Eschaton") starts at 4:18pm and ends at 4:22pm, and so on. In the interest of graduating as quickly as possible, you want to register for as many classes as possible. (Applied Chaos classes don't require any actual *work*.) The university's registration computer won't let you register for overlapping classes, and no one in the department knows how to override this "feature". Which classes should you take?

More formally, suppose you are given two arrays $S[1..n]$ and $F[1..n]$ listing the start and finish times of each class; to be concrete, we can assume that

$0 \le S[i] < F[i] \le M$ for each i, for some value M (for example, the number of picoseconds in Soberday). Your task is to choose the largest possible subset $X \in \{1, 2, \ldots, n\}$ so that for any pair $i, j \in X$, either $S[i] > F[j]$ or $S[j] > F[i]$. We can illustrate the problem by drawing each class as a rectangle whose left and right x-coordinates show the start and finish times. The goal is to find a largest subset of rectangles that do not overlap vertically.

Figure 4.1. A maximum conflict-free schedule for a set of classes.

This problem has a fairly simple recursive solution, based on the observation that either you take class 1 or you don't. Let B denote the set of classes that *end before* class 1 starts, and let A denote the set of classes that *start after* class 1 ends:

$$B := \{i \mid 2 \le i \le n \text{ and } F[i] < S[1]\}$$
$$A := \{i \mid 2 \le i \le n \text{ and } S[i] > F[1]\}$$

If class 1 is in the optimal schedule, then so are the optimal schedules for B and A, which we can find recursively. If not, we can find the optimal schedule for $\{2, 3, \ldots, n\}$ recursively. So we should try both choices and take whichever one gives the better schedule. Evaluating this recursive algorithm from the bottom up gives us a dynamic programming algorithm that runs in $O(n^3)$ time. I won't bother to go through the details, because we can do better.[2]

Intuitively, we'd like the first class to finish as early as possible, because that leaves us with the largest number of remaining classes. This intuition suggests the following simple greedy algorithm. Scan through the classes in order of finish time; whenever you encounter a class that doesn't conflict with your latest class so far, take it! See Figure 4.2 for a visualization of the resulting greedy schedule.

We can write the greedy algorithm somewhat more formally as shown in in Figure 4.3. (Hopefully the first line is understandable.) After the initial sort, the algorithm is a simple linear-time loop, so the entire algorithm runs in $\boldsymbol{O(n \log n)}$ ***time***.

[2]But you should still work out the details yourself. The dynamic programming algorithm can be used to find the "best" schedule for several different definitions of "best", but the greedy algorithm I'm describing here only works when "best" means "biggest". Also, you can improve the running time to $O(n^2)$ using a different recurrence.

Figure 4.2. The same classes sorted by finish times and the greedy schedule.

```
GreedySchedule(S[1..n], F[1..n]):
  sort F and permute S to match
  count ← 1
  X[count] ← 1
  for i ← 2 to n
      if S[i] > F[X[count]]
          count ← count + 1
          X[count] ← i
  return X[1..count]
```

Figure 4.3. A greedy algorithm for finding a maximum set of non-overlapping classes

To prove that GreedySchedule actually computes the largest conflict-free schedule, we use an exchange argument, similar to the one we used for tape sorting. We are not claiming that the greedy schedule is the *only* maximal schedule; there could be others. (Compare Figures 4.1 and 4.2!) All we can claim is that at least one of the optimal schedules is the one produced by the greedy algorithm.

Lemma 4.3. *At least one maximal conflict-free schedule includes the class that finishes first.*

Proof: Let f be the class that finishes first. Suppose we have a maximal conflict-free schedule X that does not include f. Let g be the first class in X to finish. Since f finishes before g does, f cannot conflict with any class in the set $X \setminus \{g\}$. Thus, the schedule $X' = X \cup \{f\} \setminus \{g\}$ is also conflict-free. Since X' has the same size as X, it is also maximal. □

To finish the proof, we call on our old friend induction.

Theorem 4.4. *The greedy schedule is an optimal schedule.*

Proof: Let f be the class that finishes first, and let A be the subset of classes that start after f finishes. The previous lemma implies that some optimal schedule contains f, so the best schedule that contains f is an optimal schedule. The best schedule that includes f must contain an optimal schedule for the classes that do not conflict with f, that is, an optimal schedule for A. The greedy algorithm chooses f and then, by the inductive hypothesis, computes an optimal schedule of classes from A. □

The proof might be easier to understand if we unroll the induction slightly.

Proof: Let $\langle g_1, g_2, \ldots, g_k \rangle$ be the sequence of classes chosen by the greedy algorithm, sorted by starting time. Suppose we have a maximal conflict-free schedule

$$S = \langle g_1, g_2, \ldots, g_{j-1}, \boldsymbol{c_j}, c_{j+1}, \ldots, c_m \rangle,$$

again sorted by starting time, where c_j is different from the class g_j chosen by the greedy algorithm. (We could have $j = 1$, in which case this schedule starts with a non-greedy choice c_1.) By construction, the jth greedy choice g_j does not conflict with any earlier class $g_1, g_2, \ldots, g_{j-1}$, and because our schedule S is conflict-free, neither does c_j. Moreover, g_j has the *earliest* finish time among all classes that don't conflict with the earlier classes; in particular, g_j finishes before c_j. It follows that g_j does not conflict with any of the later classes $c_{j+1}, \ldots, c_m$. Thus, the modified schedule

$$S' = \langle g_1, g_2, \ldots, g_{j-1}, \boldsymbol{g_j}, c_{j+1}, \ldots, c_m \rangle,$$

is also conflict-free. (This argument is a direct generalization of Lemma 4.3, which considers the case $j = 1$.)

By induction, it now follows that there is an optimal schedule $\langle g_1, g_2, \ldots, g_k, c_{k+1}, \ldots, c_m \rangle$ that includes *every* class chosen by the greedy algorithm. But this is impossible unless $k = m$; if some class c_{k+1} does not conflict with any of the first k greedy classes, then the greedy algorithm would choose more than k classes! □

4.3 General Pattern

The basic structure of this correctness proof is exactly the same as for the tape-sorting problem: an inductive exchange argument.

- Assume that there is an optimal solution that is different from the greedy solution.
- Find the "first" difference between the two solutions.

- Argue that we can exchange the optimal choice for the greedy choice without making the solution worse (although the exchange might not make it better).

This argument implies by induction that some optimal solution *contains* the entire greedy solution, and therefore *equals* the greedy solution. Sometimes, as in the scheduling problem, an additional step is required to show no optimal solution *strictly* improves the greedy solution.

4.4 Huffman Codes

A *binary code* assigns a string of 0s and 1s to each character in the alphabet. A binary code is *prefix-free* if no code is a prefix of any other. (Confusingly, prefix-free codes are also commonly called *prefix codes*.) 7-bit ASCII and Unicode's UTF-8 are both prefix-free binary codes. Morse code is a binary code with symbols • and —, but it is *not* prefix-free, because the code for `E` (•) is a prefix of the codes for `I` (••), `S` (•••), and `H` (••••).[3]

Any prefix-free binary code can be visualized as a binary tree with the encoded characters stored at the leaves. The code word for any symbol is given by the path from the root to the corresponding leaf; `0` for left, `1` for right. Thus, the length of any symbol's codeword is the depth of the corresponding leaf in the code tree. Although they are superficially similar, binary code trees are *not* binary search trees; we don't care at all about the order of symbols at the leaves.

Suppose we want to encode a message written in an n-character alphabet so that the encoded message is as short as possible. Specifically, given an array of frequency counts $f[1..n]$, we want to compute a prefix-free binary code that minimizes the total encoded length of the message:

$$\sum_{i=1}^{n} f[i] \cdot depth(i).$$

This is exactly the same cost function we considered for optimizing binary search trees, but the optimization problem is different, because code trees are not required to keep the keys in any particular order.

[3]For this reason, Morse code is arguably better described as a prefix-free *ternary* code, with three symbols: •, —, and pause. Alternatively, Morse code can be considered a prefix-free binary code, with one beat of sound/light/current/high voltage/smoke/gas (■) and one beat of silence/darkness/ground/low voltage/air/liquid (□) as the two symbols. Then each "dit" is encoded as ■□, each "dah" as ■■■□, and each pause as □□. In standard Morse code, each letter is followed by one pause, and each word is followed by two additional pauses; however, □s at the end of the entire coded message are omitted. For example, the string "`MORSE CODE`" is unambiguously encoded as the following bit string:

In 1951, as a PhD student at MIT, David Huffman developed the following greedy algorithm to produce such an optimal code:[4]

Huffman: Merge the two least frequent letters and recurse.

Huffman's algorithm is best illustrated through an example. Suppose we want to encode the following helpfully self-descriptive sentence, discovered by Lee Sallows:[5]

> This sentence contains three a's, three c's, two d's, twenty-six e's, five f's, three g's, eight h's, thirteen i's, two l's, sixteen n's, nine o's, six r's, twenty-seven s's, twenty-two t's, two u's, five v's, eight w's, four x's, five y's, and only one z.

To keep things simple, let's ignore the forty-four spaces, nineteen apostrophes, nineteen commas, three hyphens, and only one period, and encode only the letters, as though the message were written in *scriptio continua*:

```
THISSENTENCECONTAINSTHREEASTHREECSTWODSTWENTYSIXESFIVEFST
HREEGSEIGHTHSTHIRTEENISTWOLSSIXTEENNSNINEOSSIXRSTWENTYSEV
ENSSTWENTYTWOTSTWOUSFIVEVSEIGHTWSFOURXSFIVEYSANDONLYONEZ
```
[6]

Here is the frequency table for Sallows' sentence:

A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1

Huffman's algorithm picks out the two least frequent letters, breaking ties arbitrarily—in this case, say, Z and D—and merges them together into a single

[4]Huffman was a student in an information theory class taught by Robert Fano, who was a close colleague of Claude Shannon, the father of information theory. Fano and Shannon had previously developed a different greedy algorithm for producing prefix codes—split the frequency array into two subarrays as evenly as possible, and then recursively build a code for each subarray—but these Fano-Shannon codes were known not to be optimal. Fano posed the problem of finding an optimal prefix code to his class. Huffman decided to solve the problem as a class project, instead of taking a final exam, not realizing that the problem was open, or that Fano and Shannon had already tried and failed to solve it. After several months of fruitless effort, Huffman eventually gave up and decided to take the final exam after all. As he was throwing his notes in the trash, the solution dawned on him. Huffman would later describe the epiphany as "the absolute lightning of sudden realization".

[5]This sentence was first reported by Alexander Dewdney in his October 1984 "Computer Recreations" column in *Scientific American*. Sallows himself published the remarkable story of its discovery in 1985, along with several other self-descriptive sentences; you can find Sallows' paper on his web site. Frustrated with the slow progress of his code running on a VAX 11/780, Sallows designed and built dedicated hardware to perform a brute-force search for self-descriptive sentences with various flavor text ("This pangram has. . . ", "This sentence contains exactly. . . ", and so on). Careful theoretical analysis limited the search space to just over six billion possibilities, which his 1-MHz Pangram Machine enumerated in just under two hours.

[6]. . . and he talked for forty-five minutes, and nobody understood a word that he said, but we had fun fillin' out the forms and playin' with the pencils on the bench there.

new character DZ with frequency 3. This new character becomes an internal node in the code tree we are constructing, with Z and D as its children; it doesn't matter which child is which. The algorithm then recursively constructs a Huffman code for the new frequency table

A	C	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	DZ
3	3	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	3

After 19 merges, all 20 letters have been merged together. The record of merges gives us our code tree. The algorithm makes a number of arbitrary choices; as a result, there are actually several different Huffman codes. One such Huffman code is shown below; numbers in non-leaf nodes are frequencies for merged characters. For example, the code for A is 101000, and the code for S is 111.

Encoding Sallows' sentence with this particular Huffman code would yield a bit string that starts like so:

100	0110	1011	111	111	110	010	100	110	010	101001	110	101001	0001	010	100	...
T	H	I	S	S	E	N	T	E	N	C	E	C	O	N	T	

Here is the list of costs for encoding each character in Sallows' sentence, along with that character's contribution to the total length of the encoded sentence:

char	A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
freq	3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1
depth	6	6	7	3	5	6	4	4	6	3	4	5	3	3	6	5	4	5	5	7
total	18	18	14	78	25	18	32	52	12	48	36	30	81	66	12	25	32	20	25	7

Altogether, the encoded message is 649 bits long. Different Huffman codes encode the same characters differently, possibly with code words of different length, but the overall length of the encoded message is the same for *every* Huffman code: 649 bits.

Given the simple structure of Huffman's algorithm, it's rather surprising that it produces an *optimal* prefix-free binary code.[7] Encoding Sallows' sentence using *any* prefix-free code requires at least 649 bits! Fortunately, the recursive structure makes this claim easy to prove using an exchange argument, similar to our earlier optimality proofs. We start by proving that the algorithm's very first choice is correct.

Lemma 4.5. *Let x and y be the two least frequent characters (breaking ties between equally frequent characters arbitrarily). There is an optimal code tree in which x and y are siblings.*

Proof: I'll actually prove a stronger statement: There is an optimal code in which x and y are siblings *and* have the largest depth of any leaf.

Let T be an optimal code tree, and suppose this tree has depth d. Because T is a full binary tree, it has at least two leaves at depth d that are siblings. (Verify this claim by induction!) Suppose those two leaves are *not* x and y, but some other characters a and b.

Let T' be the code tree obtained by swapping x and a, and let $\Delta = d - depth_T(x)$. This swap increases the depth of x by Δ and decreases the depth of a by Δ, so

$$cost(T') = cost(T) + \Delta \cdot (f[x] - f[a]).$$

Our assumption that x is one of the two least frequent characters but a is not implies $f[x] \leq f[a]$, and our assumption that a has maximum depth implies $\Delta \geq 0$. It follows that $cost(T') \leq cost(T)$. On the other hand, T is an optimal code tree, so we must also have $cost(T') \geq cost(T)$. We conclude that T' is *also* an optimal code tree.

Similarly, swapping y and b must give yet another optimal code tree. In this final optimal code tree, x and y are maximum-depth siblings, as required. □

Now optimality is guaranteed by our dear friend the Recursion Fairy! Our recursive argument relies on the following non-standard recursive definition: *A full binary tree is either a single node, or a full binary tree where some leaf has been replaced by an internal node with two leaf children.*

Theorem 4.6. *Every Huffman code is an optimal prefix-free binary code.*

Proof: If the message has only one or two distinct characters, the theorem is trivial, so assume otherwise.

Let $f[1..n]$ be the original input frequencies, and assume without loss of generality that $f[1]$ and $f[2]$ are the two smallest frequencies. To set up the

[7] It was certainly surprising to both Huffman and Fano!

recursive subproblem, define $f[n+1] = f[1] + f[2]$. Our earlier exchange argument implies that 1 and 2 are (deepest) siblings in some optimal code for $f[1..n]$.

Let T' be the Huffman tree for $f[3..n+1]$; the inductive hypothesis implies that T' is an optimal code tree for the smaller set of frequencies. To obtain the final code tree T, we replace the leaf labeled $n+1$ with an internal node with two children, labelled 1 and 2. I claim that T is optimal for the original frequency array $f[1..n]$.

To prove this claim, we can express the cost of T in terms of the cost of T' as follows. (In these equations, $depth(i)$ denotes the depth of the leaf labelled i in either T or T'; each leaf that appears in both T and T' has the same depth in both trees.)

$$\begin{aligned}
cost(T) &= \sum_{i=1}^{n} f[i] \cdot depth(i) \\
&= \sum_{i=3}^{n+1} f[i] \cdot depth(i) \;+\; f[1] \cdot depth(1) + f[2] \cdot depth(2) \\
&\qquad\qquad\qquad\qquad\qquad\qquad - \; f[n+1] \cdot depth(n+1) \\
&= cost(T') + (f[1] + f[2]) \cdot depth(T) \;-\; f[n+1] \cdot (depth(T) - 1) \\
&= cost(T') + f[1] + f[2] \;+\; (f[1] + f[2] - f[n+1]) \cdot (depth(T) - 1) \\
&= cost(T') + f[1] + f[2]
\end{aligned}$$

This equation implies that minimizing the cost of T is equivalent to minimizing the cost of T'; in particular, attaching leaves labeled 1 and 2 to the leaf in T' labeled $n+1$ gives an optimal code tree for the original frequencies. □

To efficiently construct a Huffman code, we keep the characters in a priority queue, using the character frequencies as priorities. We can represent the code tree as three arrays of indices, listing the *L*eft and *R*ight children and the *P*arent of each node. The leaves of the final code tree are nodes at indices 1 through n, and the root is the node with index $2n-1$. Pseudocode for the algorithm is shown in Figure 4.4. BuildHuffman performs $O(n)$ priority-queue operations: exactly $2n-1$ Inserts and $2n-2$ ExtractMins. If we implement the priority queue as a standard binary heap, each of these operations requires $O(\log n)$ time, and thus the entire algorithm runs in $\boldsymbol{O(n \log n)}$ ***time***.

Finally, simple algorithms to encode and decode messages using a fixed Huffman code are shown in Figure 4.5; both algorithms run in $O(m)$ time, where m is the length of the encoded message.

```
BuildHuffman(f[1..n]):
  for i ← 1 to n
      L[i] ← 0;  R[i] ← 0
      Insert(i, f[i])
  for i ← n to 2n − 1
      x ← ExtractMin()        ⟨⟨find two rarest symbols⟩⟩
      y ← ExtractMin()
      f[i] ← f[x] + f[y]      ⟨⟨merge into a new symbol⟩⟩
      Insert(i, f[i])
      L[i] ← x;  P[x] ← i     ⟨⟨update tree pointers⟩⟩
      R[i] ← y;  P[y] ← i
  P[2n − 1] ← 0
```

Figure 4.4. Building a Huffman code.

```
HuffmanEncode(A[1..k]):
  m ← 1
  for i ← 1 to k
      HuffmanEncodeOne(A[i])

HuffmanEncodeOne(x):
  if x < 2n − 1
      HuffmanEncodeOne(P[x])
      if x = L[P[x]]
          B[m] ← 0
      else
          B[m] ← 1
      m ← m + 1
```

```
HuffmanDecode(B[1..m]):
  k ← 1
  v ← 2n − 1
  for i ← 1 to m
      if B[i] = 0
          v ← L[v]
      else
          v ← R[v]
      if L[v] = 0
          A[k] ← v
          k ← k + 1
          v ← 2n − 1
```

Figure 4.5. Encoding and decoding algorithms for Huffman codes

4.5 Stable Matching

Every year, thousands of new doctors must obtain internships at hospitals around the United States. During the first half of the 20th century, competition among hospitals for the best doctors led to earlier and earlier offers of internships, sometimes as early as the second year of medical school, along with tighter deadlines for acceptance. In the 1940s, medical schools agreed not to release information until a common date during their students' fourth year. In response, hospitals began demanding faster decisions. By 1950, hospitals would regularly call doctors, offer them internships, and demand *immediate* responses. Interns were forced to gamble if their third-choice hospital called first—accept and risk losing a better opportunity later, or reject and risk having no position at all.[8]

[8]The American academic job market involves similar gambles, at least in computer science. Some departments start making offers in February with two-week decision deadlines; other departments don't even start interviewing until March; MIT notoriously waits until May, when all

Finally, a central clearinghouse for internship assignments, now called the National Resident Matching Program (NRMP), was established in the early 1950s. Each year, doctors submit a ranked list of all hospitals where they would accept an internship, and each hospital submits a ranked list of doctors they would accept as interns. The NRMP then computes an matching between doctors and hospitals that satisfies the following *stability* requirement. A matching is ***unstable*** if there is a doctor α and hospital B that would be both happier with each other than with their current match; that is,

- α is matched with some other hospital A, even though she prefers B.
- B is matched with some other doctor β, even though they prefer α.

In this case, we call (α, B) an *unstable pair* for the matching. The goal of the Resident Match is a ***stable matching***, which is a matching with no unstable pairs.

For simplicity, I'll assume from now on that there are exactly the same number of doctors and hospitals; each hospital offers exactly one internship; each doctor ranks all hospitals and vice versa; and finally, there are no ties in the doctors' or hospitals' rankings.[9]

Some Bad Ideas

At first glance, it is not even clear that a stable matching always exists! Certainly not *every* matching of doctors and hospitals is stable. Suppose there are three doctors (Dr. Quincy, Dr. Rotwang, Dr. Shephard, represented by lower-case letters) and three hospitals (Arkham Asylum, Bethlem Royal Hospital, and County General Hospital, represented by upper-case letters), who rank each other as follows:

q	r	s
A	C	A
C	A	B
B	B	C

A	B	C
r	s	q
q	q	r
s	r	s

The matching $\{Aq, Br, Cs\}$ is unstable, because Arkham would rather hire Dr. Rotwang than Dr. Quincy, and Dr. Rotwang would rather work at Arkham than at Bedlam. (A, r) is an unstable pair for this matching.

its interviews are over, before making *any* faculty offers. Needless to say, the mishmash of offer dates and decision deadlines causes tremendous stress, for candidates and departments alike. For similar reasons, since 1965, most American universities have agreed to a common April 15 deadline for prospective graduate students to accept offers of financial support (and by extension, offers of admission).

[9] In reality, most hospitals offer multiple internships, each doctor ranks only a subset of the hospitals and vice versa, and there are typically more internships than interested doctors. And then it starts getting complicated.

One might imagine using an incremental algorithm that starts with an arbitrary matching, and then greedily performs exchanges to resolve instabilities. Unfortunately, resolving one instability can create new ones; in fact, this incremental "improvement" can lead to an infinite loop. For example, if we start with our earlier unstable matching $\{Aq, Br, Cs\}$, each of the following exchanges resolves one unstable pair (indicated over the arrow), but the sequence of exchanges leads back to the original matching:[10]

$$\{Aq, Br, Cs\} \xrightarrow{Ar} \{Ar, Bq, Cs\} \xrightarrow{Cr} \{As, Bq, Cr\} \xrightarrow{Cq} \{As, Br, Cq\} \xrightarrow{Aq} \{Aq, Br, Cs\}$$

Alternatively, we might try the following multi-round greedy protocol. In each round, every unmatched hospital makes an offer to their favorite unmatched doctor, then every unmatched doctor with an offer accepts their favorite offer. It's not hard to prove that at least one new doctor-hospital pair is matched in each round, so the algorithm always ends with a matching. For the previous example input, we already have a stable matching $\{Ar, Bs, Cq\}$ at the end of the first round! But consider the following input instead:

q	r	s
C	A	A
B	C	B
A	B	C

A	B	C
q	q	s
s	r	r
r	s	q

In the first round, Dr. Shephard accepts an offer from County, and Dr. Quincy accepts an offer from Bedlam (rejecting Arkham's offer), leaving only Dr. Rotwang and Arkham unmatched. Thus, the protocol ends with the matching $\{Ar, Bq, Cs\}$ after two rounds. Unfortunately, this matching is unstable; Arkham and Dr. Shephard prefer each other to their matches.

The Boston Pool and Gale-Shapley Algorithms

In 1952, the NRMP adopted the "Boston Pool" algorithm to assign interns, so named because it had been previously used by a regional clearinghouse in the Boston area. Ten years later, David Gale and Lloyd Shapley described and formally analyzed a generalization of the Boston Pool algorithm and proved that it computes a stable matching. Gale and Shapley used the metaphor of college admissions. Essentially the same algorithm was independently developed by Elliott Peranson in 1972 for use in medical school admissions. Similar algorithms have since been adopted for many other matching markets, including faculty hiring in France, hiring of new economics PhDs in the United States, university admission in Germany, public school admission in New York and Boston, billet assignments for US Navy sailors, and kidney-matching programs.

[10] This example was discovered by Donald Knuth.

Shapley was awarded the 2012 Nobel Prize in Economics for his research on stable matchings, together with Alvin Roth, who significantly extended Shapley's work and used it to develop several real-world exchanges. (Gale did not share the prize, because he died in 2008.)

Like our last failed greedy algorithm, the Gale-Shapley algorithm proceeds in rounds until every position has been accepted. Each round has two stages:

1. An arbitrary unmatched hospital A offers its position to the best doctor α (according to A's preference list) who has not already rejected it.
2. If α is unmatched, she (tentatively) accepts A's offer. If α already has a match but prefers A, she rejects her current match and (tentatively) accepts the new offer from A. Otherwise, α rejects the new offer.

Each doctor ultimately accepts the best offer that she receives, according to her preference list.[11] In short, hospitals make offers greedily, and doctors accept offers greedily. The doctors' ability to reject their current matches in favor of better offers is the key to making this mutual greedy strategy work.

For example, suppose that there are four doctors (Dr. Quincy, Dr. Rotwang, Dr. Shephard, and Dr. Tam) and four hospitals (Arkham Asylum, Bethlem Royal Hospital, County General Hospital, and The Dharma Initiative), who rank each other as follows:

q	r	s	t
A	A	B	D
B	D	A	B
C	C	C	C
D	B	D	A

A	B	C	D
t	r	t	s
s	t	r	r
r	q	s	q
q	s	q	t

Given these preference lists as input, the Gale-Shapley algorithm might proceed as follows:

1. Arkham makes an offer to Dr. Tam.
2. Bedlam makes an offer to Dr. Rotwang.
3. County makes an offer to Dr. Tam, who rejects her earlier offer from Arkham.
4. Dharma makes an offer to Dr. Shephard. (From this point on, there is only one unmatched hospital, so the algorithm has no more choices.)
5. Arkham makes an offer to Dr. Shephard, who rejects her earlier offer from Dharma.

[11] The 1952 Boston Pool algorithm is a special case of the Gale-Shapley algorithm that executes offers in a particular order. Roughly speaking, each offer is made by a hospital X whose favorite doctor (among those who haven't rejected X already) ranks X highest. Because the order of offers depends on the entire set of preference lists, this algorithm *must* be executed by a central authority; in contrast, the Gale-Shapley algorithm does not even require each participant to know their own preferences in advance, as long as they behave consistently with some fixed rankings.

6. Dharma makes an offer to Dr. Rotwang, who rejects her earlier offer from Bedlam.
7. Bedlam makes an offer to Dr. Tam, who rejects her earlier offer from County.
8. County makes an offer to Dr. Rotwang, who rejects it.
9. County makes an offer to Dr. Shephard, who rejects it.
10. County makes an offer to Dr. Quincy.

After the tenth round, all pending offers are accepted, and the algorithm returns the matching $\{As, Bt, Cq, Dr\}$. You can (and should) verify by brute force that this matching is stable, even though no doctor was hired by her favorite hospital, and no hospital hired their favorite doctor; in fact, County ended up hiring their *least* favorite doctor. This is not the only stable matching for these preference lists; the matching $\{Ar, Bs, Cq, Dt\}$ is also stable.

Running Time

Analyzing the number of offers performed by the algorithm is relatively straightforward (which is why we're doing it first). Each hospital makes an offer to each doctor at most once, so the algorithm makes at most n^2 offers.

To analyze the actual *running time*, however, we need to specify the algorithm in more detail. How are the preference lists given to the algorithm? How does the algorithm decide whether any hospital is unmatched, and if so, how does it find an unmatched hospital? How does the algorithm store the tentative matchings? How does the algorithm decide whether a doctor prefers her new offer to her current match? Most fundamentally: *How does the algorithm actually represent doctors and hospitals?*

One possibility is to represent each doctor and hospital by a unique integer between 1 and n, and to represent preferences as two arrays $Dpref[1..n, 1..n]$ and $Hpref[1..n, 1..n]$, where $Dpref[i, r]$ represents the rth hospital in doctor i's preference list, and $HPref[j, r]$ represents the rth doctor in hospital j's preference list. With the input in this form, the Boston Pool algorithm can execute each offer in constant time, after some initial preprocessing; the overall implementation runs in $\boldsymbol{O(n^2)}$ ***time***. We leave the remaining details as a straightforward exercise.

A somewhat harder exercise is to prove that there are inputs (and choices of who makes offers when) that force $\Omega(n^2)$ offers to be made before the algorithm halts. Thus, our $O(n^2)$ upper bound on the worst-case running time is *tight*.

Correctness

But why is the algorithm *correct* at all? How do we know that it always computes a stable matching, or *any* complete matching for that matter?

Once a doctor receives an offer, she has at least a tentative match for the rest of time. Equivalently, if any doctor is unmatched, then no hospital has offered that doctor a job, which implies that the hospitals have not exhausted their preference lists. It follows that when the algorithm terminates (after at most n^2 rounds), every doctor is matched, and therefore every position is filled. In other words, the algorithm always computes a perfect matching between doctors and hospitals. (Whew!) It remains only to prove that the resulting matching is stable.

Suppose the algorithm matches some doctor α to some hospital A, even though she prefers another hospital B. Because every doctor accepts the best offer she receives, α received no offer she liked more than A; in particular, B never made an offer to α. On the other hand, B made offers to every doctor they prefer over their final match β. It follows that B prefers β over α, which means (α, B) is *not* an unstable pair. We conclude that there are no unstable pairs; the matching is stable!

Optimality!

Surprisingly, the correctness of the Gale-Shapley algorithm does not depend on which hospital makes its offer in each round. In fact, no matter which unassigned hospital makes an offer in each round, *the algorithm always computes the same matching*! Let's say that α is a ***feasible*** doctor for A if there is a stable matching that assigns doctor α to hospital A.

Lemma 4.7. *During the Gale-Shapley algorithm, each hospital A is rejected only by doctors that are infeasible for A.*

Proof: We prove the lemma by induction on the number of rounds. Consider an arbitrary round of the algorithm, in which doctor α rejects one hospital A for another hospital B. The rejection implies that α prefers B to A. Each doctor that appears higher than α in B's preference list already rejected B in an earlier round and therefore, by the inductive hypothesis, is infeasible for B.

Now consider an arbitrary matching (of the same doctors and hospitals) that assigns α to A. We already established that α prefers B to A. If B prefers α to its partner, the matching is unstable. On the other hand, if B prefers its partner to α, then (by our earlier argument) its partner is infeasible, and again the matching is unstable. We conclude that there is no stable matching that assigns α to A. □

Now let ***best*(*A*)** denote the highest-ranked *feasible* doctor on A's preference list. Lemma 4.7 implies that every doctor that A prefers to its final match is infeasible for A. On the other hand, the final matching is stable, so the doctor assigned to A must be feasible for A. The following result is now immediate:

Corollary 4.8. *The Gale-Shapley algorithm matches best(A) with A, for every hospital A.*

In other words, the Gale-Shapley algorithm computes the *best possible* stable matching from the hospitals' point of view. It turns out that this matching is also the *worst* possible from the doctors' point of view! Let ***worst(α)*** denote the lowest-ranked feasible hospital on doctor α's preference list.

Corollary 4.9. *The Gale-Shapley algorithm matches α with worst(α), for every doctor α.*

Proof: Suppose Gale and Shapley assign doctor α to hospital A; we need to show that $A = worst(\alpha)$. Consider an arbitrary stable matching where A is *not* matched with α but with another doctor β. The previous corollary implies that A prefers $\alpha = best(A)$ to β. Because the matching is stable, α must therefore prefer her assigned hospital to A. This argument works for *any* stable matching, so α prefers *every* other feasible match to A; in other words, $A = worst(\alpha)$. □

A subtle consequence of these two corollaries, discovered by Lester Dubins and David Freedman in 1981, is that a doctor can potentially improve her match by lying about her preferences, but a hospital cannot. (However, a set of hospitals can collude so that *some* of their matches improve.) Partly for this reason, the National Residency Matching Program reversed its matching algorithm in 1998, so that potential residents offer to work for hospitals, according to their preference orders, and each hospital accepts its best offer. Thus, the new algorithm computes the best possible stable matching for the doctors, and the worst possible stable matching for the hospitals. In practice, however, this reversal altered less than 1% of the residents' matches. As far as I know, the precise effect of this change on the *patients* is an open problem.

Exercises

Caveat lector: Some of these exercises *cannot* be solved using greedy algorithms! Whenever you describe and analyze a greedy algorithm, you ***must*** also include a proof that your algorithm is correct; this proof will typically take the form of an exchange argument. These proofs are especially important in classes (like mine) that do *not* normally require proofs of correctness.

1. The GreedySchedule algorithm we described for the class scheduling problem is not the only greedy strategy we could have tried. For each of the following alternative greedy strategies, either prove that the resulting algorithm always constructs an optimal schedule, or describe a small input

example for which the algorithm does *not* produce an optimal schedule. Assume that all algorithms break ties arbitrarily (that is, in a manner that is completely out of your control). *[Hint: **Three** of these algorithms are actually correct.]*

(a) Choose the course x that *ends last*, discard classes that conflict with x, and recurse.

(b) Choose the course x that *starts first*, discard all classes that conflict with x, and recurse.

(c) Choose the course x that *starts last*, discard all classes that conflict with x, and recurse.

(d) Choose the course x with *shortest duration*, discard all classes that conflict with x, and recurse.

(e) Choose a course x that *conflicts with the fewest other courses*, discard all classes that conflict with x, and recurse.

(f) If no classes conflict, choose them all. Otherwise, discard the course with *longest duration* and recurse.

(g) If no classes conflict, choose them all. Otherwise, discard a course that *conflicts with the most other courses* and recurse.

(h) Let x be the class with the *earliest start time*, and let y be the class with the *second earliest start time*.

- If x and y are disjoint, choose x and recurse on everything but x.
- If x completely contains y, discard x and recurse.
- Otherwise, discard y and recurse.

(i) If any course x completely contains another course, discard x and recurse. Otherwise, choose the course y that *ends last*, discard all classes that conflict with y, and recurse.

2. Now consider a weighted version of the class scheduling problem, where different classes offer different number of credit hours (totally unrelated to the duration of the class lectures). Your goal is now to choose a set of non-conflicting classes that give you the largest possible number of credit hours, given arrays of start times, end times, and credit hours as input.

(a) Prove that the greedy algorithm described at the beginning of this chapter—Choose the class that ends first and recurse—does *not* always return an optimal schedule.

(b) Prove that ***none*** of the greedy algorithms described in Exercise 1 always return an optimal schedule. *[Hint: Solve Exercise 1 first; the algorithms that don't work there don't work here, either.]*

(c) Describe and analyze an algorithm that always computes an optimal schedule. *[Hint: Your algorithm will not be greedy.]*

3. Let X be a set of n intervals on the real line. We say that a subset of intervals $Y \subseteq X$ *covers* X if the union of all intervals in Y is equal to the union of all intervals in X. The *size* of a cover is just the number of intervals.

   Describe and analyze an efficient algorithm to compute the smallest cover of X. Assume that your input consists of two arrays $L[1..n]$ and $R[1..n]$, representing the left and right endpoints of the intervals in X. If you use a greedy algorithm, you must prove that it is correct.

A set of intervals, with a cover (shaded) of size 7.

4. Let X be a set of n intervals on the real line. We say that a set P of points *stabs* X if every interval in X contains at least one point in P. Describe and analyze an efficient algorithm to compute the smallest set of points that stabs X. Assume that your input consists of two arrays $L[1..n]$ and $R[1..n]$, representing the left and right endpoints of the intervals in X. As usual, If you use a greedy algorithm, you must prove that it is correct.

A set of intervals stabbed by four points (shown here as vertical segments)

5. Let X be a set of n intervals on the real line. A *proper coloring* of X assigns a color to each interval, so that any two overlapping intervals are assigned different colors. Describe and analyze an efficient algorithm to compute the minimum number of colors needed to properly color X. Assume that your input consists of two arrays $L[1..n]$ and $R[1..n]$, representing the left and right endpoints of the intervals in X. As usual, if you use a greedy algorithm, you must prove that it is correct.

A proper coloring of a set of intervals using five colors.

6. (a) For every integer n, find a frequency array $f[1..n]$ whose Huffman code tree has depth $n-1$, such that the largest frequency is as small as possible.

   (b) Suppose the total length N of the *unencoded* message is bounded by a polynomial in the alphabet size n. Prove that the any Huffman tree for the frequencies $f[1..n]$ has depth $O(\log n)$.

♥7. Call a frequency array $f[1..n]$ ***α-heavy*** if it satisfies two conditions:

   - $f[1] > f[i]$ for all $i > 1$; that is, 1 is the *unique* most frequent symbol.
   - $f[1] \geq \alpha \sum_{i=1}^{n} f[i]$; that is, at least an α fraction of the symbols are 1s.

   Find the largest real number α such that in every Huffman code for every α-heavy frequency array, symbol 1 is represented by a single bit. *[Hint: First prove that* $1/3 \leq \alpha \leq 1/2$*.]*

8. Describe and analyze an algorithm to compute an optimal *ternary* prefix-free code for a given array of frequencies $f[1..n]$. Don't forget to prove that your algorithm is correct for *all* n.

9. Describe in detail how to implement the Gale-Shapley stable matching algorithm, so that the worst-case running time is $O(n^2)$, as claimed earlier in this chapter.

10. (a) Prove that it is possible for the Gale-Shapley algorithm to perform $\Omega(n^2)$ offers before termination. (You need to describe both a suitable input and a sequence of $\Omega(n^2)$ valid offers.)

    (b) Describe for any integer n a set of preferences for n doctors and n hospitals that forces the Gale-Shapley algorithm to execute $\Omega(n^2)$ rounds, *no matter which valid proposal is made in each round. [Hint: Part (b) implies part (a).]*

11. Describe and analyze an efficient algorithm to determine whether a given set of hospital and doctor preferences has to a *unique* stable matching.

12. Consider a generalization of the stable matching problem, where some doctors do not rank all hospitals and some hospitals do not rank all doctors, and a doctor can be assigned to a hospital only if each appears in the other's preference list. In this case, there are three additional unstable situations:

    - A matched hospital prefers an unmatched doctor to its assigned match.
    - A matched doctor prefers an unmatched hospital to her assigned match.
    - An unmatched doctor and an unmatched hospital appear in each other's preference lists.

A stable matching in this setting may leave some doctors and/or hospitals unmatched, even though their preference lists are non-empty. For example, if every doctor lists Harvard as their only acceptable hospital, and every hospital lists Dr. House as their only acceptable intern, then only House and Harvard will be matched.

Describe and analyze an efficient algorithm that computes a stable matching in this more general setting. *[Hint: Reduce to an instance where every doctor ranks every hospital and vice versa, and then invoke Gale-Shapley.]*

13. The Scandinavian furniture company Fürni has hired n drivers to deliver n identical orders to n different addresses in Wilmington, Delaware. Each driver has their own well-established delivery route through Wilmington that visits all n addresses. Assuming they follow their routes as they always do, two drivers never visit the same addresses at the same time.

    In principle, each of the n drivers can deliver their furniture to any of the n addresses, but there's a complication. One of the drivers has secretly wired proximity sensors and explosives to the Johannshamn sofas (with the Strinne green stripe pattern). If two sofas are ever at the same address at the same time, both will explode, destroying both the delivery truck and the building at that address. This can only happen if one driver delivers an order to that address, and then later another driver visits that same address *while the furniture is still on their truck*.

    Your job as the Fürni dispatcher is to assign each driver to a delivery address. Describe an algorithm to assign addresses to drivers so that each of the n addresses receives their furniture order and there are no explosions.

    For example, suppose Jack's route visits 537 Paper Street at 6pm and 1888 Franklin Street at 8pm, and Marla's route visits 537 Paper at 7pm and 1888 Franklin at 9pm. Then Jack should deliver to 1888 Franklin, and Marla should deliver to 537 Paper; otherwise, there would be an explosion at 1888 Franklin at 8pm. (Cue the Pixies.) *[Hint: Jack and Marla are a bit unstable.]*

14. Suppose you are a simple shopkeeper living in a country with n different types of coins, with values $1 = c[1] < c[2] < \cdots < c[n]$. (In the U.S., for example, $n = 6$ and the values are 1, 5, 10, 25, 50 and 100 cents.) Your beloved and benevolent dictator, El Generalissimo, has decreed that whenever you give a customer change, you must use the smallest possible number of coins, so as not to wear out the image of El Generalissimo lovingly engraved on each coin by servants of the Royal Treasury.

    (a) In the United States, there is a simple greedy algorithm that always results in the smallest number of coins: subtract the largest coin and

recursively give change for the remainder. El Generalissimo does not approve of American capitalist greed. Show that there is a set of coin values for which the greedy algorithm does *not* always give the smallest possible of coins.

(b) Now suppose El Generalissimo decides to impose a currency system where the coin denominations are consecutive powers $b^0, b^1, b^2, \ldots, b^k$ of some integer $b \geq 2$. Prove that despite El Generalissimo's disapproval, the greedy algorithm described in part (a) does make optimal change in this currency system.

(c) Describe and analyze an efficient algorithm to determine, given a target amount T and a sorted array $c[1..n]$ of coin denominations, the smallest number of coins needed to make T cents in change. Assume that $c[1] = 1$, so that it is possible to make change for any amount T.

15. Suppose you are given an array $A[1..n]$ of integers, each of which may be positive, negative, or zero. A contiguous subarray $A[i..j]$ is called a ***positive interval*** if the sum of its entries is greater than zero. Describe and analyze an algorithm to compute the minimum number of positive intervals that cover every positive entry in A. For example, given the following array as input, your algorithm should output 3. If every entry in the input array is negative, your algorithm should output 0.

$$\overbrace{\boxed{+3 \mid -5 \mid +7 \mid -4 \mid +1}}^{\text{sum}=2} \boxed{-8} \overbrace{\boxed{+3 \mid -7 \mid +5}}^{\text{sum}=1} \boxed{-9} \overbrace{\boxed{+5 \mid -2 \mid +4}}^{\text{sum}=7}$$

16. Consider the following process. At all times you have a single positive integer x, which is initially equal to 1. In each step, you can either *increment* x or *double* x. Your goal is to produce a target value n. For example, you can produce the integer 10 in four steps as follows:

$$1 \xrightarrow{+1} 2 \xrightarrow{\times 2} 4 \xrightarrow{+1} 5 \xrightarrow{\times 2} 10$$

Obviously you can produce any integer n using exactly $n-1$ increments, but for almost all values of n, this is horribly inefficient. Describe and analyze an algorithm to compute the *minimum* number of steps required to produce any given integer n.

17. Suppose we have n skiers with heights given in an array $P[1..n]$, and n skis with heights given in an array $S[1..n]$. Describe an efficient algorithm to assign a ski to each skier, so that the average difference between the height of a skier and her assigned ski is as small as possible. The algorithm should

compute a permutation σ such that the expression

$$\frac{1}{n}\sum_{i=1}^{n}\left|P[i]-S[\sigma(i)]\right|$$

is as small as possible.

18. Alice wants to throw a party and she is trying to decide who to invite. She has n people to choose from, and she knows which pairs of these people know each other. She wants to invite as many people as possible, subject to two constraints:

    - For each guest, there should be at least five other guests that they already know.
    - For each guest, there should be at least five other guests that they *don't* already know.

    Describe and analyze an algorithm that computes the largest possible number of guests Alice can invite, given a list of n people and the list of pairs who know each other.

19. Suppose we are given two arrays $C[1..n]$ and $R[1..n]$ of positive integers. An $n \times n$ matrix of 0s and 1s *agrees with R and C* if, for every index i, the ith row contains $R[i]$ 1s, and the ith column contains $C[i]$ 1s. Describe and analyze an algorithm that either constructs a matrix that agrees with R and C, or correctly reports that no such matrix exists.

20. You've just accepted a job from Elon Musk, delivering burritos from San Francisco to New York City. You get to drive a Burrito-Delivery Vehicle through Elon's new Transcontinental Underground Burrito-Delivery Tube, which runs in a direct line between these two cities.[12]

    Your Burrito-Delivery Vehicle runs on single-use batteries, which must be replaced after at most 100 miles. The actual fuel is virtually free, but the batteries are expensive and fragile, and therefore must be installed only by official members of the Transcontinental Underground Burrito-Delivery Vehicle Battery-Replacement Technicians' Union.[13] Thus, even if you replace your battery early, you must still pay full price for each new battery to be installed. Moreover, your Vehicle is too small to carry more than one battery at a time.

[12]. . . and which was clearly modeled after Maciej Cegłowski's fictional "Alameda-Weehauken Burrito Tunnel"

[13]or as they call themselves in German, Die Transkontinentaluntergrundburritolieferfahrzeugbatteriewechseltechnikervereinigung.

There are several fueling stations along the Tube; each station charges a different price for installing a new battery. Before you start your trip, you carefully print the Wikipedia page listing the locations and prices of every fueling station along the Tube. Given this information, how do you decide the best places to stop for fuel?

More formally, suppose you are given two arrays $D[1..n]$ and $C[1..n]$, where $D[i]$ is the distance from the start of the Tube to the ith station, and $C[i]$ is the cost to replace your battery at the ith station. Assume that your trip starts and ends at fueling stations (so $D[1] = 0$ and $D[n]$ is the total length of your trip), and that your car starts with an empty battery (so you must install a new battery at station 1).

(a) Describe and analyze a greedy algorithm to find the minimum number of refueling stops needed to complete your trip. Don't forget to prove that your algorithm is correct.

(b) But what you really want to minimize is the total *cost* of travel. Show that your greedy algorithm in part (a) does *not* produce an optimal solution when extended to this setting.

(c) Describe an efficient algorithm to compute the locations of the fuel stations you should stop at to minimize the total cost of travel.

21. You've been hired to store a sequence of n books on shelves in a library. The order of the books is fixed by the cataloging system and cannot be changed; each shelf must store a contiguous interval of the given sequence of books. You are given two arrays $H[1..n]$ and $T[1..n]$, where $H[i]$ and $T[i]$ are respectively the height and thickness of the ith book in the sequence. All shelves in this library have the same length L; the total thickness of all books on any single shelf cannot exceed L.

(a) Suppose all the books have the same height h and the shelves have height larger than h, so every book fits on every shelf. Describe and analyze a greedy algorithm to store the books in as few shelves as possible. *[Hint: The algorithm is obvious, but why is it correct?]*

(b) That was a nice warmup, but now here's the real problem. In fact the books have different heights, but you can adjust the height of each shelf to match the tallest book on that shelf. (In particular, you can change the height of any empty shelf to zero.) Now your task is to store the books so that the sum of the heights of the shelves is as small as possible. Show that your greedy algorithm from part (a) does *not* always give the best solution to this problem.

(c) Describe and analyze an algorithm to find the best matching between books and shelves as described in part (b).

22. A string w of parentheses (and) is ***balanced*** if it satisfies one of the following conditions:

    - w is the empty string.
    - $w = (x)$ for some balanced string x
    - $w = xy$ for some balanced strings x and y

    For example, the string

    $$w = ((())()())(()())()$$

    is balanced, because $w = xy$, where

    $$x = ((())()()) \qquad \text{and} \qquad y = (()()))().$$

    (a) Describe and analyze an algorithm to determine whether a given string of parentheses is balanced.

    (b) Describe and analyze a greedy algorithm to compute the length of a longest balanced subsequence of a given string of parentheses. As usual, don't forget to prove your algorithm is correct.

    For both problems, your input is an array $w[1..n]$, where for each i, either $w[i] = ($ or $w[i] =)$. Both of your algorithms should run in $O(n)$ time.

23. One day Alex got tired of climbing in a gym and decided to take a large group of climber friends outside to climb. They went to a climbing area with a huge wide boulder, not very tall, with several marked hand and foot holds. Alex quickly determined an "allowed" set of moves that her group of friends can perform to get from one hold to another.

    The overall system of holds can be described by a rooted tree T with n vertices, where each vertex corresponds to a hold and each edge corresponds to an allowed move between holds. The climbing paths converge as they go up the boulder, leading to a unique hold at the summit, represented by the root of T.

    Alex and her friends (who are all excellent climbers) decided to play a game, where as many climbers as possible are simultaneously on the boulder and each climber needs to perform a sequence of *exactly* k moves. Each climber can choose an arbitrary hold to start from, and all moves must move away from the ground. Thus, each climber traces out a path of k edges in the tree T, all directed toward the root. However, no two climbers are allowed to touch the same hold; the paths followed by different climbers cannot intersect at all.

    (a) Describe and analyze a greedy algorithm to compute the maximum number of climbers that can play this game. Your algorithm is given

a rooted tree T and an integer k as input, and it should compute the largest possible number of disjoint paths in T, where each path has length k. Do ***not*** assume that T is a binary tree. For example, given the tree below as input, your algorithm should return the integer 8.

Figure 4.6. Seven disjoint paths of length $k = 3$. This is *not* the largest such set of paths in this tree.

(b) Now suppose each vertex in T has an associated *reward*, and your goal is to maximize the total reward of the vertices in your paths, instead of the total number of paths. Show that your greedy algorithm does *not* always return the optimal reward.

(c) Describe an efficient algorithm to compute the maximum possible reward, as described in part (b).

24. Congratulations! You have successfully conquered Camelot, transforming the former battle-scarred hereditary monarchy into an anarcho-syndicalist commune, where citizens take turns to act as a sort of executive-officer-for-the-week, but with all the decisions of that officer ratified at a special bi-weekly meeting, by a simple majority in the case of purely internal affairs, but by a two-thirds majority in the case of more major. . . .

As a final symbolic act, you order the Round Table (surprisingly, an actual circular table) to be split into pizza-like wedges and distributed to the citizens of Camelot as trophies. Each citizen has submitted a request for an angular wedge of the table, specified by two angles—for example: Sir Robin the Brave might request the wedge from 17.23° to 42°, and Sir Lancelot the Pure might request the 2° wedge from 359° to 1°. Each citizen will be happy if and only if they receive *precisely* the wedge that they requested. Unfortunately, some of these ranges overlap, so satisfying *all* the citizens' requests is simply impossible. Welcome to politics.

Describe and analyze an algorithm to find the maximum number of requests that can be satisfied. *[Hint: The output of your algorithm should not change if you rotate the table. Do not assume that angles are integers.]*

25. Suppose you are standing in a field surrounded by several large balloons. You want to use your brand new Acme Brand Zap-O-Matic™ to pop all the balloons, without moving from your current location. The Zap-O-Matic™ shoots a high-powered laser beam, which pops all the balloons it hits. Since each shot requires enough energy to power a small country for a year, you want to fire as few shots as possible.

Figure 4.7. Nine balloons popped by four shots of the Zap-O-Matic™

The *minimum zap* problem can be stated more formally as follows. Given a set C of n circles in the plane, each specified by its radius and the (x, y) coordinates of its center, compute the minimum number of rays from the origin that intersect every circle in C. Your goal is to find an efficient algorithm for this problem.

(a) Suppose it is possible to shoot a ray that does not intersect any balloons. Describe and analyze a greedy algorithm that solves the minimum zap problem in this special case. *[Hint: See Exercise 4.]*

(b) Describe and analyze a greedy algorithm whose output is within 1 of optimal. That is, if m is the minimum number of rays required to hit every balloon, then your greedy algorithm must output either m or $m+1$. (Of course, you must prove this fact.)

(c) Describe an algorithm that solves the minimum zap problem in $O(n^2)$ time.

♥(d) Describe an algorithm that solves the minimum zap problem in $O(n \log n)$ time.

Assume you have a subroutine that tells you the range of angles of rays that intersects an arbitrary circle c in $O(1)$ time. This subroutine is not difficult to write, but it's not the interesting part of the problem.

[T]he distributions and partitions of knowledge are not like several lines that meet in one angle, and so touch but in a point, but are like branches of a tree that meet in a stem, which hath a dimension and quantity of entireness and continuance before it come to discontinue and break itself into arms and boughs.

— Francis Bacon, *The Advancement of Learning* (1605)

Thus you see, most noble Sir, how this type of solution bears little relationship to mathematics, and I do not understand why you expect a mathematician to produce it, rather than anyone else.

— Leonhard Euler, describing the Königsburg bridge problem in a letter to Carl Leonhard Gottlieb Ehler (April 3, 1736)

Well, ya turn left by the fire station in the village and take the old post road by the reservoir and. . . no, that won't do.

Best to continue straight on by the tar road until you reach the schoolhouse and then turn left on the road to Bennett's Lake until. . . no, that won't work either.

East Millinocket, ya say? Come to think of it, you can't get there from here.

— Robert Bryan and Marshall Dodge, *Bert and I and Other Stories from Down East* (1961)

5

Basic Graph Algorithms

5.1 Introduction and History

A graph is a collection of pairs—pairs of integers, pairs of people, pairs of cities, pairs of stars, pairs of countries, pairs of scientific papers, pairs of web pages, pairs of game positions, pairs of recursive subproblems, even pairs of graphs. Mirroring the most common method for visualizing graphs, the underlying objects being paired are usually called ***vertices*** or ***nodes***, and the pairs themselves are called ***edges*** or ***arcs***, but in fact the objects and pairs can be anything at all.

One of the earliest examples of graphs are *road networks* and maps thereof. Roman engineers constructed a network of more than 400 000 km of public roads across Europe, western and central Asia, and northern Africa during the height of the Roman empire. Travelers on the road network would carry *itineraria*, which were either simple lists or more pictorial representations of the landmarks and distances along various roads. The *Tabula Peutingeriana*, a 13th-century

scroll depicting the entire Roman *cursus publicus*, is widely believed to be a medieval copy of a 5th-century revision of a 1st-century *itinerarium pictum*, commissioned during the reign of Augustus Caesar. The Peutinger Table is not a geographically accurate map—historians debate whether it qualifies as a "map" at all!—but an abstract representation of the road network, similar to a modern subway map. Cities along each road are indicated by kinks in the curve representing that road; the names of these cites and the lengths of road segments between them are also indicated on the map. Thus, the map contains enough information to find the shortest route between any two cities in the 5th-century Roman empire. See Figure 5.1.

Figure 5.1. A small excerpt of Konrad Miller's 1872 restoration of the *Tabula Peutingeriana*, showing the Roman road from modern-day Birten (*Veteribus*, top left) through Köln (*Agripina*) and Bonn (*Bonnae*) to Mainz (*Mogontiaco*, top right), with branches to Trier (*Avg Tresvirorvm*, center) and Metz (*Matricorvm*, bottom center). (See Image Credits at the end of the book.)

One of the oldest classical applications of graphs—and specifically trees—is in representing genealogies. Complex family "trees" have been used for centuries to settle legal questions about marriage, inheritance, and royal succession. Civil law in the Roman empire, later adopted as canon law by the early Catholic Church, forbade marriage between first cousins or closer relatives. In the early ninth century, the Church changed both the required distance and the method of computation. Where the Roman *computatio legalis* required the sum of the distances to the nearest common ancestor to be at least four, the newer *computatio canonica* required the maximum of the two distances to be at least seven. In 1215, bowing to practical considerations (and actual practice), the Church relaxed the minimum required distance for marriage to four.[1] The left diagram in Figure 5.2 illustrates a particularly convoluted case: Tirius and Theburga marry and have a son Gaius, after which Tirius dies; Theburga then

[1]During the 11th and 12th centuries, this restriction gradually expanded to include up to four links by affinity, initially through marriage, and later through extra-marital sex, betrothal, and even godparenting. For example, marriage between a man and his sister's husband's sister's husband's sister was formally forbidden, as was a marriage between a widower and his son's wife's widowed mother. These affinity requirements were significantly reduced but not eliminated in 1215; the Church only abandoned the concept of affinity *ex copula illicita* in 1917.

marries Lothar, bears him a son, and dies; finally, Lothar and Bertha marry and have a daughter Gemma. Can Gaius's son legally marry Gemma's daughter?

Figure 5.2. Two diagrams describing a complex marriage case, from an anonymous 15th-century treatise on Johannes Andreae's *Super arboribus consanguinitatis et affinitatis*, an early 14th-century treatise on canon law. (See Image Credits at the end of the book.)

In the late 1600s, French mathematician Pierre Varignon developed a graphical method for finding the equilibrium position of a tree-like network of ropes under tension, building on earlier work by Simon Stevin published a century earlier. Varignon observed that when the ropes are at equilibrium, one can draw a graph whose edges are segments parallel to the ropes, with lengths equal to the forces along those ropes, such that the ropes meeting at any point in the network define a closed cycle in the graph. Varignon's method of "graphical statics" was not published in complete detail until 1725, two years after his death. These graphs are now known as *reciprocal force diagrams* or *Maxwell-Cremona diagrams*, after James Clerk Maxwell and Luigi Cremona, who (along with Carl Culmann and others) developed a rich theory of reciprocal diagrams in the late 1800s.

Figure 5.3. Reciprocal force diagrams (dotted), from Varignon's posthumous *Nouvelle mécanique, ou statique, dont le projet fut donné en MDCLXXXVII [New mechanics, or statics, whose project was given in 1687]* (See Image Credits at the end of the book.)

Of course, there are many other familiar examples of graphs, like board games (dating to antiquity); vertices and edges of convex polyhedra (formally studied by ancient Greek philosophers, but much older); visualizations of star patterns (already developed in East Asia by the 7th century CE); knight's tours (described by al-Adli, Rudraṭa, al-Suli, and others in the 9th and 10th centuries), mazes (introduced in their modern form by Giovanni Fontana circa 1420); geodetic triangulations (introduced by Gemma Frisius in 1533, and used to calculate the circumference of the earth by Willebrod Snell in 1615 and to define the meter in 1799), Leonhard Euler's well-known partial[2] solution to the Bridges of Königsburg puzzle (1735); telegraph and other communication networks (first proposed in 1753, developed by Ronalds, Schilling, Gauss, Weber, and others in the early 1800s, and deployed worldwide by the late 1800s); electrical circuits (formalized in the early 1800s by Ohm, Maxwell, Kirchhoff, and others); molecular structural formulas (introduced independently by August Kekulé in 1857 and Archibald Couper in 1858); social networks (first studied in the mid-1930s by sociologist Jacob Moreno); digital electronic circuits (proposed by Charles Sanders Peirce in 1886, and cast into their modern form by Claude Shannon in 1937); and yeah, okay, if you insist, the modern internet.

The word "graph" for the abstract mathematical was coined by James Sylvester in 1878, who adapted Kekulé's "chemicographs" to describe certain algebraic invariants, at the suggestion of his colleague William Clifford. The word "tree" was first used for connected acyclic graphs by Arthur Cayley in 1857, although the abstract concept of trees had already been used by Gustav Kirchhoff and Karl von Staudt ten years earlier. The zeroth book on graph theory was published by André Sainte-Laguë in 1926; Dénes Kőnig published the first graph theory book ten years later.

5.2 Basic Definitions

Formally, a (simple) ***graph*** is a pair of sets (V, E), where V is an arbitrary non-empty finite set, whose elements are called ***vertices***[3] or ***nodes***, and E is a set of pairs of elements of V, which we call ***edges***. In an ***undirected*** graph, the edges are unordered pairs, or just sets of size two; I usually write $\boldsymbol{uv}$ instead of $\{u, v\}$ to denote the undirected edge between u and v. In a ***directed*** graph, the edges are ordered pairs of vertices; I usually write $\boldsymbol{u \to v}$ instead of (u, v) to denote the directed edge from u to v.

[2]Euler dismissed the final step of his argument—actually finding an Euler tour of a graph when every vertex has even degree—as obvious. Euler also failed to notice that a graph with an Euler tour must be connected. The first complete proof that a graph has an Euler tour if and only if it is connected and every vertex has even degree was published by Carl Hierholzer in 1873.

[3]The singular of the English word "vertices" is ***vertex***. Similarly, the singular of "matrices" is ***matrix***, and the singular of "indices" is ***index***. Unless you're speaking Italian, there is no such

Following standard (but admittedly confusing) practice, I will also use V to denote the *number* of vertices in a graph, and E to denote the *number* of edges. Thus, in any undirected graph we have $0 \leq E \leq \binom{V}{2}$, and in any directed graph we have $0 \leq E \leq V(V-1)$.

The ***endpoints*** of an edge uv or $u{\to}v$ are its vertices u and v. We distinguish the endpoints of a directed edge $u{\to}v$ by calling u the ***tail*** and v the ***head***.

The definition of a graph as a pair of *sets* forbids multiple undirected edges with the same endpoints, or multiple directed edges with the same head and the same tail. (The same directed graph can contain both a directed edge $u{\to}v$ and its reversal $v{\to}u$.) Similarly, the definition of an undirected edge as a *set* of vertices forbids an undirected edge from a vertex to itself. Graphs *without* loops and parallel edges are often called ***simple*** graphs; non-simple graphs are sometimes called ***multigraphs***. Despite the formal definitional gap, most algorithms for simple graphs extend to multigraphs with little or no modification, and for that reason, I see no need for a formal definition here.

For any edge uv in an undirected graph, we call u a ***neighbor*** of v and vice versa, and we say that u and v are ***adjacent***. The ***degree*** of a node is its number of neighbors. In directed graphs, we distinguish two kinds of neighbors. For any directed edge $u{\to}v$, we call u a ***predecessor*** of v, and we call v a ***successor*** of u. The ***in-degree*** of a vertex is its number of predecessors; the ***out-degree*** is its number of successors.

A graph $G' = (V', E')$ is a ***subgraph*** of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. A ***proper subgraph*** of G is any subgraph other than G itself.

A ***walk*** in an undirected graph G is a sequence of vertices, where each adjacent pair of vertices are adjacent in G; informally, we can also think of a walk as a sequence of edges. A walk is called a ***path*** if it visits each vertex at most once. For any two vertices u and v in a graph G, we say that v is ***reachable*** from u if G contains a walk (and therefore a path) between u and v. An undirected graph is ***connected*** if every vertex is reachable from every other vertex. Every undirected graph consists of one or more ***components***, which are its maximal connected subgraphs; two vertices are in the same component if and only if there is a path between them.[4]

A walk is ***closed*** if it starts and ends at the same vertex; a ***cycle*** is a closed walk that enters and leaves each vertex at most once. An undirected graph is ***acyclic*** if no subgraph is a cycle; acyclic graphs are also called ***forests***. A ***tree*** is a connected acyclic graph, or equivalently, one component of a forest. A ***spanning***

thing as a vertice, matrice, indice, appendice, helice, apice, vortice, radice, simplice, codice, directrice, dominatrice, Unice, Kleenice, Asterice, Obelice, Dogmatice, Getafice, Cacofonice, Vitalstatistice, Geriatrice, or Jimi Hendrice! If you have trouble remembering this rule, stick to "node".

[4]Components are often called "connected components", but this usage is redundant; components are connected by definition.

tree of an undirected graph G is a subgraph that is a tree and contains every vertex of G. A graph has a spanning tree if and only if it is connected. A ***spanning forest*** of G is a collection of spanning trees, one for each component of G.

Directed graphs require slightly different definitions. A ***directed walk*** is a sequence of vertices $v_0 \to v_1 \to v_2 \to \cdots \to v_\ell$ such that $v_{i-1} \to v_i$ is a directed edge for every index i; directed paths and directed cycles are defined similarly. Vertex v is ***reachable*** from vertex u in a directed graph G if and only if G contains a directed walk (and therefore a directed path) from u to v. A directed graph is ***strongly connected*** if every vertex is reachable from every other vertex. A directed graph is ***acyclic*** if it does not contain a directed cycle; directed acyclic graphs are often called ***dags***.

5.3 Representations and Examples

The most common way to visually represent graphs is by ***drawing*** them. A drawing of a graph maps each vertex to a point in the plane (typically drawn as a small circle or some other shape) and each edge to a curve or straight line segment between the two vertices. A graph is ***planar*** if it has a drawing where no two edges cross; such a drawing is also called an ***embedding***.[5] The same graph can have many different drawings, so it is important not to confuse a particular drawing with the graph itself. In particular, planar graphs can have non-planar drawings!

Figure 5.4. Two drawings of the same disconnected planar graph with 13 vertices, 19 edges, and two components. Only the drawing on the left is an embedding.

However, drawings are far from the only useful representation of graphs. For example, the ***intersection graph*** of a collection of geometric objects has a node for every object and an edge for every intersecting pair of objects. Whether a particular graph can be represented as an intersection graph depends on what kind of object you want to use for the vertices. Different types of objects—line segments, rectangles, circles, etc.—define different classes of graphs. One particularly useful type of intersection graph is an *interval graph*, whose vertices

[5]Confusingly, the word "embedding" is often used as a synonym for "drawing", even when the edges intersect. Please don't do that.

are intervals on the real line, with an edge between any two intervals that overlap.

Figure 5.5. The graph in Figure 5.4 is also the intersection graph of (a) a set of line segments and (b) a set of circles.

Another good example is the ***dependency graph*** of a recursive algorithm. Dependency graphs are directed acyclic graphs. The vertices are all the distinct recursive subproblems that arise when executing the algorithm on a particular input. There is an edge from one subproblem to another if evaluating the second subproblem requires a recursive evaluation of the first. For example, for the Fibonacci recurrence

$$F_n = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ F_{n-1} + F_{n-2} & \text{otherwise,} \end{cases}$$

the vertices of the dependency graph are the integers $0, 1, 2, \ldots, n$, and the edges are the pairs $(i-1)\to i$ and $(i-2)\to i$ for every integer i between 2 and n.

Figure 5.6. The dependency graph of the Piṅgala-Fibonacci recurrence.

As a more complex example, recall the recurrence for the *edit distance* problem from Chapter 3:

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \left\{ \begin{array}{c} Edit(i-1,j)+1 \\ Edit(i,j-1)+1 \\ Edit(i-1,j-1)+[A[i] \neq B[j]] \end{array} \right\} & \text{otherwise} \end{cases}$$

The dependency graph of this recurrence is an $m \times n$ grid of vertices (i,j) connected by vertical edges $(i-1,j)\to(i,j)$, horizontal edges $(i,j-1)\to(i,j)$, and diagonal edges $(i-1,j-1)\to(i,j)$. Dynamic programming works efficiently for any recurrence that has a reasonably small dependency graph; a proper evaluation order ensures that each subproblem is visited *after* its predecessors.

Figure 5.7. The dependency graph of the edit distance recurrence.

Another interesting example is the ***configuration graph*** of a game, puzzle, or mechanism like tic-tac-toe, checkers, the Rubik's Cube, the Tower of Hanoi, or a Turing machine. The vertices of the configuration graph are all the valid configurations of the puzzle; there is an edge from one configuration to another if it is possible to transform one configuration into the other with a single simple "move". (Obviously, the precise definition depends on what moves are allowed.) Even for reasonably simple mechanisms, the configuration graph can be extremely complex, and we typically only have access to local information about the configuration graph.

Figure 5.8. The configuration graph of the 4-disk Tower of Hanoi.

Configuration graphs are close relatives of the *game trees* we considered in Chapter 2, but with one crucial difference. Each state of a game appears *exactly* once in its configuration graph, but can appear many times in its game tree. In short, configuration graphs are *memoized* game trees!

Finite-state automata used in formal language theory can be modeled as labeled directed graphs. Recall that a deterministic finite-state automaton is formally defined as a 5-tuple $M = (\Sigma, Q, s, A, \delta)$, where Σ is a finite set called the *alphabet*, Q is a finite set of *states*, $s \in Q$ is the *start state*, $A \subseteq Q$ is the set of

accepting states, and $\delta : Q \times \Sigma \to Q$ is a *transition function*. But it is often more useful to think of M as a directed graph G_M whose vertices are the states Q, and whose edges have the form $q \to \delta(q, a)$ for every state $q \in Q$ and symbol $a \in \Sigma$. Basic questions about the language $L(M)$ accepted by M can then be phrased as questions about the graph G_M. For example, $L(M) = \varnothing$ if and only if no accepting state/vertex is reachable from the start state/vertex s.

Finally, sometimes one graph can be used to implicitly represent other larger graphs. A good example of this implicit representation is the *subset construction*, which is normally used to convert NFAs into DFAs, but can be applied to *arbitrary* directed graphs as follows. Given *any* directed graph $G = (V, E)$, we can define a new directed graph $G' = (2^V, E')$ whose vertices are all *subsets* of vertices in V, and whose edges E' are defined as follows:

$$E' := \left\{ A \to B \;\middle|\; u \to v \in E \text{ for some } u \in A \text{ and } v \in B \right\}$$

We can mechanically translate this definition into an algorithm to construct G' from G, but strictly speaking, this construction is unnecessary, because G is *already* an *implicit* representation of G'.

It's important not to confuse any of these examples/representations with the actual formal *definition*: A graph is a pair of sets (V, E), where V is an *arbitrary* non-empty finite set, and E is a set of pairs (either ordered or unordered) of elements of V. In short: A graph is a set of pairs of things.

5.4 Data Structures

In practice, graphs are usually represented by one of two standard data structures: *adjacency lists* and *adjacency matrices*. At a high level, both data structures are arrays indexed by vertices; this requires that each vertex has a unique integer identifier between 1 and V. In a formal sense, these integers *are* the vertices.

Adjacency Lists

By far the most common data structure for storing graphs is the ***adjacency list***. An adjacency list is an array of lists, each containing the neighbors of one of the vertices (or the out-neighbors if the graph is directed).[6] For undirected graphs, each edge uv is stored twice, once in u's neighbor list and once in v's neighbor list; for directed graphs, each edge $u \to v$ is stored only once, in the neighbor list of the tail u. For both types of graphs, the overall space required for an adjacency list is $O(V + E)$.

[6] Attentive students might notice that despite is name, an adjacency list is not a list. This nomenclature is an example of the Red Herring Principle: In computer science, as in mathematics, a red herring is neither necessarily red nor necessarily a fish.

There are several different ways to represent these neighbor lists, but the standard implementation uses a simple singly-linked list. The resulting data structure allows us to list the (out-)neighbors of a node v in $O(1+\deg(v))$ time; just scan v's neighbor list. Similarly, we can determine whether $u \to v$ is an edge in $O(1+\deg(u))$ time by scanning the neighbor list of u. For undirected graphs, we can improve the time to $O(1+\min\{\deg(u), \deg(v)\})$ by simultaneously scanning the neighbor lists of both u and v, stopping either when we locate the edge or when we fall of the end of a list.

Figure 5.9. An adjacency list for our example graph.

Of course, linked lists are not the only data structure we could use; any other structure that supports searching, listing, insertion, and deletion will do. For example, we can reduce the time to determine whether uv is an edge to $O(1+\log(\deg(u)))$ by using a balanced binary search tree to store the neighbors of u, or even to $O(1)$ time by using an appropriately constructed hash table.[7]

One common implementation of adjacency lists is the ***adjacency array***, which uses a single array to store all edge records, with the records of edges incident to each vertex in a contiguous interval, and with a separate array storing the index of the first edge incident to each vertex. Moreover, it is useful to keep the intervals for each vertex in sorted order, as shown in Figure 5.10, so that we can check in $O(\log \deg(u))$ time whether two vertices u and v are adjacent.

Adjacency Matrices

The other standard data structure for graphs is the ***adjacency matrix***,[8] first proposed by Georges Brunel in 1894. The adjacency matrix of a graph G is a $V \times V$ matrix of 0s and 1s, normally represented by a two-dimensional array $A[1..V, 1..V]$, where each entry indicates whether a particular edge is present in G. Specifically, for all vertices u and v:

- if the graph is undirected, then $A[u,v] := 1$ if and only if $uv \in E$, and

[7]This is a *lot* more subtle than it sounds. Most popular hashing techniques do *not* guarantee fast query times, and even most *good* hashing methods can guarantee only $O(1)$ *expected* time. See http://algorithms.wtf for a more thorough discussion of hashing.

[8]See footnote 3.

Figure 5.10. An abstract adjacency array for our example graph, and its actual implementation as a pair of integer arrays.

- if the graph is directed, then $A[u,v] := 1$ if and only if $u \to v \in E$.

For undirected graphs, the adjacency matrix is always *symmetric*, meaning $A[u,v] = A[v,u]$ for all vertices u and v, because uv and vu are just different names for the same edge, and the diagonal entries $A[u,u]$ are all zeros. For directed graphs, the adjacency matrix may or may not be symmetric, and the diagonal entries may or may not be zero.

	a	b	c	d	e	f	g	h	i	j	k	l	m
a	0	1	0	0	1	0	0	0	0	0	0	0	0
b	1	0	1	0	1	1	0	0	0	0	0	0	0
c	0	1	0	1	0	1	1	0	0	0	0	0	0
d	0	0	1	0	0	0	1	0	0	0	0	0	0
e	1	1	0	0	0	1	1	1	0	0	0	0	0
f	0	1	1	0	1	0	1	0	0	0	0	0	0
g	0	0	1	1	1	1	0	0	1	0	0	0	0
h	0	0	0	0	1	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	1	0	0	0	0	0	0
j	0	0	0	0	0	0	0	0	0	0	1	1	1
k	0	0	0	0	0	0	0	0	0	1	0	1	0
l	0	0	0	0	0	0	0	0	0	1	1	0	1
m	0	0	0	0	0	0	0	0	0	1	0	1	0

Figure 5.11. An adjacency matrix for our example graph.

Given an adjacency matrix, we can decide in $\Theta(1)$ time whether two vertices are connected by an edge just by looking in the appropriate slot in the matrix. We can also list all the neighbors of a vertex in $\Theta(V)$ time by scanning the corresponding row (or column). This running time is optimal in the worst case, but even if a vertex has few neighbors, we still have to scan the entire row to find them all. Similarly, adjacency matrices require $\Theta(V^2)$ space, regardless of how many edges the graph actually has, so they are only space-efficient for very *dense* graphs.

Comparison

Table 5.1 summarizes the performance of the various standard graph data structures. Stars* indicate expected amortized time bounds for maintaining dynamic hash tables.[9]

	Standard adjacency list (linked lists)	Fast adjacency list (hash tables)	Adjacency matrix
Space	$\Theta(V+E)$	$\Theta(V+E)$	$\Theta(V^2)$
Test if $uv \in E$	$O(1+\min\{\deg(u),\deg(v)\}) = O(V)$	$O(1)$	$O(1)$
Test if $u \to v \in E$	$O(1+\deg(u)) = O(V)$	$O(1)$	$O(1)$
List v's (out-)neighbors	$\Theta(1+\deg(v)) = O(V)$	$\Theta(1+\deg(v)) = O(V)$	$\Theta(V)$
List all edges	$\Theta(V+E)$	$\Theta(V+E)$	$\Theta(V^2)$
Insert edge uv	$O(1)$	$O(1)^*$	$O(1)$
Delete edge uv	$O(\deg(u)+\deg(v)) = O(V)$	$O(1)^*$	$O(1)$

Table 5.1. Times for basic operations on standard graph data structures.

In light of this comparison, one might reasonably wonder why anyone would *ever* use an adjacency matrix; after all, adjacency lists with hash tables support the same operations in the same time, using less space. The main reason is that for sufficiently dense graphs, adjacency matrices are simpler and more efficient in practice, because they avoid the overhead of chasing pointers and computing hash functions; they're just contiguous blocks of memory.

Similarly, why would anyone use *linked lists* in an adjacency list structure to store neighbors, instead of balanced binary search trees or hash tables? Although the primary reason in practice is almost surely *tradition*—If they were good enough for Donald Knuth's code, they should be good enough for yours!—there are more principled arguments. One is that standard adjacency lists are in fact *good enough* for most applications. Most standard graph algorithms never (or rarely) actually ask whether an arbitrary edge is present or absent, or attempt to insert or delete edges, and so optimizing the data structures to support those operations is unnecessary.

But in my opinion, the most compelling reason for *both* standard data structures is that many graphs are *implicitly* represented by adjacency matrices and standard adjacency lists. For example:

- Intersection graphs are usually represented as a list of the underlying geometric objects. As long as we can test whether two objects intersect in constant time, we can apply any graph algorithm to an intersection graph by *pretending* that the input graph is stored explicitly as an adjacency matrix.
- Any data structure composed from records with pointers between them can be seen as a directed graph. Graph algorithms can be applied to these data structures by *pretending* that the graph is stored in a standard adjacency list.

[9] Don't worry if you don't understand the phrase "expected amortized".

- Similarly, we can apply any graph algorithm to a configuration graph *as though* it were represented as a standard adjacency list, provided we can enumerate all possible moves from a given configuration in constant time each.

For the last two examples, we can enumerate the edges leaving any vertex in time proportional to its degree, but we *cannot* necessarily determine in constant time if two vertices are adjacent. (Is there a pointer from this record to that record? Can we get from this configuration to that configuration in one move?) Moreover, we usually don't have the luxury of reorganizing the pointers in each record or the moves out of a given configuration into a more efficient data structure. Thus, a standard adjacency list, with neighbors stored in linked lists, is the appropriate model data structure.

In the rest of this book, unless explicitly stated otherwise, all time bounds for graph algorithms assume that the input graph is represented by a standard adjacency list. Similarly, unless explicitly stated otherwise, when an exercise asks you to design and analyze a graph algorithm, you should assume that the input graph is represented in a standard adjacency list.

5.5 Whatever-First Search

So far we have only discussed *local* operations on graphs; arguably the most fundamental *global* question we can ask about graphs is ***reachability***. Given a graph G and a vertex s in G, the reachability question asks which vertices are reachable from s; that is, for which vertices v is there a path from s to v? For now, let's consider only undirected graphs; I'll consider directed graphs briefly at the end of this section. For undirected graphs, the vertices reachable from s are precisely the vertices in the same component as s.

Perhaps the most natural reachability algorithm—at least for people like us who are used to thinking recursively—is ***depth-first search***. This algorithm can be written either recursively or iteratively. It's exactly the same algorithm either way; the only difference is that we can actually see the "recursion" stack in the non-recursive version.

```
RecursiveDFS(v):
  if v is unmarked
      mark v
      for each edge vw
          RecursiveDFS(w)
```

```
IterativeDFS(s):
  Push(s)
  while the stack is not empty
      v ← Pop
      if v is unmarked
          mark v
          for each edge vw
              Push(w)
```

Depth-first search is just one (perhaps the most common) species of a general family of graph traversal algorithms that I call ***whatever-first search***. The generic traversal algorithm stores a set of candidate edges in some data structure that I'll call a "bag". The only important properties of a "bag" are that we can put stuff into it and then later take stuff back out. A stack is a particular type of bag, but certainly not the only one. Here is the generic algorithm:

```
WhateverFirstSearch(s):
  put s into the bag
  while the bag is not empty
      take v from the bag
      if v is unmarked
          mark v
          for each edge vw
              put w into the bag
```

I claim that WhateverFirstSearch marks every node reachable from s and nothing else. The algorithm clearly marks each vertex in G *at most* once. To show that it visits every node in a connected graph *at least* once, we modify the algorithm slightly; the modifications are in bold red. Instead of keeping vertices in the bag, the modified algorithm stores pairs of vertices. This modification allows us to remember, whenever we visit a vertex v for the first time, which previously-visited neighbor vertex put v into the bag. We call this earlier vertex the *parent* of v.

```
WhateverFirstSearch(s):
  put (∅, s) in bag
  while the bag is not empty
      take (p, v) from the bag                (⋆)
      if v is unmarked
          mark v
          parent(v) ← p
          for each edge vw                    (†)
              put (v, w) into the bag         (⋆⋆)
```

Lemma 5.1. *WhateverFirstSearch(s) marks every vertex reachable from s and only those vertices. Moreover, the set of all pairs $(v, parent(v))$ with $parent(v) \neq \varnothing$ defines a spanning tree of the component containing s.*

Proof: First we argue that the algorithm marks every vertex v that is reachable from s, by induction on the shortest-path distance from s to v. The algorithm marks s. Let v be any other vertex reachable from s, and let $s \to \cdots \to u \to v$ be any path from s to v with the minimum number of edges. (There must be such a path, because v is reachable from s.) The prefix path $s \to \cdots \to u$ is shorter than the shortest path from s to u, so the inductive hypothesis implies that the

algorithm marks u. When the algorithm marks u, it must immediately put the pair (u, v) into the bag, so it must later take (u, v) out of the bag, at which point the algorithm immediately marks v, unless it was already marked.

Every pair $(v, parent(v))$ with $parent(v) \neq \varnothing$ is actually an edge in the underlying graph G. We claim that for any marked vertex v, the path of parent edges $v \to parent(v) \to parent(parent(v)) \to \cdots$ eventually leads back to s; we prove this claim by induction on the order in which vertices are marked. Trivially s is reachable from s, so let v be any other marked vertex. The parent of v must be marked before v is marked, so the inductive hypothesis implies that the parent path $parent(v) \to parent(parent(v)) \to \cdots$ leads to s; adding one more parent edge $s \to parent(s)$ establishes the claim.

The previous claim implies that every vertex marked by the algorithm is reachable from s, and that the set of all parent edges forms a connected graph. Because every marked node except s has a unique parent, the number of parent edges is exactly one less than the number of marked vertices. We conclude that the parent edges form a tree. □

Analysis

The running time of the traversal algorithm depends on what data structure we use for the "bag", but we can make a few general observations. Let T is the time required to insert a single item into the bag or delete a single item from the bag. The for loop (†) is executed exactly once for each marked vertex, and therefore at most V times. Each edge uv in the component of s is put into the bag exactly twice; once as the pair (u, v) and once as the pair (v, u), so line (⋆⋆) is executed at most $2E$ times. Finally, we can't take more things out of the bag than we put in, so line (⋆) is executed at most $2E + 1$ times. Thus, assuming the underlying graph G is stored in a standard adjacency list, WhateverFirstSearch runs in $\boldsymbol{O(V + ET)}$ ***time***. (If G is stored in an adjacency matrix, the running time of WhateverFirstSearch increases to $O(V^2 + ET)$.)

5.6 Important Variants

Stack: Depth-First

If we implement the "bag" using a *stack*, we recover our original depth-first search algorithm. Stacks support insertions (push) and deletions (pop) in $O(1)$ time each, so the algorithm runs in $\boldsymbol{O(V + E)}$ ***time***. The spanning tree formed by the parent edges is called a ***depth-first spanning tree***. The exact shape of the tree depends on the start vertex and on the order that neighbors are visited inside the for loop (†), but in general, depth-first spanning trees are long

and skinny. We will consider several important properties and applications of depth-first search in Chapter 6.

Queue: Breadth-First

If we implement the "bag" using a *queue*, we get a different graph-traversal algorithm called ***breadth-first search***. Queues support insertions (push) and deletions (pull) in $O(1)$ time each, so the algorithm runs in $\boldsymbol{O(V+E)}$ ***time***. In this case, the ***breadth-first spanning tree*** formed by the parent edges contains ***shortest paths*** from the start vertex s to every other vertex in its component; we will consider shortest paths in detail in Chapter 8. Again, the exact shape of a breadth-first spanning tree depends on the start vertex and on the order that neighbors are visited in the for loop (†), but in general, breadth-first spanning trees are short and bushy.

Figure 5.12. A depth-first spanning tree and a breadth-first spanning tree of the same graph, both starting at the center vertex.

Priority Queue: Best-First

Finally, if we implement the "bag" using a *priority queue*, we get yet another family of algorithms called ***best-first search***. Because the priority queue stores at most one copy of each edge, inserting an edge or extracting the minimum-priority edge requires $O(\log E)$ time, which implies that best-first search runs in $\boldsymbol{O(V+E\log E)}$ ***time***.

I describe best-first search as a "family of algorithms", rather than a single algorithm, because there are different methods to assign priorities to the edges, and these choices lead to different algorithmic behavior. I'll describe three well-known variants below, but there are many others. In all three examples, we assume that every edge uv or $u{\to}v$ in the input graph has a non-negative weight $w(uv)$ or $w(u{\to}v)$.

First, if the input graph is undirected and we use the weight of each edge as its priority, best-first search constructs the ***minimum spanning tree*** of the component of s. Surprisingly, as long as all the edge weights are distinct, the resulting tree does *not* depend on the start vertex or the order that neighbors

are visited; in this case, the minimum spanning tree is actually unique. This instantiation of best-first search is commonly (but, as usual, incorrectly) known as *Prim's algorithm*; we'll discuss this and other minimum-spanning-trees in more detail in Chapter 7.

Define the *length* of a path to be the sum of the weights of its edges. We can also compute ***shortest paths*** in weighted graphs using best-first search, as follows. Every marked vertex v stores a distance $dist(v)$. Initially we set $dist(s) = 0$. For every other vertex v, when we set $parent(v) \leftarrow p$, we also set $dist(v) \leftarrow dist(p) + w(p \to v)$, and when we insert the edge $v \to w$ into the priority queue, we use the priority $dist(v) + w(v \to w)$. Assuming all edge weights are positive, $dist(v)$ is the length of the shortest path from s to v. This instantiation of best-first search is commonly (but, as usual, strictly speaking, incorrectly) known as *Dijkstra's algorithm*; we'll see this algorithm again in Chapter 8.

Finally, define the *width* of a path to be the *minimum* weight of any edge in the path. A simple modification of "Dijkstra's" best-first search algorithm computes ***widest paths*** from s to every other reachable vertex; widest paths are also called ***bottleneck shortest paths***. Every marked vertex v stores a value $width(v)$. Initially we set $width(s) = \infty$. For every other vertex v, when we set $parent(v) \leftarrow p$, we also set $width(v) \leftarrow \min\{width(p), w(p \to v)\}$, and when we insert the edge $v \to w$ into the priority queue, we use the priority $\min\{width(v), w(v \to w)\}$. Widest paths are useful in algorithms for computing *maximum flows*, which (you guessed it) we'll consider in Chapter 10.

Disconnected Graphs

WhateverFirstSearch(s) only visits the vertices reachable from a single start vertex s. To visit *every* vertex in G, we can use the following simple "wrapper" function.

```
WFSAll(G):
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          WhateverFirstSearch(v)
```

Wait, I hear you ask, why are you making this so complicated? Why not just[10] scan the vertex array?

```
MarkEveryVertexDuh(G):
  for all vertices v
      mark v
```

[10]This word is almost always a signal that you are missing something important.

Well, sure, *if* you have an complete list of vertices, then you can do that, but remember that not all graphs are represented so explicitly.[11] More importantly, even if we do have an explicit vertex list, the order in which this naive algorithm visits vertices is determined by their order in the *data structure*, not by the abstract structure of the graph.

In particular, unlike a naive scan through the vertices, WFSAll visits all the vertices in one component, and then all the vertices in the next component, and so on through each component of the input graph. This component-by-component traversal allows us, for example, to count the components of a disconnected graph using a single counter.

```
CountComponents(G):
  count ← 0
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          count ← count + 1
          WhateverFirstSearch(v)
  return count
```

With just a bit more work, we can record which component contains each vertex, instead of merely marking it.

```
CountAndLabel(G):
  count ← 0
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          count ← count + 1
          LabelOne(v, count)
  return count
```

```
⟨⟨Label one component⟩⟩
LabelOne(v, count):
  while the bag is not empty
      take v from the bag
      if v is unmarked
          mark v
          comp(v) ← count
          for each edge vw
              put w into the bag
```

WFSAll marks every vertex once, puts every edge into the bag once, and takes every edge out of the bag once, so the overall running time is $O(V + ET)$, where T is the time for a bag operation. In particular, if we run depth-first search or breadth-first search at every vertex, the resulting algorithm still requires only $O(V + E)$ time.

Moreover, because WhateverFirstSearch computes a spanning tree of one component, we can use WFSAll to compute a spanning *forest* of the entire

[11] On the other hand, if we store a time-stamp at every vertex indicating the last time it was "marked", then we can "unmark every vertex" in $O(1)$ time by recording the start time of our traversal, and considering a vertex "marked" if its time stamp is later than the recorded start time.

graph. In particular, best-first search with edge weights as priorities computes the minimum-weight spanning forest in $O(V + E \log E)$.

Shockingly, at least one *extremely* popular algorithms textbook claims that this wrapper can only be used with depth-first search.[12] This claim is flatly incorrect. In fact, the *very first* implementation of breadth-first search, written around 1945 by Konrad Zuse in his proto-language *Plankalkül*, was developed for the specific purpose of counting and labeling the components of an undirected graph.

Directed Graphs

Whatever-first search is easy to adapt to directed graphs; the only difference is that when we mark a vertex, we put all of its *out*-neighbors into the bag. In fact, if we are using standard adjacency lists or adjacency matrices, we do not have to change the code at all!

```
WhateverFirstSearch(s):
  put s into the bag
  while the bag is not empty
      take v from the bag
      if v is unmarked
          mark v
          for each edge v→w
              put w into the bag
```

Our earlier proof implies that the algorithm marks every vertex reachable from s, and the directed edges $parent(v) \to p$ define a rooted tree, with all edges directed away from the root s. However, even if the graph is connected, we no longer necessarily obtain a *spanning* tree of the graph, because reachability is no longer symmetric.

On the gripping hand, WhateverFirstSearch does define a spanning tree of the vertices reachable from s. Moreover, by varying the instantiation of the "bag", we can obtain a depth-first spanning tree, a breadth-first spanning tree, a minimum-weight directed spanning tree, a shortest-path tree, or a widest-path tree of those reachable vertices.

5.7 Graph Reductions: Flood Fill

One of the earliest modern examples of whatever-first search was proposed by Edward Moore in the mid-1950s. A *pixel map* is a two-dimensional array

[12] To quote directly: "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."

whose value represent colors; the individual entries in the array are called *pixels*, an abbreviation of *picture elements*.[13] A *connected region* in a pixel map is a connected subset of pixels that all have the same color, where two pixels are considered adjacent if they are immediate horizontal or vertical neighbors. The *flood fill* operation, commonly represented by a paint can in raster-graphics editing software, changes every pixel in a connected region to a new color; the input to the operation consists of the indices i and j of one pixel in the target region and the new color.

Figure 5.13. An example of flood fill

The flood-fill problem can be reduced to the reachability problem by chasing the definitions. We define an undirected graph $G = (V, E)$, whose vertices are the individual pixels, and whose edges connect neighboring pixels with the same color. Each connected region in the pixel map is a component of G; thus, the flood-fill problem ***reduces*** to a reachability problem in G. We can solve this reachability problem using whatever-first search in G, starting at the given pixel (i, j), with one minor modification; whenever we mark a vertex, we immediately change its color. For an $n \times n$ pixel map, the graph G has n^2 vertices and at most $2n^2$ edges, so whatever-first search runs in $O(V + E) = \boldsymbol{O(n^2)}$ ***time***.

This simple example demonstrates the essential ingredients of a ***reduction***. Rather than solving the flood-fill problem from scratch, we use an existing algorithm as a black-box subroutine. *How* whatever-first search works is utterly irrelevant here; all that matters is its *specification*: Given a graph G and a starting vertex s, mark every vertex in G that is reachable from s. Like any other subroutine, we still have to describe how to construct the input and how to use its output. We also have to analyze *our* resulting algorithm in terms of *our* input parameters, not the vertices and edges of whatever intermediate graph our algorithm constructs.

Now that we have an algorithm that works—but *only* now—we can apply two easy optimizations to make it faster, one practical and the other theoretical:

[13]Before the advent of modern raster display devices in the 1960s, pixels were more commonly known as *stitches* or *tesserae*, depending on whether they were made of thread or very small rocks. The word *pix* became a standard abbreviation for *picture(s)* in the early 20th century—not long after *sox* became a common plural of *sock*—supplanting the earlier colloquialism *piccy*. See also voxel (volume element), texel (texture element), and taxel (tactile element and/or badger).

- In an actual implementation, we would not actually build a separate graph data structure for G. Instead, we can use the pixel map directly *as though* it were a standard adjacency list, because we can list the same-color neighbors of any pixel in $O(1)$ time each. In particular, there is no need to separately "mark" vertices; we can use the color of the pixels instead.
- More careful analysis implies that the running time is proportional to the number of pixels in the region being filled—equivalently, the number of vertices in component of G containing vertex (i, j)—which could be considerably smaller than $O(n^2)$.

Exercises

Graphs

1. Prove that the following definitions are all equivalent.
   - A tree is a connected acyclic graph.
   - A tree is one component of a forest. (A forest is an acyclic graph.)
   - A tree is a connected graph with *at most* $V - 1$ edges.
   - A tree is a minimally connected graph; removing any edge disconnects the graph.
   - A tree is an acyclic graph with *at least* $V - 1$ edges.
   - A tree is a maximally acyclic graph; adding an edge between any two vertices creates a cycle.
   - A tree is a graph that contains a unique path between each pair of vertices.

2. Prove that any connected acyclic graph with $n \geq 2$ vertices has at least two vertices with degree 1. Do not use the words "tree" or "leaf", or any well-known properties of trees; your proof should follow entirely from the definitions of "connected" and "acyclic".

3. A graph (V, E) is *bipartite* if the vertices V can be partitioned into two subsets L and R, such that every edge has one vertex in L and the other in R.
   (a) Prove that every tree is a bipartite graph.
   (b) Prove that a graph G is bipartite if and only if every cycle in G has an even number of edges.
   (c) Describe and analyze an efficient algorithm that determines whether a given undirected graph is bipartite.

4. Whenever groups of pigeons gather, they instinctively establish a *pecking order*. For any pair of pigeons, one pigeon always pecks the other, driving it away from food or potential mates. The same pair of pigeons always chooses the same pecking order, even after years of separation, no matter what other pigeons are around. Surprisingly, the overall pecking order can contain cycles—for example, pigeon i pecks pigeon j, which pecks pigeon k, which pecks pigeon ℓ, which pecks pigeon i.

   (a) Prove that any finite population of pigeons can be placed in a procession (perhaps a parade?) so that each pigeon pecks the preceding pigeon's posterior. Pretty please.

   (b) Suppose you are given a directed graph representing the pecking relationships among a set of n pigeons. The graph contains one vertex per pigeon, and it contains an edge $i \to j$ if and only if pigeon i pecks pigeon j. Describe and analyze an algorithm to compute a pecking order for the pigeons, as guaranteed by part (a).

   (c) Prove that for any set of at least three pigeons, either the pecking order described in part (a) is unique, or there are three pigeons i, j, and k, such that pigeon i pecks pigeon j, which pecks pigeon k, which pecks pigeon i.

5. An ***Euler tour*** of a graph G is a closed walk through G that traverses every edge of G exactly once.

   (a) Prove that if a connected graph G has an Euler tour, then every vertex in G has even degree. (Euler proved this.)

   (b) Prove that if every vertex in a connected graph G has even degree, then G has an Euler tour. (Euler did *not* prove this.)

   (c) Describe and analyze an algorithm to compute an Euler tour in a given graph, or correctly report that no such tour exists. (Euler vaguely waved his hands at this.)

6. The d-dimensional hypercube is the graph defined as follows. There are 2^d vertices, each labeled with a different string of d bits. Two vertices are joined by an edge if their labels differ in exactly one bit.

   (a) A Hamiltonian cycle in a graph G is a cycle of edges in G that enters each vertex of G exactly once. Prove that for all $d \geq 2$, the d-dimensional hypercube has a Hamiltonian cycle.

   (b) Which hypercubes have an Euler tour (a closed walk that traverses every edge exactly once)? *[Hint: This is very easy.]*

Traversal Algorithms

7. Recall that a directed graph G is *strongly connected* if, for any two vertices u and v, there is a path in G from u to v and a path in G from v to u.

   Describe an algorithm to determine, given an *undirected* graph G as input, whether it is possible to direct each edge of G so that the resulting directed graph is strongly connected.

8. Let G be a connected graph, and let T be a depth-first spanning tree of G rooted at some node v. Prove that if T is also a breadth-first spanning tree of G rooted at v, then $G = T$.

9. Professors Epprich and Goodstein propose the following optimization of the generic whatever-first search algorithm. Instead of checking whether the vertices we take out of the bag are marked, their algorithm checks before it even puts the vertex into the bag, thereby ensuring that each vertex is put into the bag at most once. Their algorithm also assigns the parent of each vertex when that vertex is marked.

```
EagerWFS(s):
  mark s
  put s into the bag
  while the bag is not empty
      take v from the bag
      for each edge vw
          if w is unmarked
              mark w
              parent(w) ← v
              put w into the bag
```

   (a) Prove that EagerWFS(s) marks every node reachable from s and nothing else. Equivalently, prove that the parent edges $v \to parent(v)$ computed by EagerWFS(s) define a spanning tree of the component containing s.

   (b) Prove that if the bag is implemented as a queue, EagerWFS is equivalent to breadth-first search, meaning the two algorithms mark the same vertices in the same order and construct the same spanning tree. *[Hint: What is the definition of a queue?]*

   (c) Prove that EagerWFS is *never* equivalent to depth-first search, no matter what data structure is used as the bag (and thus, in particular, when the bag is a stack).

   Neither EagerWFS nor RecursiveDFS specify the order that edges vw at each vertex v are considered, and different edge orders may lead to different spanning trees. Thus, you need to argue, for some explicit graph G, that no spanning tree of G produced by RecursiveDFS can be constructed by EagerWFS (using *any* bag data structure), or vice versa.

10. One of the earliest published descriptions of whatever-first search as a generic class of algorithms was by Edsger Dijkstra, Leslie Lamport, Alain Martin, Carel Scholten, and Elisabeth Steffens in 1975, as part of the design of an automatic garbage collector. Instead of maintaining marked and unmarked vertices, their algorithm maintains a color for each vertex, which is either white, gray, or black. As usual, in the following algorithm, we imagine a fixed underlying graph G.

```
ThreeColorSearch(s):
  color all nodes white
  color s gray
  while at least one vertex is gray
      ThreeColorStep( )
```

```
ThreeColorStep( ):
  v ← any gray vertex
  if v has no white neighbors
      color v black
  else
      w ← any white neighbor of v
      parent(w) ← v
      color w gray
```

(a) Prove that ThreeColorSearch maintains the following invariant at all times: No black vertex is a neighbor of a white vertex. *[Hint: This should be easy.]*

(b) Prove that after ThreeColorSearch(s) terminates, all vertices reachable from s are black, all vertices not reachable from s are white, and that the parent edges $v \to parent(v)$ define a rooted spanning tree of the component containing s.

[Hint: Intuitively, black nodes are "marked" and gray nodes are "in the bag". Unlike our formulation of WhateverFirstSearch, however, the three-color algorithm is not required to process all edges out of a node at the same time.]

(c) Prove that the following variant of ThreeColorSearch, which maintains the set of gray vertices in a standard stack, is equivalent to depth-first search. *[Hint: The order of the last two lines of ThreeColorStackStep matters!]*

```
ThreeColorStackSearch(s):
  color all nodes white
  color s gray
  push s onto the stack
  while at least one vertex is gray
      ThreeColorStackStep( )
```

```
ThreeColorStackStep( ):
  pop v from the stack
  if v has no white neighbors
      color v black
  else
      w ← any white neighbor of v
      parent(w) ← v
      color w gray
      push v onto the stack
      push w onto the stack
```

(d) Prove that the following variant of ThreeColorSearch, which maintains the set of gray vertices in a standard queue, is ***not*** equivalent

to breadth-first search. *[Hint: The order of the last two lines of* ThreeColorQueueStep *doesn't matter!]*

```
ThreeColorQueueSearch(s):
    color all nodes white
    color s gray
    push s into the queue
    while at least one vertex is gray
        ThreeColorQueueStep( )
```

```
ThreeColorQueueStep( ):
    pull v from the queue
    if v has no white neighbors
        color v black
    else
        w ← any white neighbor of v
        parent(w) ← v
        color w gray
        push v into the queue
        push w into the queue
```

♥(e) Now suppose that another process is adding edges to G while ThreeColorSearch is running. These new edges could violate the color invariant described in part (a) and therefore destroy the correctness of the algorithm—in particular, when ThreeColorSearch terminates, some vertices reachable from s could be white. This would be disastrous if we are relying on "white" to mean "unreachable and therefore safe to delete".

However, if the other process explicitly preserves the color invariant, we can still use the three-color algorithm to safely identify unreachable vertices. We model the two concurrent algorithms as follows; the either/or choice in GarbageCollect and the choice of which vertices u and w to Mutate are entirely out of the main algorithm's control.[14]

```
GarbageCollect(s):
    color all vertices white
    color s gray
    while at least one vertex is gray
        either
            CollectStep( )
        or
            Mutate( )
```

```
CollectStep( ):
    v ← any gray vertex
    if v has no white neighbors
        color v black
    else
        w ← any white neighbor of v
        color w gray
```

[14]This is a *dramatic* oversimplification of the "mark and sweep" garbage-collection algorithms actually used in multi-threaded languages like Lua and Go. A more thorough discussion of multi-threaded dynamic memory management is unfortunately beyond the scope of this book, except for the First Commandment: **Thou Shalt Not Roll Thine Own Garbage Collector.**

```
MUTATE():
  u ← any vertex
  w ← any vertex
  if uw is not an edge
      add edge uw
      if u is black and w is white
          color u gray
      if u is white and w is black
          color w gray
```

Prove that GarbageCollect eventually terminates with every vertex reachable from s colored black and every vertex not reachable from s colored white.

♥(f) Suppose instead of recoloring black vertices gray, Mutate maintains the color invariant by coloring some *white* vertices gray:

```
MUTATE():
  u ← any vertex
  w ← any vertex
  if uw is not an edge
      add edge uw
      if u is black and w is white
          color w gray
      if u is white and w is black
          color u gray
```

Prove that GarbageCollect eventually terminates with s colored black, every vertex reachable from a black vertex colored black, and every vertex not reachable from a black vertex colored white.

Reductions

11. A ***number maze*** is an $n \times n$ grid of positive integers. A token starts in the upper left corner; your goal is to move the token to the lower-right corner. On each turn, you are allowed to move the token up, down, left, or right; the distance you may move the token is determined by the number on its current square. For example, if the token is on a square labeled 3, then you may move the token three steps up, three steps down, three steps left, or three steps right. However, you are never allowed to move the token off the edge of the board.

    Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given number maze, or correctly reports that the maze has no solution. For example, given the number maze in Figure 5.14, your algorithm should return the integer 8.

12. ***Snakes and Ladders*** is a classic board game, originating in India no later than the 16th century. The board consists of an $n \times n$ grid of squares,

Figure 5.14. A 5 × 5 number maze that can be solved in eight moves.

numbered consecutively from 1 to n^2, starting in the bottom left corner and proceeding row by row from bottom to top, with rows alternating to the left and right. Certain pairs of squares in this grid, always in different rows, are connected by either "snakes" (leading down) or "ladders" (leading up). Each square can be an endpoint of at most one snake or ladder.

You start with a token in cell 1, in the bottom left corner. In each move, you advance your token up to k positions, for some fixed constant k. If the token ends the move at the *top* end of a snake, it slides down to the bottom of that snake. Similarly, if the token ends the move at the *bottom* end of a ladder, it climbs up to the top of that ladder.

Describe and analyze an algorithm to compute the smallest number of moves required for the token to reach the last square of the grid.

Figure 5.15. A Snakes and Ladders board. Upward straight arrows are ladders; downward wavy arrows are snakes.

13. The infamous Mongolian puzzle-warrior Vidrach Itky Leda invented the following puzzle in the year 1473. The puzzle consists of an $n \times n$ grid of squares, where each square is labeled with a positive integer, and two tokens, one red and the other blue. The tokens always lie on distinct squares of the grid. The tokens start in the top left and bottom right corners of the grid; the goal of the puzzle is to swap the tokens.

In a single turn, you may move either token up, right, down, or left *by a distance determined by the* ***other*** *token*. For example, if the red token is on a square labeled 3, then you may move the blue token 3 steps up, 3 steps left, 3 steps right, or 3 steps down. However, you may not move either token off the grid, and at the end of a move the two tokens cannot lie on the same square.

Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given Vidrach Itky Leda puzzle, or correctly reports that the puzzle has no solution. For example, given the puzzle in Figure 5.16, your algorithm would return the number 5.

Figure 5.16. A five-move solution for a 4 × 4 Vidrach Itky Leda puzzle.

14. Suppose you are given a directed graph $G = (V, E)$ and two vertices s and t. Describe and analyze an algorithm to determine if there is a walk in G from s to t (possibly repeating vertices and/or edges) whose length is divisible by 3.

For example, given the graph shown below, with the indicated vertices s and t, your algorithm should return True, because the walk $s \to w \to y \to x \to s \to w \to t$ has length 6.

15. Suppose you are given a directed graph G where some edges are red and the remaining edges are blue. Describe an algorithm to find the shortest walk in G from one vertex s to another vertex t in which no three consecutive edges have the same color. That is, if the walk contains two red edges in a row, the next edge must be blue, and if the walk contains two blue edges in a row, the next edge must be red.

For example, given the following graph as input, your algorithm should return the integer 7, because $s \to a \to b \Rightarrow d \to c \Rightarrow a \to b \to t$ is the shortest legal walk from s to t.

16. Consider a directed graph G, where each edge is colored either red, white, or blue. A walk in G is called a *French flag walk* if its sequence of edge colors is red, white, blue, red, white, blue, and so on. More formally, a walk $v_0 \to v_1 \to \cdots \to v_k$ is a French flag walk if, for every integer i, the edge $v_i \to v_{i+1}$ is red if $i \bmod 3 = 0$, white if $i \bmod 3 = 1$, and blue if $i \bmod 3 = 2$.

    Describe an algorithm to find all vertices in G that can be reached from a given vertex v through a French flag walk.

17. There are n galaxies connected by m intergalactic teleport-ways. Each teleport-way joins two galaxies and can be traversed in both directions. Also, each teleport-way e has an associated cost of $c(e)$ dollars, where $c(e)$ is a positive integer. A teleport-way can be used multiple times, but the toll must be paid every time it is used.

    Judy wants to travel from galaxy s to galaxy t, but teleportation is not very pleasant and she would like to minimize the number of times she needs to teleport. However, she wants the total cost to be a multiple of five dollars, because carrying small change is not pleasant either.

    (a) Describe and analyze an algorithm to compute the smallest number of times Judy needs to teleport to travel from galaxy s to galaxy t so that the total cost is a multiple of five dollars.

    (b) Solve part (a), but now assume that Judy has a coupon that allows her to use exactly one teleport-way for free.

18. Three Seashells is a solitaire game, played on a connected undirected graph G. Initially, three tokens are placed on distinct start vertices a, b, c. In each turn, you *must* move *all three* tokens, by moving each token along an edge from its current vertex to an adjacent vertex. At the end of each turn, the three tokens *must* lie on three different vertices. Your goal is to move the tokens onto three goal vertices x, y, z; it does not matter which token ends up on which goal vertex.

Figure 5.17. The initial configuration of the Three Seashells puzzle and the first two turns of a solution.

Describe and analyze an algorithm to determine whether a given Three Seashells puzzle is solvable. Your input consists of the graph G, the start vertices a, b, c, and the goal vertices x, y, z. Your output is a single bit: TRUE or FALSE.

19. Let G be a connected undirected graph. Suppose we start with two coins on two arbitrarily chosen vertices of G, and we want to move the coins so that they lie on the same vertex using as few moves as possible. At every step, each coin *must* move to an adjacent vertex.

    (a) Describe and analyze an algorithm to compute the minimum number of steps to reach a configuration where both coins are on the same vertex, or to report correctly that no such configuration is reachable. The input to your algorithm consists of a graph $G = (V, E)$ and two vertices $u, v \in V$ (which may or may not be distinct).

    (b) Now suppose there are three coins. Describe and analyze an algorithm to compute the minimum number of steps to reach a configuration where both coins are on the same vertex, or to report correctly that no such configuration is reachable.

    (c) Finally, suppose there are *forty-two* coins. Describe and analyze an algorithm to determine whether it is possible to move all 42 coins to the same vertex. Again, *every* coin must move at *every* step. For full credit, your algorithm should run in $O(V + E)$ time.

20. One of my daughter's elementary-school math workbooks[15] contains several puzzles of the following type:

    Complete each angle maze below by tracing a path from start to finish that has only acute angles.

    Describe and analyze an algorithm to solve arbitrary acute-angle mazes.

    You are given a connected undirected graph G, whose vertices are points in the plane and whose edges are line segments. Edges do not intersect, except at their endpoints. For example, a drawing of the letter X would have five vertices and four edges, and the first maze above has 18 vertices and 21 edges. You are also given two vertices Start and Finish.

    Your algorithm should return True if G contains a walk from Start to Finish that has only acute angles, and False otherwise. Formally, a walk through G is valid if, for any two consecutive edges $u \to v \to w$ in the walk, either $\angle uvw = \pi$ or $0 < \angle uvw < \pi/2$. Assume you have a subroutine that can determine in $O(1)$ time whether the angle between two given segments is straight, obtuse, right, or acute.

[15] Jason Batterson and Shannon Rogers, *Beast Academy Math: Practice 3A*, 2012. See https://www.beastacademy.com/resources/printables.php for several more examples.

21. Suppose you are given a set of n horizontal and vertical line segments and two points s and t in the plane. Describe an efficient algorithm to determine if there is a path from s to t that does not intersect any of the given line segments.

    Each horizontal line segment is specified by its left and right x-coordinates and its unique y-coordinate; similarly, each vertical line segment is specified by its unique x-coordinate and its top and bottom y-coordinates. Finally, the points s and t are each specified by their x- and y-coordinates.

Figure 5.18. A path between two points in a maze of horizontal and vertical line segments.

22. Every cheesy romance movie has a scene where the romantic couple, after a long and frustrating separation, suddenly see each other across a long distance, and then slowly approach one another with unwavering eye contact as the music rolls in and the rain lifts and the sun shines through the clouds and the music swells and everyone starts dancing with rainbows and kittens and chocolate unicorns and. . . .[16]

    Suppose a romantic couple—in grand computer science tradition, named Alice and Bob—enters their favorite park at the east and west entrances and immediately establish eye-contact. They can't just run directly to each other; instead, they must stay on the path that zig-zags through the park between the east and west entrances. To maintain the proper dramatic tension, Alice and Bob must traverse the path so that they always lie on a direct east-west line.

    We can describe the zigzag path as two arrays $X[0..n]$ and $Y[0..n]$, containing the x- and y-coordinates of the corners of the path, in order from the southwest endpoint to the southeast endpoint. The X array is sorted in increasing order, and $Y[0] = Y[n]$. The path is a sequence of straight line segments connecting these corners.

    (a) Suppose $Y[0] = Y[n] = 0$ and $Y[i] > 0$ for every other index i; that is, the endpoints of the path are strictly below every other point on the path.

[16] Fun fact: Damien Chazelle, the director of *Whiplash* and *La La Land*, is the son of Princeton computer science professor and electric guitarist Bernard Chazelle.

Figure 5.19. Alice and Bob meet. Alice walks backward in step 2, and Bob walks backward in steps 5 and 6.

Prove that for any path P meeting these conditions, Alice and Bob can *always* meet. *[Hint: Describe a graph that models all possible locations of the couple along the path. What are the vertices of this graph? What are the edges? Use the* ***Handshake Lemma****: Every graph has an even number of vertices with odd degree.]*

(b) If the endpoints of the path are *not* below every other vertex, Alice and Bob might still be able to meet, or they might not. Describe an algorithm to decide whether Alice and Bob can meet, without either breaking east-west eye contact or stepping off the path, given the arrays $X[0..n]$ and $Y[0..n]$ as input.

♥(c) Describe an algorithm for part (b) that runs in $O(n)$ time.

23. The famous puzzle-maker Kaniel the Dane invented a solitaire game played with two tokens on an $n \times n$ square grid. Some squares of the grid are marked as *obstacles*, and one grid square is marked as the *target*. In each turn, the player must move one of the tokens from is current position *as far as possible* upward, downward, right, or left, stopping just before the token hits (1) the edge of the board, (2) an obstacle square, or (3) the other token. The goal is to move either of the tokens onto the target square.

For example, we can solve the puzzle shown in Figure 5.20 by moving the red token down until it hits the obstacle, then moving the green token left until it hits the red token, and then moving the red token left, down, right, and up. The red token stops at the target on the 6th move *because* the green token is just above the target square.

Describe and analyze an algorithm to determine whether an instance of this puzzle is solvable. Your input consist of the integer n, a list of obstacle locations, the target location, and the initial locations of the tokens. The

Figure 5.20. An instance of Kaniel the Dane's puzzle that can be solved in six moves. Circles indicate initial token positions; black squares are obstacles; the center square is the target.

output of your algorithm is a single boolean: TRUE if the given puzzle is solvable and FALSE otherwise. *[Hint: Don't forget about the time required to construct the graph.]*

♥24. ***Rectangle Walk*** is a new abstract puzzle game, available for only 99¢ on Steam, iOS, Android, Xbox One, Playstation 5, Nintendo Wii U, Atari 2600, Palm Pilot, Commodore 64, TRS-80, Sinclair ZX-1, DEC PDP-8, PLATO, Zuse Z3, Duramesc, Odhner Arithmometer, Analytical Engine, Jacquard Loom, Horologium Mirabile Lundense, Leibniz Stepped Reckoner, Al-Jazari's Robot Band, Yan Shi's Automaton, Antikythera Mechanism, Knotted Rope, Ishango Bone, and Pile of Rocks.

The game is played on an $n \times n$ grid of black and white squares. The player moves a rectangle through this grid, subject to the following conditions:

- The rectangle must be aligned with the grid; that is, the top, bottom, left, and right coordinates must be integers.
- The rectangle must fit within the $n \times n$ grid, and it must contain at least one grid cell.
- The rectangle must not contain a black square.
- In a single move, the player can replace the current rectangle r with any rectangle r' that either contains r or is contained in r.

Initially, the player's rectangle is a 1×1 square in the upper right corner. The player's goal is to reach a 1×1 square in the bottom left corner using as few moves as possible.

Figure 5.21. The first five steps of a Rectangle Walk.

Describe and analyze an algorithm to compute the length of the shortest Rectangle Walk in a given bitmap. Your input is an array $M[1..n, 1..n]$, where $M[i,j] = 1$ indicates a black square and $M[i,j] = 0$ indicates a white square. Assume that a valid rectangle walk exists; in particular, $M[1,1] = M[n,n] = 0$. For example, given the bitmap shown above, your algorithm should return the integer 18. *[Hint: Don't forget about the time required to construct the graph!!]*

25. ***Racetrack*** (also known as *Graph Racers* and *Vector Rally*) is a two-player paper-and-pencil racing game that Jeff played on the bus in 5th grade.[17] The game is played with a track drawn on a sheet of graph paper. The players alternately choose a sequence of grid points that represent the motion of a car around the track, subject to certain constraints explained below.

velocity	position
$(0,0)$	$(1,5)$
$(1,0)$	$(2,5)$
$(2,-1)$	$(4,4)$
$(3,0)$	$(7,4)$
$(2,1)$	$(9,5)$
$(1,2)$	$(10,7)$
$(0,3)$	$(10,10)$
$(-1,4)$	$(9,14)$
$(0,3)$	$(9,17)$
$(1,2)$	$(10,19)$
$(2,2)$	$(12,21)$
$(2,1)$	$(14,22)$
$(2,0)$	$(16,22)$
$(1,-1)$	$(17,21)$
$(2,-1)$	$(19,20)$
$(3,0)$	$(22,20)$
$(3,1)$	$(25,21)$

Figure 5.22. A 16-step Racetrack run, on a 25 × 25 track. This is *not* the shortest run on this track.

Each car has a *position* and a *velocity*, both with integer x- and y-coordinates. A subset of grid squares is marked as the *starting area*, and another subset is marked as the *finishing area*. The initial position of each car is chosen by the player somewhere in the starting area; the initial velocity of each car is always $(0,0)$. At each step, the player optionally increments or decrements either or both coordinates of the car's velocity; in other words, each component of the velocity can change by at most 1 in a single step. The car's new position is then determined by adding the new velocity to the car's previous position. The new position must be inside the track; otherwise, the car crashes and that player loses the race. The race ends when the first car reaches a position inside the finishing area.

[17]The actual game is a bit more complicated than the version described here. See http://harmmade.com/vectorracer/ for an excellent online version.

Suppose the racetrack is represented by an $n \times n$ array of bits, where each 0 bit represents a grid point inside the track, each 1 bit represents a grid point outside the track, the "starting area" is the first column, and the "finishing area" is the last column.

Describe and analyze an algorithm to find the minimum number of steps required to move a car from the starting line to the finish line of a given racetrack.

26. A *rolling die maze* is a puzzle involving a standard six-sided die (a cube with numbers on each side) and a grid of squares. You should imagine the grid lying on a table; the die always rests on and exactly covers one square of the grid. In a single step, you can *roll* the die 90 degrees around one of its bottom edges, moving it to an adjacent square one step north, south, east, or west.

Some squares in the grid may be *blocked*; the die can never rest on a blocked square. Other squares may be *labeled* with a number; whenever the die rests on a labeled square, the number on the *top* face of the die must equal the label. Squares that are neither labeled nor marked are *free*. You may not roll the die off the edges of the grid. A rolling die maze is *solvable* if it is possible to place a die on the lower left square and roll it to the upper right square under these constraints.

Figure 5.23. Rolling a die

Figure 5.24 shows four rolling die mazes. Assuming we use a standard die with 1 and 6 on opposite sides, only the first two mazes are solvable. For example, the first maze is solvable by by placing the die on the lower left square with 1 on the top face, and then rolling the die east, then north, then north, then east.

Figure 5.24. Four rolling die mazes; only the first two are solvable.

(a) Suppose the input is a two-dimensional array $L[1..n, 1..n]$, where each entry $L[i, j]$ stores the label of the square in the ith row and jth column, where 0 means the square is free and -1 means the square is blocked.

Describe and analyze a polynomial-time algorithm to determine whether the given rolling die maze is solvable.

♥(b) Now suppose the maze is specified *implicitly* by a list of labeled and blocked squares. Specifically, suppose the input consists of an integer M, specifying the height and width of the maze, and an array $S[1..n]$, where each entry $S[i]$ is a triple (x, y, L) indicating that square (x, y) has label L. As in the explicit encoding, label -1 indicates that the square is blocked; free squares are not listed in S at all. Describe and analyze an efficient algorithm to determine whether the given rolling die maze is solvable. For full credit, the running time of your algorithm should be polynomial in the input size n.

[Hint: You have some freedom in how to place the initial die. There are rolling die mazes that can be solved only if the initial position is chosen correctly.]

♥27. Suppose you are given an arbitrary directed graph G in which each edge is colored either red or blue, along with two special vertices s and t.

(a) Describe an algorithm that either computes a walk from s to t such that the pattern of red and blue edges along the walk is a palindrome, or correctly reports that no such walk exists.

(b) Describe an algorithm that either computes the *shortest* walk from s to t such that the pattern of red and blue edges along the walk is a palindrome, or correctly reports that no such walk exists.

[Hint: Where did we last see palindromes?]

♠♥28. Draughts, also known in the United States as "checkers", is a game played on an $m \times m$ grid of squares, alternately colored light and dark.[18] The game is usually played on an 8×8 or 10×10 board, but the rules easily generalize to any board size. Each dark square is occupied by at most one game piece (usually called a *checker* in the U.S.), which is either black or white; light squares are always empty. One player ("White") moves the white pieces; the other ("Black") moves the black pieces. A player loses when her last piece is taken off the board.

[18]The counting tables used by medieval English government accountants were covered by a green cloth with black squares in a checker pattern; disk-shaped counters were placed in these squares to represent values. For this reason, the British government's accountants have been collectively known since the 10th century as the *Exchequer*. The actual counting tables were used by the Exchequer to tally tax payments well into the 19th century.

Consider the following simple version of the game, essentially American checkers or British draughts, but where every piece is a king.[19] Pieces can be moved in any of the four diagonal directions. On each turn, a player either *moves* one of her pieces one step diagonally into an empty square, or makes a series of *jumps* with one of her pieces. In each jump, the piece moves to an empty square two steps away in any diagonal direction, but only if the intermediate square is occupied by a piece of the opposite color; this enemy piece is *captured* and immediately removed from the board. All jumps in the same turn must be made with the same piece.

Describe an algorithm to decide whether White can capture every black piece, thereby winning the game, *in a single turn*. The input consists of the width of the board (m), a list of positions of white pieces, and a list of positions of black pieces. For full credit, your algorithm should run in $O(n)$ time, where n is the total number of pieces. *[Hint: The greedy strategy—make arbitrary jumps until you get stuck—does* ***not*** *always find a winning sequence of jumps even when one exists. See problem 5. Parity, parity, parity.]*

Figure 5.25. White wins in one turn.

Figure 5.26. White cannot win in one turn from either of these positions.

[19] Most other variants of draughts have "flying kings", which behave *very* differently than kings in the British/American game, and which make this problem *much* more difficult, as we will see in Chapter 12.

And, for the hous is crinkled to and fro,
And hath so queinte weyes for to go—
For hit is shapen as the mase is wroght—
Therto have I a remedie in my thoght,
That, by a clewe of twyne, as he hath goon,
The same wey he may returne anoon,
Folwing alwey the threed, as he hath come.

— Geoffrey Chaucer, *The Legend of Good Women* (c. 1385)

"Com'è bello il mondo e come sono brutti i labirinti!" dissi sollevato.
"Come sarebbe bello il mondo se ci fosse una regola per girare nei labirinti,"
rispose il mio maestro.
["How beautiful the world is, and how ugly labyrinths are," I said, relieved.
"How beautiful the world would be if there were a procedure for moving through
labyrinths," my master replied.]

— Umberto Eco, *Il nome della rosa* (1980)
English translation (*The Name of the Rose*) by William Weaver (1983)

6

Depth-First Search

In the previous chapter, we considered a generic algorithm—whatever-first search—for traversing arbitrary graphs, both undirected and directed. In this chapter, we focus on a particular instantiation of this algorithm called *depth-first search*, and primarily on the behavior of this algorithm in directed graphs.

Although depth-first search can be accurately described as "whatever-first search with a stack", the algorithm is normally implemented recursively, rather than using an explicit stack:

```
DFS(v):
  if v is unmarked
      mark v
      for each edge v→w
          DFS(w)
```

We can make this algorithm slightly faster (in practice) by checking whether a node is marked *before* we recursively explore it. This modification ensures that we call DFS(v) only once for each vertex v. We can further modify the algorithm to compute other useful information about the vertices and edges, by introducing two black-box subroutines, PreVisit and PostVisit, which we leave unspecified for now.

```
DFS(v):
  mark v
  PreVisit(v)
  for each edge vw
      if w is unmarked
          parent(w) ← v
          DFS(w)
  PostVisit(v)
```

Recall that a node w is *reachable* from another node v in a directed graph G—or more simply, v can reach w—if and only if G contains a directed path from v to w. Let ***reach*(v)** denote the set of vertices reachable from v (including v itself). If we unmark all vertices in G, and then call DFS(v), the set of marked vertices is precisely *reach*(v).

Reachability in undirected graphs is symmetric: v can reach w if and only if w can reach v. As a result, after unmarking all vertices of an undirected graph G, calling DFS(v) traverses the entire component of v, and the parent pointers define a spanning tree of that component.

The situation is more subtle with directed graphs, as shown in the figure below. Even though the graph is "connected", different vertices can reach different, and potentially overlapping, portions of the graph. The parent pointers assigned by DFS(v) define a tree rooted at v whose vertices are precisely *reach*(v), but this is not necessarily a spanning tree of the graph.

Figure 6.1. Depth-first trees rooted at different vertices in the same directed graph.

As usual, we can extend our reachability algorithm to traverse the *entire* input graph, even if it is disconnected, using the standard wrapper function shown on the left in Figure 6.2. Here we add a generic black-box subroutine

PREPROCESS to perform any necessary preprocessing for the PREVISIT and POSTVISIT functions.

```
DFSALL(G):
  PREPROCESS(G)
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          DFS(v)
```

```
DFSALL(G):
  PREPROCESS(G)
  add vertex s
  for all vertices v
      add edge s→v
      unmark v
  DFS(s)
```

Figure 6.2. Two formulations of the standard wrapper algorithm for depth-first search

Alternatively, if we are allowed to modify the graph, we can add a new *source* vertex s, with edges to every other vertex in G, and then make a single call to DFS(s), as shown on the right of Figure 6.2. Now the resulting parent pointers always define a spanning tree of the *augmented* input graph, but not of the *original* input graph. Otherwise, the two wrapper functions have essentially identical behavior; choosing one or the other is entirely a matter of convenience.[1]

Again, this algorithm behaves slightly differently for undirected and directed graphs. In undirected graphs, as we saw in the previous chapter, it is easy to adapt DFSALL to count the components of a graph; in particular, the parent pointers computed by DFSALL define a spanning forest of the input graph, containing a spanning tree for each component. When the graph is directed, however, DFSALL may discover any number of "components" between 1 and V, even when the graph is "connected", depending on the precise structure of the graph and the order in which the wrapper algorithm visits the vertices.

6.1 Preorder and Postorder

Hopefully you are already familiar with preorder and postorder traversals of rooted *trees*, both of which can be computed using depth-first search. Similar traversal orders can be defined for arbitrary directed graphs—even if they are disconnected—by passing around a counter, as shown in Figure 6.3. Equivalently, we can use our generic depth-first-search algorithm with the following subroutines PREPROCESS, PREVISIT, and POSTVISIT.

```
PREPROCESS(G):
  clock ← 0
```

```
PREVISIT(v):
  clock ← clock + 1
  v.pre ← clock
```

```
POSTVISIT(v):
  clock ← clock + 1
  v.post ← clock
```

[1]The equivalence of these two wrapper functions is a specific feature of ***depth***-first search. In particular, wrapping *breadth*-first search in a for-loop to visit every vertex does *not* yield the same traversal order as adding a source vertex and invoking breadth-first search at s.

```
DFSAll(G):
  clock ← 0
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          clock ← DFS(v, clock)
```

```
DFS(v, clock):
  mark v
  clock ← clock + 1;  v.pre ← clock
  for each edge v→w
      if w is unmarked
          w.parent ← v
          clock ← DFS(w, clock)
  clock ← clock + 1;  v.post ← clock
  return clock
```

Figure 6.3. Defining preorder and postorder via depth-first search.

In either formulation, this algorithm assigns assigns $v.pre$ (and advances the clock) just after pushing v onto the recursion stack, and it assigns $v.post$ (and advances the clock) just before popping v off the recursion stack. It follows that for any two vertices u and v, the intervals $[u.pre, u.post]$ and $[v.pre, v.post]$ are either disjoint or nested. Moreover, $[u.pre, u.post]$ contains $[v.pre, v.post]$ if and only if DFS(v) is called during the execution of DFS(u), or equivalently, if and only if u is an ancestor of v in the final forest of parent pointers.

After DFSAll labels every node in the graph, the labels $v.pre$ define a ***preordering*** of the vertices, and the labels $v.post$ define a ***postordering*** of the vertices.[2] With a few trivial exceptions, every graph has several different pre- and postorderings, depending on the order that DFS considers edges leaving each vertex, and the order that DFSAll considers vertices.

For the rest of this chapter, we refer to $v.pre$ as the *starting time* of v (or less formally, "when v starts"), $v.post$ as the *finishing time* of v (or less formally, "when v finishes"), and the interval between the starting and finishing times as the *active interval* of v (or less formally, "while v is active").

Classifying Vertices and Edges

During the execution of DFSAll, each vertex v of the input graph has one of three states:

- ***new*** if DFS(v) has not been called, that is, if $clock < v.pre$;
- ***active*** if DFS(v) has been called but has not returned, that is, if $v.pre \leq clock < v.post$;
- ***finished*** if DFS(v) has returned, that is, if $v.post \leq clock$.

Because starting and finishing times correspond to pushes and pops on the recursion stack, a vertex is active if and only if it is on the recursion stack. It follows that the active nodes always comprise a directed path in G.

[2]Confusingly, *both* of these orders are sometimes called "depth-first ordering". Please don't do that.

Figure 6.4. A depth-first forest of a directed graph, and the corresponding active intervals of its vertices, defining the preordering *abfgchdlokpeinjm* and the postordering *dkoplhcgfbamjnie*. Forest edges are solid; dashed edges are explained in Figure 6.5.

The edges of the input graph fall into four different classes, depending on how their active intervals intersect. Fix your favorite edge $u \to v$.

- If v is new when DFS(u) begins, then DFS(v) must be called during the execution of DFS(u), either directly or through some intermediate recursive calls. In either case, u is a proper ancestor of v in the depth-first forest, and $u.pre < v.pre < v.post < u.post$.
  - If DFS(u) calls DFS(v) directly, then $u = v.parent$ and $u \to v$ is called a ***tree edge***.
  - Otherwise, $u \to v$ is called a ***forward edge***.
- If v is active when DFS(u) begins, then v is already on the recursion stack, which implies the opposite nesting order $v.pre < u.pre < u.post < v.post$. Moreover, G must contain a directed path from v to u. Edges satisfying this condition are called ***back edges***.
- If v is finished when DFS(u) begins, we immediately have $v.post < u.pre$. Edges satisfying this condition are called ***cross edges***.
- Finally, the fourth ordering $u.post < v.pre$ is impossible.

These edge classes are illustrated in Figure 6.5. Again, the actual classification of edges depends on the order in which DFSAll considers vertices and the order in which DFS considers the edges leaving each vertex.

Figure 6.5. Classification of edges by depth-first search.

Finally, the following key lemma characterizes ancestors and descendants in any depth-first forest according to vertex states during the traversal.

Lemma 6.1. *Fix an arbitrary depth-first traversal of any directed graph G. The following statements are equivalent for all vertices u and v of G.*

(a) u is an ancestor of v in the depth-first forest.
(b) $u.pre \leq v.pre < v.post \leq u.post$.
(c) Just after DFS(v) is called, u is active.
(d) Just before DFS(u) is called, there is a path from u to v in which every vertex (including u and v) is new.

Proof: First, suppose u is an ancestor of v in the depth-first forest. Then by definition there is a path P of tree edges u to v. By induction on the path length, we have $u.pre \leq w.pre < w.post \leq u.post$ for every vertex w in P, and thus every vertex in P is new before DFS(u) is called. In particular, we have $u.pre \leq v.pre < v.post \leq u.post$, which implies that u is active while DFS(v) is executing.

Because parent pointers correspond to recursive calls, $u.pre \leq v.pre < v.post \leq u.post$ implies that u is an ancestor of v.

Suppose u is active just after DFS(v) is called. Then $u.pre \leq v.pre < v.post \leq u.post$, which implies that there is a path of (zero or more) tree edges from u, through the intermediate nodes on the recursion stack (if any), to v.

Finally, suppose u is not an ancestor of v. Fix an arbitrary path P from u to v, let x be the first vertex in P that is not a descendant of u, and let w be the predecessor of x in P. The edge $w \to x$ guarantees that $x.pre < w.post$, and $w.post < u.post$ because w is a descendant of u, so $x.pre < u.post$. It follows that $x.pre < u.pre$, because otherwise x would be a descendant of u. Because active intervals are properly nested, there are only two possibilities:

- If $u.post < x.post$, then x is active when DFS(u) is called.
- If $x.post < u.pre$, then x is already finished when DFS(u) is called.

We conclude that *every* path from u to v contains a vertex that is not new when DFS(u) is called. □

6.2 Detecting Cycles

A ***directed acyclic graph*** or ***dag*** is a directed graph with no directed cycles. Any vertex in a dag that has no incoming vertices is called a ***source***; any vertex with no outgoing edges is called a ***sink***. An isolated vertex with no incident edges at all is both a source and a sink. Every dag has at least one source and one sink, but may have more than one of each. For example, in the graph with n vertices but no edges, every vertex is a source and every vertex is a sink.

Figure 6.6. A directed acyclic graph. Vertices e, f, and j are sources; vertices b, c, and p are sinks.

Recall from our earlier case analysis that if $u.post < v.post$ for any edge $u \to v$, the graph contains a directed path from v to u, and therefore contains a directed *cycle* through the edge $u \to v$. Thus, we can determine whether a given directed graph G is a dag in $O(V + E)$ time by computing a postordering of the vertices and then checking each edge by brute force.

Alternatively, instead of numbering the vertices, we can explicitly maintain the status of each vertex and immediately return FALSE if we ever discover an edge to an active vertex. This algorithm also runs in $O(V + E)$ time; see Figure 6.7.

```
IsAcyclic(G):
  for all vertices v
      v.status ← New
  for all vertices v
      if v.status = New
          if IsAcyclicDFS(v) = False
              return False
  return True
```

```
IsAcyclicDFS(v):
  v.status ← Active
  for each edge v→w
      if w.status = Active
          return False
      else if w.status = New
          if IsAcyclicDFS(w) = False
              return False
  v.status ← Finished
  return True
```

Figure 6.7. A linear-time algorithm to determine if a graph is acyclic.

6.3 Topological Sort

A ***topological ordering*** of a directed graph G is a total order $\prec$ on the vertices such that $u \prec v$ for every edge $u \to v$. Less formally, a topological ordering arranges the vertices along a horizontal line so that all edges point from left to right. A topological ordering is clearly impossible if the graph G has a directed cycle—the rightmost vertex of the cycle would have an edge pointing to the left!

On the other hand, consider an arbitrary postordering of an arbitrary directed graph G. Our earlier analysis implies that $u.post < v.post$ for any edge $u \to v$, then G contains a directed path from v to u, and therefore contains a directed cycle through $u \to v$. Equivalently, if G is acyclic, then $u.post > v.post$ for every edge $u \to v$. It follows that every directed acyclic graph G has a topological ordering; in particular, the reversal of any postordering of G is a topological ordering of G.

Figure 6.8. Reversed postordering of the dag from Figure 6.6.

If we require the topological ordering in a separate data structure, we can simply write the vertices into an array in reverse postorder, in $O(V + E)$ time, as shown in Figure 6.9.

Implicit Topological Sort

But recording the topological order into a separate data structure is usually overkill. In most applications of topological sort, the ordered list of the vertices is not our actual goal; rather, we want to perform some fixed computation at each vertex of the graph, either in topological order or in reverse topological order. For these applications, it is not necessary to *record* the topological order at all!

```
TopologicalSort(G):
  for all vertices v
      v.status ← New
  clock ← V
  for all vertices v
      if v.status = New
          clock ← TopSortDFS(v, clock)
  return S[1..V]
```

```
TopSortDFS(v, clock):
  v.status ← Active
  for each edge v→w
      if w.status = New
          clock ← TopSortDFS(v, clock)
      else if w.status = Active
          fail gracefully
  v.status ← Finished
  S[clock] ← v
  clock ← clock − 1
  return clock
```

Figure 6.9. Explicit topological sort

If we want to process a directed acyclic graph in *reverse* topological order, it suffices to process each vertex at the end of its recursive depth-first search. After all, topological order is the same as reversed postorder!

```
PostProcess(G):
  for all vertices v
      v.status ← New
  for all vertices v
      if v is unmarked
          PostProcessDFS(v)
```

```
PostProcessDFS(v):
  v.status ← Active
  for each edge v→w
      if w.status = New
          PostProcessDFS(w)
      else if w.status = Active
          fail gracefully
  v.status ← Finished
  Process(v)
```

If we already *know* that the input graph is acyclic, we can further simplify the algorithm by simply marking vertices instead of recording their search status.

```
PostProcessDag(G):
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          PostProcessDagDFS(s)
```

```
PostProcessDagDFS(v):
  mark v
  for each edge v→w
      if w is unmarked
          PostProcessDagDFS(w)
  Process(v)
```

This is just the standard depth-first search algorithm, with PostVisit renamed to Process!

Because it is such a common operation on directed acyclic graphs, I sometimes express postorder processing of a dag idiomatically as follows:

```
PostProcessDag(G):
  for all vertices v in postorder
      Process(v)
```

For example, our earlier explicit topological sort algorithm can be written as follows:

```
TopologicalSort(G):
  clock ← V
  for all vertices v in postorder
      S[clock] ← v
      clock ← clock − 1
  return S[1..V]
```

To process a dag in *forward* topological order, we can record a topological ordering of the vertices into an array and then run a simple for-loop. Alternatively, we can apply depth-first search to the ***reversal*** of G, denoted $\boldsymbol{rev(G)}$, obtained by replacing each each $v \to w$ with its reversal $w \to v$. Reversing a directed cycle gives us another directed cycle with the opposite orientation, so the reversal of a dag is another dag. Every source in G is a sink in $rev(G)$ and vice versa; it follows inductively that every topological ordering of $rev(G)$ is the reversal of a topological ordering of G.[3] The reversal of any directed graph (represented in a standard adjacency list) can be computed in $O(V + E)$ time; the details of this construction are left as an easy exercise.

6.4 Memoization and Dynamic Programming

Our topological sort algorithm is arguably the model for a wide class of dynamic programming algorithms. Recall that the ***dependency graph*** of a recurrence has a vertex for every recursive subproblem and an edge from one subproblem to another if evaluating the first subproblem requires a recursive evaluation of the second. The dependency graph must be acyclic, or the naïve recursive algorithm would never halt.

Evaluating any recurrence with memoization is *exactly* the same as performing a depth-first search of the dependency graph. In particular, a vertex of the dependency graph is "marked" if the value of the corresponding subproblem has already been computed. The black-box subroutines PreVisit and PostVisit are proxies for the actual value computation. See Figure 6.10.

Carrying this analogy further, evaluating a recurrence *using dynamic programming* is the same as evaluating all subproblems in the dependency graph of the recurrence in reverse topological order—every subproblem is considered *after* the subproblems it depends on. Thus, *every* dynamic programming algorithm is equivalent to a postorder traversal of the dependency graph of its underlying recurrence!

[3] A postordering of the reversal of G is not necessarily the reversal of a postordering of G, even though both are topological orderings of G.

```
MEMOIZE(x):
  if value[x] is undefined
      initialize value[x]

      for all subproblems y of x
          MEMOIZE(y)
          update value[x] based on value[y]
      finalize value[x]
```

```
DFS(v):
  if v is unmarked
      mark v
      PREVISIT(x)
      for all edges v→w
          DFS(w)

      POSTVISIT(x)
```

Figure 6.10. Memoized recursion is depth-first search. Depth-first search is memoized recursion.

```
DYNAMICPROGRAMMING(G):
  for all subproblems x in postorder
      initialize value[x]
      for all subproblems y of x
          update value[x] based on value[y]
      finalize value[x]
```

Figure 6.11. Dynamic programming is postorder traversal.

However, there are some minor differences between most dynamic programming algorithms and topological sort. First, in most dynamic programming algorithms, the dependency graph is *implicit*—the nodes and edges are not explicitly stored in memory, but rather are encoded by the underlying recurrence. But this difference really is minor; as long as we can enumerate recursive subproblems in constant time each, we can traverse the dependency graph exactly *as if* it were explicitly stored in an adjacency list.

More significantly, most dynamic programming recurrences have highly structured dependency graphs. For example, as we discussed in Chapter 5, the dependency graph for the edit distance recurrence is a regular grid with diagonals, and the dependency graph for optimal binary search trees is an upper triangular grid with all possible rightward and upward edges. This regular structure allows us to hard-wire a suitable evaluation order directly into the algorithm, typically as a collection of nested loops, so there is no need to topologically sort the dependency graph at run time. We previously called the reverse topological order an *evaluation order*.

Dynamic Programming in Dags

Conversely, we can use depth-first search to build dynamic programming algorithms for problems with less structured dependency graphs. For example, consider the ***longest path*** problem, which asks for the path of *maximum* total weight from one node s to another node t in a directed graph G with weighted edges. In general directed graphs, the longest path problem is NP-hard (by an easy reduction from the traveling salesman problem; see Chapter 12), but it is

Figure 6.12. The dependency **dag** of the edit distance recurrence.

easy to if the input graph G is acyclic, we can compute the longest path in G in linear time, as follows.

Fix the target vertex t, and for any node v, let $\boldsymbol{LLP(v)}$ denote the Length of the Longest Path in G from v to t. If G is a dag, this function satisfies the recurrence

$$LLP(v) = \begin{cases} 0 & \text{if } v = t, \\ \max\left\{\ell(v{\to}w) + LLP(w) \;\middle|\; v{\to}w \in E\right\} & \text{otherwise,} \end{cases}$$

where $\ell(v{\to}w)$ denotes the given weight ("length") of edge $v{\to}w$, and $\max\varnothing = -\infty$. In particular, if v is a *sink* but not equal to t, then $LLP(v) = -\infty$.

The dependency graph for this recurrence is the input graph G itself: subproblem $LLP(v)$ depends on subproblem $LLP(w)$ if and only if $v{\to}w$ is an edge in G. Thus, we can evaluate this recursive function in $O(V + E)$ time by performing a depth-first search of G, starting at s. The algorithm memoizes each length $LLP(v)$ into an extra field in the corresponding node v.

```
LongestPath(v, t):
  if v = t
      return 0
  if v.LLP is undefined
      v.LLP ← −∞
      for each edge v→w
          v.LLP ← max{v.LLP, ℓ(v→w) + LongestPath(w, t)}
      return v.LLP
```

In principle, we can transform this memoized recursive algorithm into a dynamic programming algorithm via topological sort:

```
LongestPath(s, t):
  for each node v in postorder
      if v = t
          v.LLP ← 0
      else
          v.LLP ← −∞
          for each edge v→w
              v.LLP ← max{v.LLP, ℓ(v→w) + w.LLP}
  return s.LLP
```

These two algorithms are arguably identical—the recursion in the first algorithm and the for-loop in the second algorithm represent the "same" depth-first search! Choosing one of these formulations over the other is entirely a matter of convenience.

Almost any dynamic programming problem that asks for an optimal *sequence* of decisions can be recast as finding an optimal *path* in some associated dag. For example, the text segmentation, subset sum, longest increasing subsequence, and edit distance problems we considered in Chapters 2 and 3 can all be reformulated as finding either a longest path or a shortest path in a dag, possibly with weighted vertices or edges. In each case, the dag in question is the dependency graph of the underlying recurrence. On the other hand, "tree-shaped" dynamic programming problems, like finding optimal binary search trees or maximum independent sets in trees, cannot be recast as finding an optimal *path* in a dag.

6.5 Strong Connectivity

Let's go back to the proper definition of connectivity in directed graphs. Recall that one vertex u can *reach* another vertex v in a directed graph G if G contains a directed path from u to v, and that $reach(u)$ denotes the set of all vertices that u can reach. Two vertices u and v are ***strongly connected*** if u can reach v and v can reach u. A directed graph is strongly connected if and only if every pair of vertices is strongly connected.

Tedious definition-chasing implies that strong connectivity is an equivalence relation over the set of vertices of any directed graph, just like connectivity in undirected graphs. The equivalence classes of this relation are called the ***strongly connected components***—or more simply, the ***strong components***—of G. Equivalently, a strong component of G is a maximal strongly connected subgraph of G. A directed graph G is strongly connected if and only if G has exactly one strong component; at the other extreme, G is a dag if and only if every strong component of G consists of a single vertex.

The ***strong component graph scc(G)*** is another directed graph obtained from G by contracting each strong component to a single vertex and collapsing

parallel edges. (The strong component graph is sometimes also called the *meta-graph* or *condensation* of G.) It's not hard to prove (hint, hint) that $scc(G)$ is always a dag. Thus, at least in principle, it is possible to topologically order the strong components of G; that is, the vertices can be ordered so that every *back* edge joins two edges in the same strong component.

Figure 6.13. The strong components of a graph G and the strong component graph $scc(G)$.

It is straightforward to compute the strong component of a single vertex v in $O(V + E)$ time. First we compute $reach(v)$ via whatever-first search. Then we compute $reach^{-1}(v) = \{u \mid v \in reach(u)\}$ by searching the reversal of G. Finally, the strong component of v is the intersection $reach(v) \cap reach^{-1}(v)$. In particular, we can determine whether the entire graph is strongly connected in $O(V + E)$ time.

Similarly, we can compute *all* the strong components in a directed graph by combining the previous algorithm with our standard wrapper function. However, the resulting algorithm runs in $O(VE)$ time; there are at most V strong components, and each requires $O(E)$ time to discover, even when the graph is a dag. Surely we can do better! After all, we only need $O(V + E)$ time to decide whether every strong component is a single vertex.

6.6 Strong Components in Linear Time

In fact, there are several algorithms to compute strong components in $O(V + E)$ time, all of which rely on the following observation.

Lemma 6.2. *Fix a depth-first traversal of any directed graph G. Each strong component C of G contains exactly one node that does not have a parent in C. (Either this node has a parent in another strong component, or it has no parent.)*

Proof: Let C be an arbitrary strong component of G. Consider any path from one vertex $v \in C$ to another vertex $w \in C$. Every vertex on this path can reach w, and thus can reach every vertex in C; symmetrically, every node on this path can be reached by v, and thus can be reached by every vertex in C. We conclude that every vertex on this path is also in C.

Let v be the vertex in C with the earliest starting time. If v has a parent, then $parent(v)$ starts before v and thus cannot be in C.

Now let w be another vertex in C. Just before DFS(v) is called, every vertex in C is new, so there is a path of new vertices from v to w. Lemma 6.1 now implies that w is a descendant of v in the depth-first forest. Every vertex on the path of tree edges v to w lies in C; in particular, $parent(w) \in C$. □

The previous lemma implies that each strong component of a directed graph G defines a connected subtree of any depth-first forest of G. In particular, for any strong component C, the vertex in C with the earliest starting time is the lowest common ancestor of all vertices in C; we call this vertex the ***root*** of C.

Figure 6.14. Strong components are contiguous in the depth-first forest.

I'll present two algorithms, both of which follow the same intuitive outline. Let C be any strong component of G that is a sink in $scc(G)$; we call C a ***sink component***. Equivalently, C is a sink component if the reach of any vertex in C is precisely C. We can find all the strong components in G by repeatedly finding a vertex v in some sink component (somehow), finding the vertices reachable from v, and removing that sink component from the input graph, until no vertices remain. This isn't quite an algorithm yet, because it's not clear how to find a vertex in a sink component!

```
StrongComponents(G):
  count ← 0
  while G is non-empty
      C ← ∅
      count ← count + 1
      v ← any vertex in a sink component of G      ⟨⟨Magic!⟩⟩
      for all vertices w in reach(v)
          w.label ← count
          add w to C
      remove C and its incoming edges from G
```

Figure 6.15. Almost an algorithm to compute strong components.

Kosaraju and Sharir's Algorithm

At first glance, finding a vertex in a sink component *quickly* seems quite difficult. However, it's actually quite easy to find a vertex in a *source* component—a strong component of G that corresponds to a *source* in $scc(G)$—using depth-first search.

Lemma 6.3. *The last vertex in any postordering of G lies in a source component of G.*

Proof: Fix a depth-first traversal of G, and let v be the last vertex in the resulting postordering. Then DFS(v) must be the last direct call to DFS made by the wrapper algorithm DFSAll. Moreover, v is the root of one of the trees in the depth-first forest, so any node x with $x.post > v.pre$ is a descendant of v. Finally, v is the root of its strong component C.

For the sake of argument, suppose there is an edge $x \to y$ such that $x \notin C$ and $y \in C$. Then x can reach y, and y can reach v, so x can reach v. Because v is the root of C, vertex y is a descendant of v, and thus $v.pre < y.pre$. The edge $x \to y$ guarantees that $y.pre < x.post$ and therefore $v.pre < x.post$. It follows that x is a descendant of v. But then v can reach x (through tree edges), contradicting our assumption that $x \notin C$. □

It is easy to check (hint, hint) that $rev(scc(G)) = scc(rev(G))$ for any directed graph G. Thus, the *last* vertex in a postordering of $rev(G)$ lies in a *sink* component of the original graph G. Thus, if we traverse the graph a second time, where the wrapper function follows a reverse postordering of $rev(G)$, then each call to DFS visits exactly one strong component of G.[4]

Putting everything together, we obtain the algorithm shown in Figure 6.16, which counts and labels the strong components of any directed graph in $O(V+E)$ time. This algorithm was discovered (but never published) by Rao Kosaraju in 1978, and later independently rediscovered by Micha Sharir in 1981.[5] The Kosaraju-Sharir algorithm has two phases. The first phase performs a depth-first search of $rev(G)$, pushing each vertex onto a stack when it is finished. In the second phase, we perform a *whatever*-first traversal of the original graph G, considering vertices in the order they appear on the stack. The algorithm labels each vertex with the root of its strong component (with respect to the second depth-first traversal).

Figure 6.17 shows the Kosaraju-Sharir algorithm running on our example graph. With only minor modifications to the algorithm, we can also compute the strong component graph $scc(G)$ in $O(V+E)$ time.

[4]Again: A reverse postordering of $rev(G)$ is not the same as a postordering of G.

[5]There are rumors that the same algorithm appears in the Russian literature even before Kosaraju, but I haven't found a reliable source for that rumor yet.

```
KosarajuSharir(G):
  S ← new empty stack
  for all vertices v
      unmark v
      v.root ← None
  ⟨⟨Phase 1: Push in postorder in rev(G)⟩⟩
  for all vertices v
      if v is unmarked
          PushPostRevDFS(v, S)
  ⟨⟨Phase 2: DFS again in stack order⟩⟩
  while S is non-empty
      v ← Pop(S)
      if v.root = None
          LabelOneDFS(v, v)
```

```
PushPostRevDFS(v, S):
  mark v
  for each edge u→v    ⟨⟨Reversed!⟩⟩
      if u is unmarked
          PushPostRevDFS(u, S)
  Push(v, S)
```

```
LabelOneDFS(v, r):
  v.root ← r
  for each edge v→w
      if w.root = None
          LabelOneDFS(w, r)
```

Figure 6.16. The Kosaraju-Sharir strong components algorithm

Figure 6.17. The Kosaraju-Sharir algorithm in action. Top: Depth-first traversal of the reversed graph. Bottom: Depth-first traversal of the original graph, visiting root vertices in reversed postorder from the first traversal.

♥Tarjan's Algorithm

An earlier but considerably more subtle linear-time algorithm to compute strong components was published by Bob Tarjan in 1972.[6] Intuitively, Tarjan's algorithm identifies a *source* component of G, "deletes" it, and then "recursively" finds the remaining strong components; however, the entire computation happens during a single depth-first search.

Fix an arbitrary depth-first traversal of some directed graph G. For each vertex v, let $\boldsymbol{low(v)}$ denote the smallest *starting time* among all vertices reachable from v by a path of tree edges followed by *at most one* non-tree edge. Trivially, $low(v) \leq v.pre$, because v can reach itself through zero tree edges followed by zero non-tree edges. Tarjan observed that sink components can be characterized in terms of this *low* function.

Lemma 6.4. *A vertex v is the root of a sink component of G if and only if $low(v) = v.pre$ and $low(w) < w.pre$ for every proper descendant w of v.*

Proof: First, let v be a vertex such that $low(v) = v.pre$. Then there is no edge $w \to x$ where w is a descendant of v and $x.pre < v.pre$. On the other hand, v cannot reach any vertex y such that $y.pre > v.post$. It follows that v can reach only its descendants, and therefore any descendant of v can reach only descendants of v. In particular, v cannot reach its parent (if it has one), so v is the root of its strong component.

Now suppose in addition that $low(w) < w.pre$ for every descendant w of v. Then each descendant w can reach another vertex x (which must be another descendant of v) such that $x.pre < w.pre$. Thus, by induction, every descendant of v can reach v. It follows that the descendants of v comprise the strong component C whose root is v. Moreover, C must be a sink component, because v cannot reach any vertex outside of C.

On the other hand, suppose v is the root of a sink component C. Then v can reach another vertex w if and only if $w \in C$. But v can reach all of its descendants, and every vertex in C is a descendant of v, so v's descendants comprise C. If $low(w) = w.pre$ for any other node $w \in C$, then w would be another root of C, which is impossible. □

Computing $low(v)$ for every vertex v via depth-first search is straightforward; see Figure 6.18.

Lemma 6.4 implies that after running FindLow, we can identify the root of *every* sink component in $O(V + E)$ time (by a global whatever-first search),

[6] According to legend, Kosaraju apparently discovered his algorithm *during* an algorithms lecture. He was supposed to present Tarjan's algorithm, but he forgot his notes, so he had to make up something else on the fly. The only aspect of this story that I find surprising is that nobody tells it about Sharir or Tarjan.

```
FindLow(G):
  clock ← 0
  for all vertices v
      unmark v
      for all vertices v
          if v is unmarked
              FindLowDFS(v)
```

```
FindLowDFS(v):
  mark v
  clock ← clock + 1
  v.pre ← clock
  v.low ← v.pre
  for each edge v→w
      if w is unmarked
          FindLowDFS(w)
          v.low ← min{v.low, w.low}
      else
          v.low ← min{v.low, w.pre}
```

Figure 6.18. Computing *low*(v) for every vertex v.

and then mark and delete those sink components in $O(V + E)$ additional time (by calling whatever-first search at each root), and then recurse. Unfortunately, the resulting algorithm might require V iterations, each removing only a single vertex, naively giving us a total running time of $O(VE)$.

To speed up this strategy, Tarjan's algorithm maintains an auxiliary stack of vertices (separate from the recursion stack). Whenever we start a new vertex v, we push it onto the stack. Whenever we finish a vertex v, we compare $v.low$ with $v.pre$. Then the *first* time we discover that $v.low = v.pre$, we know three things:

- Vertex v is the root of a sink component C.
- All vertices in C appear consecutively at the top of the auxiliary stack.
- The *deepest* vertex in C on the auxiliary stack is v.

At this point, we can identify the vertices in C by popping them off the auxiliary stack one by one, stopping when we pop v.

We could delete the vertices in C and recursively compute the strong components of the remaining graph, but that would be wasteful, because we would repeat *verbatim* all computation done before visiting v. Instead, we *label* each vertex in C, identifying v as the root of its strong component, and then ignore labeled vertices for the rest of the depth-first search. Formally, this modification changes the definition of *low*(v) to the smallest starting time among all vertices ***in the same strong component as v*** that v can reach by a path of tree edges followed by at most one non-tree edge. But to prove correctness, it's easier to observe that ignoring labeled vertices leads the algorithm to exactly the same behavior as actually deleting them.

Finally, Tarjan's algorithm is shown in Figure 6.19, with the necessary modifications from FindLow (Figure 6.18) indicated in bold red. The running time of the algorithm can be split into two parts. Each vertex is pushed onto S once and popped off S once, so the total time spent maintaining the auxiliary stack (the red stuff) is $O(V)$. If we ignore the auxiliary stack maintenance, the

rest of the algorithm is just a standard depth-first search. We conclude that the algorithm runs in $\boldsymbol{O(V+E)}$ ***time***.

```
TarjanDFS(v):
  mark v
  clock ← clock + 1
  v.pre ← clock
  v.low ← v.pre
  Push(S, v)
  for each edge v→w
      if w is unmarked
          TarjanDFS(w)
          v.low ← min{v.low, w.low}
      else if w.root = None
          v.low ← min{v.low, w.pre}
  if v.low = v.pre
      repeat
          w ← Pop(S)
          w.root ← v
      until w = v
```

```
Tarjan(G):
  clock ← 0
  S ← new empty stack
  for all vertices v
      unmark v
      v.root ← None
  for all vertices v
      if v is unmarked
          TarjanDFS(v)
```

Figure 6.19. Tarjan's strong components algorithm.

Exercises

Depth-first search, topological sort, and strong components

0. (a) Describe an algorithm to compute the reversal $rev(G)$ of a directed graph in $O(V+E)$ time.

   (b) Prove that for every directed graph G, the strong component graph $scc(G)$ is acyclic.

   (c) Prove that $scc(rev(G)) = rev(scc(G))$ for every directed graph G.

   (d) Fix an arbitrary directed graph G. For any vertex v of G, let $S(v)$ denote the strong component of G that contains v. For all vertices u and v of G, prove that u can reach v in G if and only if $S(u)$ can reach $S(v)$ in $scc(G)$.

1. A directed graph G is *semi-connected* if, for every pair of vertices u and v, either u is reachable from v or v is reachable from u (or both).

   (a) Give an example of a directed acyclic graph with a unique source that is *not* semi-connected.

   (b) Describe and analyze an algorithm to determine whether a given directed *acyclic* graph is semi-connected.

(c) Describe and analyze an algorithm to determine whether an arbitrary directed graph is semi-connected.

2. The police department in the city of Sham-Poobanana has made every street in the city one-way. Despite widespread complaints from confused motorists, the mayor claims that it is possible to legally drive from any intersection in Sham-Poobanana to any other intersection.

   (a) The city needs to either verify or refute the mayor's claim. Formalize this problem in terms of graphs, and then describe and analyze an algorithm to solve it.

   (b) After running your algorithm from part (a), the mayor reluctantly admits that she was ~~lying~~ misinformed. Call an intersection x *good* if, for any intersection y that one can legally reach from x, it is possible to legally drive from y back to x. Now the mayor claims that over 95% of the intersections in Sham-Poobanana are good. Describe and analyze an efficient algorithm to verify or refute her claim.

   For full credit, both algorithms should run in linear time.

3. Suppose we are given a directed acyclic graph G with a unique source s and a unique sink t. A vertex $v \notin \{s, t\}$ is called an ***(s, t)-cut vertex*** if every path from s to t passes through v, or equivalently, if deleting v makes t unreachable from s. Describe and analyze an algorithm to find every (s, t)-cut vertex in G.

Figure 6.20. A directed acyclic graph with three (s, t)-cut vertices.

4. A vertex v in a connected undirected graph G is called a ***cut vertex*** if the subgraph $G - v$ (obtained by removing v from G) is disconnected.

Figure 6.21. An undirected graph with four cut vertices.

(a) Describe a linear-time algorithm that determines, given an undirected graph G and a vertex v, whether v is a cut vertex in G. What is the running time to find all cut vertices by trying your algorithm for each vertex?

(b) Let T be a depth-first spanning tree of an undirected graph G.

i. Prove that the root of T is a cut vertex of G if and only if it has more than one child in T.

ii. Prove that a non-root vertex v is a cut vertex of G if and only if at least one descendant (in T) of each child of v (in T) is a neighbor (in G) of some proper ancestor of v (in T).

[Hint: These claims no longer hold if T not a depth-first spanning tree and/or G is a directed graph.]

(c) Describe an algorithm that identifies every cut vertex in a given undirected graph in $O(V + E)$ time.

5. An edge e in a connected undirected graph G is called a *bridge* (or a *cut edge*) if the subgraph $G - e$ (obtained by removing e from G) is disconnected.

(a) Given G and edge e describe a linear-time algorithm that determines whether e is a bridge or not. What is the running time to find all bridges by trying your algorithm for each edge?

(b) Let T be an arbitrary spanning tree of G. Prove that *every* bridges of G is also an edge in T. This claim implies that G has at most $V - 1$ bridges. How does this information improve your algorithm from part (a) to find all bridges?

(c) Now suppose we root T at an arbitrary vertex r. For any vertex v, let T_v denote the subtree of T rooted at v; for example, $T_r = T$. Let uv be an arbitrary edge of T, where u is the parent of v. Prove that uv is a bridge of G if and only if uv is the only edge in G with exactly one endpoint in T_v.

(d) Describe a linear-time algorithm to identify every bridge in G. *[Hint: Let T be a depth-first spanning tree of G.]*

6. The ***transitive closure*** $\boldsymbol{G^T}$ of a directed graph G is a directed graph with the same vertices as G, that contains any edge $u \to v$ if and only if there is a directed path from u to v in G. A ***transitive reduction*** of G is a graph with the smallest possible number of edges whose transitive closure is G^T. The same graph may have several transitive reductions.

(a) Describe an efficient algorithm to compute the transitive closure of a given directed graph.

(b) Prove that a directed graph G has a *unique* transitive reduction if and only if G is acyclic.

(c) Describe an efficient algorithm to compute a transitive reduction of a given directed graph.

7. One of the oldest algorithms for exploring arbitrary connected graphs was proposed by Gaston Tarry in 1895, as a systematic procedure for solving mazes.[7] The input to Tarry's algorithm is an undirected graph G; however, for ease of presentation, we formally split each undirected edge uv into two directed edges $u \to v$ and $v \to u$. (In an actual implementation, this split is trivial; the algorithm simply uses the given adjacency list for G *as though* G were directed.)

```
TARRY(G):
  unmark all vertices of G
  color all edges of G white
  s ← any vertex in G
  RECTARRY(s)
```

```
RECTARRY(v):
  mark v                                  ⟨⟨"visit v"⟩⟩
  if there is a white arc v→w
      if w is unmarked
          color w→v green
      color v→w red
      RECTARRY(w)                         ⟨⟨"traverse v→w"⟩⟩
  else if there is a green arc v→w
      color v→w red
      RECTARRY(w)                         ⟨⟨"traverse v→w"⟩⟩
```

We informally say that Tarry's algorithm "visits" vertex v every time it marks v, and it "traverses" edge $v \to w$ when it colors that edge red and recursively calls RECTARRY(w). Unlike our earlier graph traversal algorithm, Tarry's algorithm can mark same vertex multiple times.

(a) Describe how to implement Tarry's algorithm so that it runs in $O(V + E)$ time.

(b) Prove that no directed edge in G is traversed more than once.

(c) When the algorithm visits a vertex v for the kth time, exactly how many edges into v are red, and exactly how many edges out of v are red? *[Hint: Consider the starting vertex s separately from the other vertices.]*

(d) Prove each vertex v is visited at most $\deg(v)$ times, except the starting vertex s, which is visited at most $\deg(s)+1$ times. This claim immediately implies that TARRY(G) terminates.

(e) Prove that the last vertex visited by TARRY(G) is the starting vertex s.

[7] Even older graph-traversal algorithms were described by Charles Trémaux in 1882, by Christian Wiener in 1873, by Carl Hierholzer in 1873, and (implicitly) by Leonhard Euler in 1736. In particular, Wiener's algorithm is equivalent to depth-first search in connected undirected graphs.

(f) For every vertex v that Tarry(G) visits, prove that all edges into v and out of v are red when Tarry(G) halts. *[Hint: Consider the vertices in the order that they are marked for the first time, starting with s, and prove the claim by induction.]*

(g) Prove that Tarry(G) visits every vertex of G. This claim and the previous claim imply that Tarry(G) traverses every edge of G exactly once.

8. Consider the following variant of Tarry's graph-traversal algorithm; this variant traverses green edges without recoloring them red and assigns two numerical labels to every vertex:

```
Tarry2(G):
  unmark all vertices of G
  color all edges of G white
  s ← any vertex in G
  RecTarry2(s, 1)
```

```
RecTarry2(v, clock):
  if v is unmarked
      v.pre ← clock;  clock ← clock + 1
      mark v
  if there is a white arc v→w
      if w is unmarked
          color w→v green
      color v→w red
      RecTarry2(w, clock)
  else if there is a green arc v→w
      v.post ← clock;  clock ← clock + 1
      RecTarry2(w, clock)
```

Prove or disprove the following claim: When Tarry2(G) halts, the green edges define a spanning tree and the labels $v.pre$ and $v.post$ define a preorder and postorder labeling that are all consistent with a single depth-first search of G. In other words, prove or disprove that Tarry2 produces the same *output* as depth-first search, even though it visits the edges in a completely different order.

9. You have a collection of n lock-boxes and m gold keys. Each key unlocks *at most* one box. However, each box might be unlocked by one key, by multiple keys, or by no keys at all. There are only two ways to open each box once it is locked: Unlock it properly (which requires having one matching key in your hand), or smash it to bits with a hammer.

Your baby brother, who loves playing with shiny objects, has somehow managed to lock all your keys inside the boxes! Luckily, your home security system recorded everything, so you know exactly which keys (if any) are inside each box. You need to get all the keys back out of the boxes, because they are made of gold. Clearly you have to smash at least one box.

(a) Your baby brother has found the hammer and is eagerly eyeing one of the boxes. Describe and analyze an algorithm to determine if it is

possible to retrieve all the keys without smashing any box except the one your brother has chosen.

(b) Describe and analyze an algorithm to compute the minimum number of boxes that must be smashed to retrieve all the keys.

10. Suppose you are teaching an algorithms course. In your second midterm, you give your students a drawing of a graph and ask then to indicate a breadth-first search tree and a depth-first search tree rooted at a particular vertex. Unfortunately, once you start grading the exam, you realize that the graph you gave the students has several such spanning trees—far too many to list. Instead, you need a way to tell whether each student's submission is correct!

In each of the following problems, suppose you are given a connected graph G, a start vertex s, and a spanning tree T of G.

(a) Suppose G is *undirected*. Describe and analyze an algorithm to decide whether T is a *depth*-first spanning tree rooted at s.

(b) Suppose G is *undirected*. Describe and analyze an algorithm to decide whether T is a *breadth*-first spanning tree rooted at s. *[Hint: It's not enough for T to be an unweighted shortest-path tree. Yes, this is the right chapter for this problem!]*

(c) Suppose G is *directed*. Describe and analyze an algorithm to decide whether T is a *breadth*-first spanning tree rooted at s. *[Hint: Solve part (b) first.]*

(d) Suppose G is *directed*. Describe and analyze an algorithm to decide whether T is a *depth*-first spanning tree rooted at s.

11. Several modern programming languages, including JavaScript, Python, Perl, and Ruby, include a feature called ***parallel assignment***, which allows multiple assignment operations to be encoded in a single line of code. For example, the Python code `x,y = 0,1` simultaneously sets `x` to 0 and `y` to 1. The values of the right-hand side of the assignment are all determined by the *old* values of the variables. Thus, the Python code `a,b = b,a` swaps the values of `a` and `b`, and the following Python code computes the nth Fibonacci number:

```
def fib(n):
    prev, curr = 1, 0
    while n > 0:
        prev, curr, n = curr, prev+curr, n-1
    return curr
```

Suppose the interpreter you are writing needs to convert every parallel assignment into an equivalent sequence of individual assignments. For

example, the parallel assignment `a,b = 0,1` can be serialized in either order—either `a=0; b=1` or `a=0; b=1`—but the parallel assignment `x,y = x+1,x+y` can only be serialized as `y=x+y; x=x+1`. Serialization may require one or more additional temporary variables; for example, serializing `a,b = b,a` requires one temporary variable, and serializing `x,y = x+y,x-y` requires two temporary variables.

(a) Describe an algorithm to determine whether a given parallel assignment can be serialized without additional temporary variables.

(b) Describe an algorithm to determine whether a given parallel assignment can be serialized with *exactly one* additional temporary variable.

Assume that the given parallel assignment involves only simple integer variables (no indirection via pointers or arrays); no variable appears on the left side more than once; and expressions on the right side have no side effects. Don't worry about the details of parsing the assignment statement; just assume (but describe!) an appropriate graph representation.

Dynamic Programming

12. Suppose we are given a directed acyclic graph G whose nodes represent jobs and whose edges represent precedence constraints; that is. each edge $u \to v$ indicates the job u must be completed before job v begins. Each node v also has a weight $T(v)$ indicating the time required to execute job v.

    (a) Describe an algorithm to determine the shortest interval of time in which all jobs in G can be executed.

    (b) Suppose the first job starts at time 0. Describe an algorithm to determine, for each vertex v, the earliest time when job v can begin.

    (c) Now describe an algorithm to determine, for each vertex v, the *latest* time when job v can begin without violating the precedence constraints or increasing the overall completion time (computed in part (a)), assuming that every job except v starts at its earliest start time (computed in part (b)).

13. Let G be a directed acyclic graph with a unique source s and a unique sink t.

    (a) A *Hamiltonian path* in G is a directed path in G that contains every vertex in G. Describe an algorithm to determine whether G has a Hamiltonian path.

    (b) Suppose the *vertices* of G have weights. Describe an efficient algorithm to find the path from s to t with maximum total weight.

(c) Suppose we are also given an integer ℓ. Describe an efficient algorithm to find the maximum-weight path from s to t that contains at most ℓ edges. (Assume there is at least one such path.)

(d) Suppose some of the vertices of G are marked as *important*, and we are also given an integer k. Describe an efficient algorithm to find the maximum-weight path from s to t that visits at least k important vertices. (Assume there is at least one such path.)

(e) Describe an algorithm to compute the number of paths from s to t in G. (Assume that you can add arbitrarily large integers in $O(1)$ time.)

14. Let G be a directed acyclic graph whose vertices have labels from some fixed alphabet, and let $A[1..\ell]$ be a string over the same alphabet. Any directed path in G has a label, which is a string obtained by concatenating the labels of its vertices.

(a) Describe an algorithm that either finds a path in G whose label is A or correctly reports that there is no such path.

(b) Describe an algorithm to find the *number* of paths in G whose label is A. (Assume that you can add arbitrarily large integers in $O(1)$ time.)

(c) Describe an algorithm to find the longest path in G whose label is a subsequence of A.

(d) Describe an algorithm to find the *shortest* path in G whose label is a *supersequence* of A.

(e) Describe an algorithm to find a path in G whose label has minimum edit distance from A.

15. A ***polygonal path*** is a sequence of line segments joined end-to-end; the endpoints of these line segments are called the ***vertices*** of the path. The ***length*** of a polygonal path is the sum of the lengths of its segments. A polygonal path with vertices $(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)$ is ***monotonically increasing*** if $x_i < x_{i+1}$ and $y_i < y_{i+1}$ for every index i—informally, each vertex of the path is above and to the right of its predecessor.

Figure 6.22. A monotonically increasing polygonal path with seven vertices through a set of points

Suppose you are given a set S of n points in the plane, represented as two arrays $X[1..n]$ and $Y[1..n]$. Describe and analyze an algorithm to compute the length of the longest monotonically increasing path with vertices in S. Assume you have a subroutine $\textsc{Length}(x, y, x', y')$ that returns the length of the segment from (x, y) to (x', y').

16. For any two nodes u and w in a directed acyclic graph G, the ***interval*** $\boldsymbol{G[u, w]}$ is the union of all directed paths in G from u to v. Equivalently, $G[u, w]$ consists of all vertices v such that $v \in \mathit{reach}(u)$ and $w \in \mathit{reach}(x)$, together with all the edges in G connecting those vertices.

    Suppose we are given a directed acyclic graph G, in which every vertex has a numerical weight, which may be positive, negative, or zero.

    (a) Describe an efficient algorithm to find the maximum-weight interval in G, where the weight of each interval is the sum of the weights of its vertices.

    (b) Describe an efficient algorithm to find the largest vertex weight in every interval in G. Your algorithm should compute a two-dimensional array $\mathit{MaxWt}[1..V, 1..V]$ where each entry $\mathit{MaxWt}[u, w]$ is the maximum weight among all vertices in the interval $G[u, w]$. In particular, if $G[u, w]$ is empty, then $\mathit{MaxWt}[u, w]$ should be $-\infty$.

17. Let G be a directed acyclic graph whose vertices have labels from some fixed alphabet. Any directed path in G has a label, which is a string obtained by concatenating the labels of its vertices. Recall that a *palindrome* is a string that is equal to its reversal.

    (a) Describe and analyze an algorithm to find the length of the longest palindrome that is the label of a path in G. For example, given the graph in Figure 6.23, your algorithm should return the integer 6, which is the length of the palindrome HANNAH.

Figure 6.23. A dag whose longest palindrome path label has length 6.

(b) Describe an algorithm to find the longest palindrome that is a subsequence of the label of a path in G.

(c) Suppose G has a single source s and a single sink t. Describe an algorithm to find the shortest palindrome that is a supersequence of the label of a path in G from s to t.

18. Suppose you are given two directed acyclic graphs G and H in which every node has a *label* from some finite alphabet; different nodes may have the same label. The label of a *path* in either dag is the string obtained by concatenating the labels of its vertices.

(a) Describe and analyze an algorithm to compute the length of the longest string that is both the label of a path in G and the label of a path in H.

(b) Describe and analyze an algorithm to compute the length of the longest string that is both a subsequence of the label of a path in G and a subsequence of the label of a path in H.

(c) Describe and analyze an algorithm to compute the length of the shortest string that is both a supersequence of the label of a path in G and a supersequence of the label of a path in H. *[Hint: This is easier than it looks.]*

19. Let G be an arbitrary (*not* necessarily acyclic) directed graph in which every vertex v has an integer weight $w(v)$.

(a) Describe an algorithm to find the longest directed path in G whose vertex weights define an increasing sequence.

(b) Describe and analyze an algorithm to determine the maximum-weight vertex reachable from each vertex in G. That is, for each vertex v, your algorithm needs to compute $\mathit{maxreach}(v) := \max\{w(x) \mid x \in \mathit{reach}(v)\}$.

20. (a) Suppose you are given a directed acyclic graph G with n vertices and an integer $k \leq n$. Describe an efficient algorithm to find a set of at most k vertex-disjoint paths that visit every vertex in G.

(b) Now suppose the edges of the input dag G have weights, which may be positive, negative, or zero. Describe an efficient algorithm to find a set of at most k vertex-disjoint paths *with minimum total weight* that visit every vertex in G.

Your algorithms should run in $O(n^{k+c})$ time for some small constant c. A single vertex is a path with weight zero. (We will see a more efficient algorithm for part (a) in Chapter 11.)

21. Kris is a professional rock climber who is competing in the U.S. climbing nationals. The competition requires Kris to use as many holds on the climbing wall as possible, using only transitions that have been explicitly allowed by the route-setter.

    The climbing wall has n holds. Kris is given a list of m pairs (x, y) of holds, each indicating that moving directly from hold x to hold y is allowed; however, moving directly from y to x is not allowed unless the list also includes the pair (y, x). Kris needs to figure out a sequence of allowed transitions that uses as many holds as possible, since each new hold increases his score by one point. The rules allow Kris to choose the first and last hold in his climbing route. The rules also allow him to use each hold as many times as he likes; however, only the first use of each hold increases Kris's score.

    (a) Define the natural graph representing the input. Describe and analyze an algorithm to solve Kris's climbing problem if you are guaranteed that the input graph is a dag.

    (b) Describe and analyze an algorithm to solve Kris's climbing problem with no restrictions on the input graph.

    Both of your algorithms should output the maximum possible score that Kris can earn.

22. There are n galaxies connected by m intergalactic teleport-ways. Each teleport-way joins two galaxies and can be traversed in both directions. However, the company that runs the teleport-ways has established an extremely lucrative cost structure: Anyone can teleport *further* from their home galaxy at no cost whatsoever, but teleporting *toward* their home galaxy is prohibitively expensive.

    Judy has decided to take a sabbatical tour of the universe by visiting as many galaxies as possible, starting at her home galaxy. To save on travel expenses, she wants to teleport away from her home galaxy at every step, except for the very last teleport home.

    (a) Describe and analyze an algorithm to compute the maximum number of galaxies that Judy can visit. Your input consists of an undirected graph G with n vertices and m edges describing the teleport-way network, an integer $1 \leq s \leq n$ identifying Judy's home galaxy, and an array $D[1..n]$ containing the distances of each galaxy from s.

    ♥(b) Just before embarking on her universal tour, Judy wins the space lottery, giving her just enough money to afford *two* teleports toward her home galaxy. Describe a new algorithm to compute the maximum number of distinct galaxies Judy can visit. She can visit the same galaxy more than once, but crucially, only the first visit counts toward her total.

23. The Doctor and River Song decide to play a game on a directed acyclic graph G, which has one source s and one sink t.[8]

    Each player has a token on one of the vertices of G. At the start of the game, The Doctor's token is on the source vertex s, and River's token is on the sink vertex t. The players alternate turns, with The Doctor moving first. On each of his turns, the Doctor moves his token forward along a directed edge; on each of her turns, River moves her token *backward* along a directed edge.

    If the two tokens ever meet on the same vertex, River wins the game. ("Hello, Sweetie!") If the Doctor's token reaches t or River's token reaches s before the two tokens meet, then the Doctor wins the game.

    Describe and analyze an algorithm to determine who wins this game, assuming both players play perfectly. That is, if the Doctor can win *no matter how River moves*, then your algorithm should output "Doctor", and if River can win *no matter how the Doctor moves*, your algorithm should output "River". (Why are these the only two possibilities?) The input to your algorithm is the graph G.

♣♥24. Let $x = x_1 x_2 \ldots x_n$ be a given n-character string over some finite alphabet Σ, and let A be a deterministic finite-state machine with m states over the same alphabet.

    (a) Describe and analyze an algorithm to compute the length of the longest subsequence of x that is accepted by A. For example, if A accepts the language $(\texttt{AR})^*$ and $x = \texttt{ABRACADABRA}$, your algorithm should output the number 4, which is the length of the subsequence ARAR.

    (b) Describe and analyze an algorithm to compute the length of the shortest supersequence of x that is accepted by A. For example, if A accepts the language $(\texttt{ABCDR})^*$ and $x = \texttt{ABRACADABRA}$, your algorithm should output the number 25, which is the length of the supersequence ABCDRABCDRABCDRABCDRABCDR.

    Analyze your algorithms in terms of the length n of the input string, the number m of states in the finite-state machine, and the size of the alphabet Σ.

25. Not *every* dynamic programming algorithm can be modeled as finding an optimal path through a directed acyclic graph, but every dynamic programming algorithm does process some underlying dependency graph in postorder.

[8] The labels s and t are abbreviations for the Untempered **S**chism and the **T**ime Vortex, or the Shining World of the Seven Systems (also known as Gallifrey) and Trenzalore, or Skaro and Telos, or Something else Timey-wimey. It's all very complicated, never mind.

(a) Suppose we are given a directed acyclic graph G where every node stores a numerical search key. Describe and analyze an algorithm to find the largest binary search tree that is a subgraph of G.

(b) Suppose we are given a directed acyclic graph G and two vertices s and t. Describe an algorithm to compute the number of directed paths in G from s to t. (Assume that any arithmetic operation requires $O(1)$ time.)

(c) Let G be a directed acyclic graph with the following features:

- G has a single source s and several sinks $t_1, t_2, \ldots, t_k$.
- Each edge $v \to w$ has an associated weight $p(v \to w)$ between 0 and 1.
- For each non-sink vertex v, the total weight of all edges leaving v is 1; that is, $\sum_w p(v \to w) = 1$.

The weights $p(v \to w)$ define a random walk in G from the source s to some sink t_i; after reaching any non-sink vertex v, the walk follows edge $v \to w$ with probability $p(v \to w)$. All probabilities are mutually independent. Describe and analyze an algorithm to compute the probability that this random walk reaches sink t_i, for every index i. (Assume that each arithmetic operation takes only $O(1)$ time.)

We must all hang together, gentlemen,
or else we shall most assuredly hang separately.

— Benjamin Franklin, at the signing of the
Declaration of Independence (July 4, 1776)

I remember seeking advice from someone—who could it have been?—about whether this work was worth submitting for publication; the reasoning it uses is so very simple.... Fortunately he advised me to go ahead, and many years passed before another of my publications became as well-known as this very simple one.

— Joseph Kruskal, describing his shortest-spanning-subtree algorithm (1997)

Clean ALL the things!

— Allie Brosh, "This is Why I'll Never be an Adult",
Hyperbole and a Half, June 17, 2010.

7

Minimum Spanning Trees

Suppose we are given a connected, undirected, *weighted* graph. This is a graph $G = (V, E)$ together with a function $w\colon E \to \mathbb{R}$ that assigns a real *weight* $w(e)$ to each edge e, which may be positive, negative, or zero. This chapter describes several algorithms to find the ***minimum spanning tree*** of G, that is, the spanning tree T that minimizes the function

$$w(T) := \sum_{e \in T} w(e).$$

See Figure 7.1 for an example.

7.1 Distinct Edge Weights

An annoying subtlety in the problem statement is that weighted graphs can have more than one spanning tree with the same minimum weight; in particular, if every edge in G has weight 1, then *every* spanning tree of G is a minimum

Figure 7.1. A weighted graph and its minimum spanning tree.

spanning tree, with weight $V-1$. This ambiguity complicates the development of our algorithms; everything would be much simpler if we could simply *assume* that minimum spanning trees are unique.

Fortunately, there is an easy condition that implies the uniqueness we want.

Lemma 7.1. *If all edge weights in a connected graph G are distinct, then G has a unique minimum spanning tree.*[1]

Proof: Let G be an arbitrary connected graph with two minimum spanning trees T and T'; we need to prove that some pair of edges in G have the same weight. The proof is essentially a greedy exchange argument.

Each of our spanning trees must contain an edge that the other tree omits. Let e be a minimum-weight edge in $T \setminus T'$, and let e' be a minimum-weight edge in $T' \setminus T$ (breaking ties arbitrarily). Without loss of generality, suppose $w(e) \leq w(e')$.

The subgraph $T' \cup \{e\}$ contains exactly one cycle C, which passes through the edge e. Let e'' be *any* edge of this cycle that is *not* in T. At least one such edge must exist, because T is a tree. (We may or may not have $e'' = e'$.) Because $e \in T$, we immediately have $e'' \neq e$ and therefore $e'' \in T' \setminus T$. It follows that $w(e'') \geq w(e') \geq w(e)$.

Now consider the spanning tree $T'' = T' + e - e''$. (This new tree T'' *might* be equal to T.) We immediately have $w(T'') = w(T') + w(e) - w(e'') \leq w(T')$. But T' is a *minimum* spanning tree, so we must have $w(T'') = w(T')$; in other words, T'' is also a minimum spanning tree. We conclude that $w(e) = w(e'')$, which completes the proof. □

If we already have an algorithm that assumes distinct edge weights, we can still run it on graphs where some edges have equal weights, as long as we have a consistent method for breaking ties. One such method uses the following

[1]The converse of this lemma is false; a connected graph with repeated edge weights can still have a unique minimum spanning tree. As a trivial example, suppose G is a tree!

algorithm in place of simple weight comparisons. ShorterEdge takes as input four integers i, j, k, l, representing four (not necessarily distinct) vertices, and decides which of the two edges (i, j) and (k, l) has "smaller" weight. (Because the input graph undirected, the pairs (i, j) and (j, i) represent the same edge.)

```
ShorterEdge(i, j, k, l)
  if w(i, j) < w(k, l)        then return (i, j)
  if w(i, j) > w(k, l)        then return (k, l)
  if min(i, j) < min(k, l)    then return (i, j)
  if min(i, j) > min(k, l)    then return (k, l)
  if max(i, j) < max(k, l)    then return (i, j)
  ⟨⟨if max(i,j) > max(k,l) ⟩⟩  return (k, l)
```

In light of Lemma 7.1 and this tie-breaking rule, we will safely assume for the rest of this chapter that edge weights are *always* distinct, and therefore minimum spanning trees are *always* unique. In particular, we can freely discuss ***the*** minimum spanning tree with no confusion.

7.2 The Only Minimum Spanning Tree Algorithm

There are many algorithms to compute minimum spanning trees, but almost all of them are instances of the following generic strategy. The situation is similar to graph traversal, where several different algorithms are all variants of the generic traversal algorithm whatever-first search.

The generic minimum spanning tree algorithm maintains an acyclic subgraph F of the input graph G, which we will call the *intermediate spanning forest*. At all times, F satisfies the following invariant:

$$\boxed{F \text{ is a subgraph of the minimum spanning tree of } G.}$$

Initially, F consists of V one-vertex trees. The generic algorithm connects trees in F by adding certain edges between them. When the algorithm halts, F consists of a single spanning tree; our invariant implies that this must be the minimum spanning tree of G. Obviously, we have to be careful about *which* edges we add to the evolving forest, because not every edge is in the minimum spanning tree.

At any stage of its evolution, the intermediate spanning forest F induces two special types of edges in the rest of the graph.

- An edge is ***useless*** if it is not an edge of F, but both its endpoints are in the same component of F.
- An edge is ***safe*** if it is the minimum-weight edge with exactly one endpoint in some component of F.

The same edge could be safe for two different components of F. Some edges of $G \setminus F$ are neither safe nor useless; we call these edges *undecided*.

All minimum spanning tree algorithms are based on two simple observations. The first observation was proved by Robert Prim in 1957 (although it is implicit in several earlier algorithms), and the second is immediate.

Lemma 7.2 (Prim). *The minimum spanning tree of G contains every safe edge.*

Proof: In fact we prove the following stronger statement: For *any* subset S of the vertices of G, the minimum spanning tree of G contains the minimum-weight edge with exactly one endpoint in S. Like the previous lemma, we prove this claim using a greedy exchange argument.

Let S be an arbitrary subset of vertices of G, and let e be the lightest edge with exactly one endpoint in S. (Our assumption that all edge weights are distinct implies that e is unique.) Let T be an arbitrary spanning tree that does *not* contain e; we need to prove that T is *not* the minimum spanning tree of G.

Because T is connected, it contains a path from one endpoint of e to the other. Because this path starts at a vertex of S and ends at a vertex not in S, it must contain at least one edge with exactly one endpoint in S; let e' be *any* such edge. Because T is acyclic, removing e' from T yields a spanning *forest* with exactly two components, one containing each endpoint of e. Thus, adding e to this forest gives us a new spanning tree $T' = T - e' + e$. The definition of e implies $w(e') > w(e)$, which implies that T' has smaller total weight than T. Thus, T is not the minimum spanning tree of G, which completes the proof. □

Figure 7.2. Every safe edge is in the minimum spanning tree. Black vertices are in the subset S.

Lemma 7.3. *The minimum spanning tree contains no useless edge.*

Proof: Adding any useless edge to F would introduce a cycle. □

Our generic minimum spanning tree algorithm repeatedly adds *safe* edges to the evolving forest F. If F is not yet connected, there must be at least one safe edge, because the input graph G is connected. Thus, no matter which safe edges we add in each iteration, our generic algorithm eventually connects F. By induction, Lemma 7.2 implies that the resulting tree is in fact the minimum

spanning tree. Whenever we add new edges to F, some undecided edges may become safe, and other undecided edges may become useless. (Once an edge becomes useless, it stays useless forever.) To fully specify a particular algorithm, we must describe *which* safe edge(s) to add in each iteration, and how to find those edges.

7.3 Borůvka's Algorithm

The oldest and arguably simplest minimum spanning tree algorithm was discovered by the Czech mathematician Otakar Borůvka in 1926, about a year after Jindřich Saxel asked him how to construct an electrical network connecting several cities using the least amount of wire.[2] The algorithm was rediscovered by Gustav Choquet in 1938, rediscovered again by a team of Polish mathematicians led by Józef Łukaszewicz in 1951, and rediscovered again by George Sollin in 1961. Although Sollin never published his rediscovery, it was carefully described and credited in one of the first textbooks on graph algorithms; as a result, this algorithm is sometimes called "Sollin's algorithm".

The Borůvka / Choquet / Florek-Łukaziewicz-Perkal-Steinhaus-Zubrzycki / Prim / Sollin / Brosh[3] algorithm can be summarized in one line:

BORŮVKA: Add ***ALL*** the safe edges and recurse.

Figure 7.3. Borůvka's algorithm run on the example graph. Thick red edges are in F; dashed edges are useless. Arrows point along each component's safe edge. The algorithm ends after just two iterations.

Here is Borůvka's algorithm in more detail. The algorithm calls the COUNTANDLABEL algorithm from Chapter 5 (on page 204) to count the components of F and label each vertex v with an integer $comp(v)$ indicating its component.

[2]Saxel was an employee of the West Moravian Power Company, described by Borůvka as "very talented and hard-working", who was later executed by the Nazis as a person of Jewish descent.

[3]Go read everything in *Hyperbole and a Half*. And then go buy the book. And an extra copy for your cat. What's that? You don't have a cat? What kind of a monster are you? Go get a cat, and then buy it an extra copy of *Hyperbole and a Half*.

```
BORŮVKA(V, E):
  F = (V, ∅)
  count ← COUNTANDLABEL(F)
  while count > 1
      ADDALLSAFEEDGES(E, F, count)
      count ← COUNTANDLABEL(F)
  return F
```

It remains only to describe how to identify and add all the safe edges to F. Suppose F has more than one component, since otherwise we're already done. The following subroutine computes an array $safe[1..V]$ of safe edges, where $safe[i]$ is the minimum-weight edge with one endpoint in the ith component of F, by a brute force examination of every edge in G. For each edge uv, if u and v are in the same component, then uv is either useless or already an edge in F. Otherwise, we compare the weight of uv to the weights of $safe[comp(u)]$ and $safe[comp(v)]$ and update the array entries if necessary. Once we have identified all the safe edges, we add each edge $safe[i]$ to F.

```
ADDALLSAFEEDGES(E, F, count):
  for i ← 1 to count
      safe[i] ← NULL
  for each edge uv ∈ E
      if comp(u) ≠ comp(v)
          if safe[comp(u)] = NULL  or  w(uv) < w(safe[comp(u)])
              safe[comp(u)] ← uv
          if safe[comp(v)] = NULL  or  w(uv) < w(safe[comp(v)])
              safe[comp(v)] ← uv
  for i ← 1 to count
      add safe[i] to F
```

Each call to COUNTANDLABEL runs in $O(V)$ time, because the forest F has at most $V-1$ edges. *AddAllSafeEdges* runs in $O(V+E)$ time, because we spend constant time on each vertex, each edge of G, and each component of F. Because the input graph is connected, we have $V \leq E+1$. It follows that each iteration of the while loop of BORŮVKA takes $O(E)$ time.

Each iteration reduces the number of components of F by at least a factor of two—in the worst case, the components of F coalesce in pairs. Because F initially has V components, the while loop iterates at most $O(\log V)$ times. We conclude that the overall running time of Borůvka's algorithm is $\boldsymbol{O(E \log V)}$.

This is the MST Algorithm You Want

Despite its relatively obscure origin, early Western algorithms researchers were aware of Borůvka's algorithm, but dismissed it as being "too complicated".

As a result, despite its simplicity and efficiency, most algorithms and data structures textbooks unfortunately do not even mention Borůvka's algorithm. This omission is a serious mistake; Borůvka's algorithm has several distinct advantages over other classical MST algorithms.

- Borůvka's algorithm often runs faster than its $O(E \log V)$ worst-case running time. The number of components in F can drop by significantly more than a factor of 2 in a single iteration, reducing the number of iterations below the worst-case $\lceil \log_2 V \rceil$.
- A slight reformulation of Borůvka's algorithm (actually closer to Borůvka's original presentation) actually runs in $O(E)$ time for a broad class of interesting graphs, including graphs that can be drawn in the plane without edge crossings. In contrast, the time analysis for the other two algorithms applies to *all* graphs.
- Borůvka's algorithm allows for significant parallelism; in each iteration, each component of F can be handled in a separate independent thread. This implicit parallelism allows for even faster performance on multicore or distributed systems. In contrast, the other two classical MST algorithms are intrinsically serial.
- Several more recent minimum-spanning-tree algorithms are faster even in the worst case than the classical algorithms described here. *All* of these faster algorithms are generalizations of Borůvka's algorithm.

In short, if you ever need to implement a minimum-spanning-tree algorithm, use Borůvka. On the other hand, if you want to *prove things about* minimum spanning trees effectively, you really need to know the next two algorithms as well.

7.4 Jarník's ("Prim's") Algorithm

The next oldest minimum spanning tree algorithm was first described by the Czech mathematician Vojtěch Jarník in a 1929 letter to Borůvka; Jarník published his discovery the following year. The algorithm was independently rediscovered by Joseph Kruskal in 1956, (arguably) by Robert Prim in 1957, by Harry Loberman and Arnold Weinberger in 1957, and finally by Edsger Dijkstra in 1958. Prim, Lobermand and Weinberger, and Dijkstra all (eventually) knew of and even cited Kruskal's paper, but since Kruskal also described two other minimum-spanning-tree algorithms in the same paper, *this* algorithm is usually called "Prim's algorithm", or sometimes "the Prim/Dijkstra algorithm", even though by 1958 Dijkstra already had another algorithm (inappropriately) named after him.

In Jarník's algorithm, the intermediate forest F has only one nontrivial component T; all the other components are isolated vertices. Initially, T consists

of a single arbitrary vertex of the graph. The algorithm repeats the following step until T spans the whole graph:

Jarník: Repeatedly add T's safe edge to T.

Figure 7.4. Jarník's algorithm run on the example graph, starting with the bottom vertex. At each stage, thick red edges are in T, an arrow points along T's safe edge; and dashed edges are useless.

To implement Jarník's algorithm, we keep all the edges adjacent to T in a priority queue. When we pull the minimum-weight edge out of the priority queue, we first check whether both of its endpoints are in T. If not, we add the edge to T and then add the new neighboring edges to the priority queue. In other words, Jarník's algorithm is a variant of "best-first search", as described at the end of Chapter 5! If we implement the underlying priority queue using a standard binary heap, Jarník's algorithm runs in $O(E \log E) = \boldsymbol{O(E \log V)}$ time.

♥Improving Jarník's Algorithm

We can improve Jarník's algorithm using a more complex priority queue data structure called a ***Fibonacci heap***, first described by Michael Fredman and Robert Tarjan in 1984. Just like binary heaps, Fibonacci heaps support the standard priority queue operations Insert, ExtractMin, and DecreaseKey. However, unlike standard binary heaps, which require $O(\log n)$ time for every operation, Fibonacci heaps support Insert and DecreaseKey in constant *amortized* time. The amortized cost of ExtractMin is still $O(\log n)$.[4]

[4]Amortized time is an accounting trick that allows us to ignore infrequent fluctuations in the time for a single data structure operation. A Fibonacci heap can execute any intermixed sequence of I Inserts, D DecreaseKeys, and X ExtractMins in $O(I + D + X \log n)$ time, in the worst case. So the *average* Insert and the *average* DecreaseKey each take constant time, and the *average* ExtractMin takes $O(\log n)$ time; however, some individual operations may take

To apply this faster data structure, we keep the *vertices* of G in the priority queue instead of edges, where the priority of each vertex v is either the minimum-weight edge between v and the evolving tree T, or ∞ if there is no such edge. We can INSERT all the vertices into the priority queue at the beginning of the algorithm; then, whenever we add a new edge to T, we may need to decrease the priorities of some neighboring vertices.

To make the description easier, we break the algorithm into two parts. JARNÍKINIT initializes the priority queue; JARNÍKLOOP is the main algorithm. The input consists of the vertices and edges of the graph, along with the start vertex s. For each vertex v, we maintain both its priority $priority(v)$ and the incident edge $edge(v)$ such that $w(edge(v)) = priority(v)$.

```
JARNÍK(V, E, s):
    JARNÍKINIT(V, E, s)
    JARNÍKLOOP(V, E, s)
```

```
JARNÍKINIT(V, E, s):
    for each vertex v ∈ V \ {s}
        if vs ∈ E
            edge(v) ← vs
            priority(v) ← w(vs)
        else
            edge(v) ← NULL
            priority(v) ← ∞
        INSERT(v)
```

```
JARNÍKLOOP(V, E, s):
    T ← ({s}, ∅)
    for i ← 1 to |V| − 1
        v ← EXTRACTMIN
        add v and edge(v) to T
        for each neighbor u of v
            if u ∉ T and priority(u) > w(uv)
                edge(u) ← uv
                DECREASEKEY(u, w(uv))
```

Figure 7.5. Jarník's minimum spanning tree algorithm, ready to be used with a Fibonacci heap

The operations INSERT and EXTRACTMIN are each called $O(V)$ times once for each vertex except s, and DECREASEKEY is called $O(E)$ times, at most twice for each edge. Thus, if we use a Fibonacci heap, the improved algorithm runs in $\boldsymbol{O(E + V \log V)}$ ***time***, which is faster than Borůvka's algorithm unless $E = O(V)$.

In practice, however, this improvement is rarely faster than the naive implementation using a binary heap, unless the graph is extremely large and dense. The Fibonacci heap algorithms are quite complex, and the hidden constants in both the running time and space are significant—not outrageous, but certainly bigger than the hidden constant 1 in the $O(\log n)$ time bound for binary heap operations.

7.5 Kruskal's Algorithm

The last minimum spanning tree algorithm we'll consider was first described by Joseph Kruskal in 1956, in the same paper where he rediscovered Jarnik's algo-

longer in the worst case. Amortization uses *statistical* averaging over the sequence of operations; there is no assumption of randomness here, either in the input data or in the algorithm.

rithm. Kruskal was motivated by "a typewritten translation (of obscure origin)" of Borůvka's original paper that had been "floating around" the Princeton math department. Kruskal found Borůvka's algorithm "unnecessarily elaborate".[5] The same algorithm was rediscovered in 1957 by Harry Loberman and Arnold Weinberger, but somehow avoided being renamed after them.

Like our earlier minimum-spanning tree algorithms, Kruskal's algorithm has a memorable one-line description:

Kruskal: Scan all edges by increasing weight; if an edge is safe, add it to F.

Figure 7.6. Kruskal's algorithm run on the example graph. Thick red edges are in F; thin dashed edges are useless.

The simplest method to scan the edges in increasing weight order is to *sort* the edges by weight, in $O(E \log E)$ time, and then use a simple for-loop over the sorted edge list. As we will see shortly, this preliminary sorting dominates the running time of the algorithm.

Because we examine the edges in order from lightest to heaviest, any edge we examine is safe if and only if its endpoints are in different components of the forest F. Suppose we encounter an edge e that joins two components A and B but is not safe. Then there must be a lighter edge e' with exactly one endpoint in A. But this is impossible, because (inductively) every previously examined edge has both endpoints in the same component of F.

Just as in Borůvka's algorithm, each vertex of F needs to "know" which component of F contains it. Unlike Borůvka's algorithm, however, we do

[5]To be fair, Borůvka's first paper *was* unnecessarily elaborate, in part because it was written for mathematicians in the formal language of (linear) algebra, rather than in the language of graphs. Borůvka's followup paper, also published in 1927 but in an electrotechnical journal, was written in plain language for a much broader audience, essentially in its current modern form. Kruskal was apparently unaware of Borůvka's second paper. Stupid Iron Curtain.

not recompute all component labels from scratch every time we add an edge. Instead, when two components are joined by an edge, the smaller component inherits the label of the larger component; that is, we traverse the smaller component (via whatever-first search). This traversal requires $O(1)$ time for each vertex in the smaller component. Each time the component label of a vertex changes, the component of F containing that vertex grows by at least a factor of 2; thus, each vertex label changes at most $O(\log V)$ times. It follows that the *total* time spent updating vertex labels is only $O(V \log V)$.

More generally, Kruskal's algorithm maintains a partition of the vertices of G into disjoint subsets (in our case, the components of F), using a data structure that supports the following operations:

- MakeSet(v) — Create a set containing only the vertex v.
- Find(v) — Return an identifier unique to the set containing v.
- Union(u, v) — Replace the sets containing u and v with their union. (This operation decreases the number of sets.)

Here's a complete description of Kruskal's algorithm in terms of these operations:

```
Kruskal(V, E):
  sort E by increasing weight
  F ← (V, ∅)
  for each vertex v ∈ V
      MakeSet(v)
  for i ← 1 to |E|
      uv ← ith lightest edge in E
      if Find(u) ≠ Find(v)
          Union(u, v)
          add uv to F
  return F
```

After the initial sort, the algorithm performs exactly V MakeSet operations (one for each vertex), $2E$ Find operations (two for each edge), and $V-1$ Union operations (one for each edge in the minimum spanning tree). We just described a disjoint-set data structure for which MakeSet and Find require $O(1)$ time, and Union runs in $O(\log V)$ *amortized* time. Using this implementation, the total time spent maintaining the set partition is $O(E + V \log V)$.[6]

But recall that we already need $O(E \log E) = O(E \log V)$ time just to sort the edges. Because this is larger than the time spent maintaining the Union-Find data structure, the overall running time of Kruskal's algorithm is $\boldsymbol{O(E \log V)}$,

[6] A different disjoint-set data structure, which uses a strategy called *union-by-rank with path compression*, performs each Union or Find in $O(\alpha(V))$ amortized time, where α is the almost-but-not-quite-constant *inverse Ackerman function*. If you don't feel like consulting Wikipedia, just think of $\alpha(V)$ as 4. Using this implementation, the total time spent maintaining the set partition is $O(E\alpha(V))$, which is slightly faster when V is large and E is very close to V.

exactly the same as Borůvka's algorithm, or Jarník's algorithm with a normal (non-Fibonacci) heap.

Exercises

1. Let $G = (V, E)$ be an arbitrary connected graph with weighted edges.
   (a) Prove that for any cycle in G, the minimum spanning tree of G *excludes* the maximum-weight edge in that cycle.
   (b) Prove or disprove: The minimum spanning tree of G includes the minimum-weight edge in *every* cycle in G.

2. Throughout this chapter, we assumed that no two edges in the input graph have equal weights, which implies that the minimum spanning tree is unique. In fact, a weaker condition on the edge weights implies MST uniqueness.
   (a) Describe an edge-weighted graph that has a unique minimum spanning tree, even though two edges have equal weights.
   (b) Prove that an edge-weighted graph G has a *unique* minimum spanning tree if and only if the following conditions hold:
      - For any partition of the vertices of G into two subsets, the minimum-weight edge with one endpoint in each subset is unique.
      - The maximum-weight edge in any cycle of G is unique.
   (c) Describe and analyze an algorithm to determine whether or not a graph has a unique minimum spanning tree.

3. Most classical minimum-spanning-tree algorithms use the notions of "safe" and "useless" edges described in the text, but there is an alternate formulation. Let G be a weighted undirected graph, where the edge weights are distinct. We say that an edge e is ***dangerous*** if it is the longest edge in some cycle in G, and ***useful*** if it does not lie in any cycle in G.
   (a) Prove that the minimum spanning tree of G contains every useful edge.
   (b) Prove that the minimum spanning tree of G does not contain any dangerous edge.
   (c) Describe and analyze an efficient implementation of the following algorithm, first described by Joseph Kruskal in the same 1956 paper where he proposed "Kruskal's algorithm". Examine the edges of G in *decreasing* order; if an edge is dangerous, remove it from G. *[Hint: It won't be as fast as Kruskal's usual algorithm.]*

4. (a) Describe and analyze an algorithm to compute the *maximum*-weight spanning tree of a given edge-weighted graph.

(b) A *feedback edge set* of an undirected graph G is a subset F of the edges such that every cycle in G contains at least one edge in F. In other words, removing every edge in F makes the graph G acyclic. Describe and analyze a fast algorithm to compute the minimum-weight feedback edge set of a given edge-weighted graph.

5. Suppose we are given both an undirected graph G with weighted edges and a minimum spanning tree T of G.

   (a) Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is decreased.

   (b) Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is increased.

   In both cases, the input to your algorithm is the edge e and its new weight; your algorithms should modify T so that it is still a minimum spanning tree. *[Hint: Consider the cases $e \in T$ and $e \notin T$ separately.]*

6. (a) Describe and analyze an algorithm to find the *second smallest spanning tree* of a given graph G, that is, the spanning tree of G with smallest total weight except for the minimum spanning tree.

   ♥(b) Describe and analyze an efficient algorithm to compute, given a weighted undirected graph G and an integer k, the k spanning trees of G with smallest weight.

7. A graph $G = (V, E)$ is *dense* if $E = \Theta(V^2)$. Describe a modification of Jarník's minimum-spanning tree algorithm that runs in $O(V^2)$ time (independent of E) when the input graph is dense, using only elementary data structures—in particular, *without* using Fibonacci heaps. This variant of Jarník's algorithm was first described by Edsger Dijkstra in 1958.

8. Minimum-spanning tree algorithms are often formulated using an operation called *edge contraction*. To contract the edge uv, we insert a new node, redirect any edge incident to u or v (except uv) to this new node, and then delete u and v. After contraction, there may be multiple parallel edges between the new node and other nodes in the graph; we remove all but the lightest edge between any two nodes.

   The three classical minimum-spanning tree algorithms described in this chapter can all be expressed cleanly in terms of contraction as follows. All three algorithms start by making a clean copy G' of the input graph G and then repeatedly contract safe edges in G'; the minimum spanning tree consists of the contracted edges.

Figure 7.7. Contracting an edge and removing redundant parallel edges.

- Borůvka: Mark the lightest edge leaving each vertex, contract all marked edges, and recurse.
- Jarník: Repeatedly contract the lightest edge incident to some fixed root vertex.
- Kruskal: Repeatedly contract the lightest edge in the graph.

(a) Describe an algorithm to execute a single pass of Borůvka's contraction algorithm in $O(V + E)$ time. The input graph is represented in an adjacency list.

(b) Consider an algorithm that first performs k passes of Borůvka's contraction algorithm, and then runs Jarník's algorithm (*with* a Fibonacci heap) on the resulting contracted graph.

  i. What is the running time of this hybrid algorithm, as a function of V, E, and k?

  ii. For which value of k is this running time minimized? What is the resulting running time?

(c) Call a family of graphs *nice* if it has the following properties:

  - Contracting an edge of a nice graph yields another nice graph.
  - Every nice graph with V vertices has only $O(V)$ edges.

  For example, planar graphs—graphs that can be drawn in the plane with no crossing edges—are nice. Contracting any edge of a planar graph leaves a smaller planar graph, and Euler's formula implies that every planar graph with V vertices has at most $3V - 6$ edges.

  Prove that Borůvka's contraction algorithm computes the minimum spanning tree of any nice graph in $O(V)$ time.

9. Consider a path between two vertices s and t in a undirected weighted graph G. The *width* of this path is the minimum weight of any edge in the path. The *bottleneck distance* between s and t is the width of the widest path from s to t. (If there are no paths from s to t, the bottleneck distance is $-\infty$; on the other hand, the bottleneck distance from s to itself is ∞.)

(a) Prove that the *maximum* spanning tree of G contains widest paths between *every* pair of vertices.

The bottleneck distance between s and t is 9.

(b) Describe an algorithm to solve the following problem in $O(V+E)$ time: Given a undirected weighted graph G, two vertices s and t, and a weight W, is the bottleneck distance between s and t at most W?

(c) Suppose B is the bottleneck distance between s and t.

i. Prove that deleting any edge with weight less than B does not change the bottleneck distance between s and t.

ii. Prove that *contracting* any edge with weight *greater* than B does not change the bottleneck distance between s and t. (If contraction creates parallel edges, delete all but the *heaviest* edge between each pair of nodes.)

♥(d) Describe an algorithm to compute a minimum-bottleneck path between s and t in $O(V+E)$ time. *[Hint: Start by finding the median-weight edge in G.]*

10. Borůvka's algorithm can be reformulated to use a standard disjoint-set data structure to identify safe edges, just like Kruskal's algorithm, instead of explicitly counting and labeling components of the evolving spanning forest F in each iteration.

In this variant, each component of F is represented by an *up-tree*; each vertex v stores a pointer *parent*(v) to its parent, or to v itself if v is the root of its up-tree. The subroutine FIND(v) returns the root of v's up-tree, but also applies *path compression*, reassigning all parent pointers from v to the root to point directly to the root, to speed up future FIND operations.[7] The subroutine UNION combines two up-trees into one by making one of the two root nodes the parent of the other.[8]

[7]Path compression is a form of memoization!

[8]Normally, UNION is implemented more carefully to ensure that the root of the larger or older up-tree does not change; however, those details don't matter here.

```
FIND(v):
  if parent(v) = v
      return v
  else
      v̄ ← FIND(parent(v))
      parent(v) ← v̄
      return v̄
```

```
UNION(u, v):
  ū ← FIND(u)
  v̄ ← FIND(v)
  either
      parent(ū) ← v̄
  or
      parent(v̄) ← ū
```

In the modified version of Borůvka's algorithm, in addition to the parent pointers, the root vertex $\bar{v}$ of each component of F maintains an edge $safe(\bar{v})$, which (at the end of FindSafeEdges) is the lightest edge with one endpoint in that component.

```
FINDSAFEEDGES(V, E):
  for each vertex v ∈ V
      safe(v) ← NULL
  found ← FALSE
  for each edge uv ∈ E
      ū ← FIND(u)
      v̄ ← FIND(v)
      if ū ≠ v̄
          if safe(ū) = NULL or w(uv) < w(safe(ū))
              safe(ū) ← uv
          if safe(v̄) = NULL or w(uv) < w(safe(v̄))
              safe(v̄) ← uv
          found ← TRUE
  return found
```

```
ADDSAFEEDGES(V, E, F):
  for each vertex v ∈ V
      if safe(v) ≠ NULL
          xy ← safe(v)
          if FIND(x) ≠ FIND(y)
              UNION(x, y)
              add xy to F
```

```
BORŮVKA(V, E):
  F = ∅
  for each vertex v ∈ V
      parent(v) ← v
  while FINDSAFEEDGES(V, E)
      ADDSAFEEDGES(V, E, F)
  return F
```

Prove that each call to FindSafeEdges and AddSafeEdges requires only $O(E)$ time. *[Hint: What is the depth of the up-trees when FindSafeEdges ends?]* It follows that this variant of Borůvka also runs in $O(E \log V)$ time.

I study my Bible as I gather apples. First I shake the whole tree, that the ripest might fall. Then I climb the tree and shake each limb, and then each branch and then each twig, and then I look under each leaf.

— attributed to Martin Luther (c. 1500)

Life is an unfoldment, and the further we travel the more truth we can comprehend. To understand the things that are at our door is the best preparation for understanding those that lie beyond.

— attributed to Hypatia of Alexandria (c. 400) by Elbert Hubbard in *Little Journeys to the Homes of Great Teachers* (1908)

Your mind will answer most questions if you learn to relax and wait for the answer. Like one of those thinking machines, you feed in your question, sit back, and wait . . .

— William S. Burroughs, *Naked Lunch* (1959)

The methods given in this paper require no foresight or ingenuity, and hence deserve to be called algorithms.

— Edward R. Moore, "The Shortest Path Through a Maze" (1959)

8

Shortest Paths

Suppose we are given a weighted *directed* graph $G = (V, E, w)$ with two special vertices, and we want to find the shortest path from a *source* vertex s to a *target* vertex t. That is, we want to find the directed path P starting at s and ending at t that minimizes the function

$$w(P) := \sum_{u \to v \in P} w(u \to v).$$

For example, if I want to answer the question "What's the fastest way to drive from my old apartment in Champaign, Illinois to my wife's old apartment in Columbus, Ohio?", I might use a graph whose vertices are cities, edges are roads, weights are driving times, s is Champaign, and t is Columbus.[1] The graph is directed, because driving times along the same road might be different

[1]West on Church, north on Prospect, east on I-74, south on I-465, east on Airport Expressway, north on I-65, east on I-70, north on Grandview, east on 5th, north on Olentangy River, east on Dodridge, north on High, west on Kelso, south on Neil. Depending on traffic. We live in Urbana now.

in different directions. (At one time, there was a speed trap on I-70 just east of the Indiana/Ohio border, but only for eastbound traffic.)

8.1 Shortest Path Trees

Almost every algorithm known for computing shortest paths from one vertex to another actually solves (large portions of) the following more general ***single source shortest path*** or ***SSSP*** problem: Find shortest paths from the source vertex s to *every* other vertex in the graph. This problem is usually solved by finding a ***shortest path tree*** rooted at s that contains all the desired shortest paths.

It's not hard to see that if shortest paths are unique, then they form a tree, because any subpath of a shortest path is itself a shortest path. If there are multiple shortest paths to some vertices, we can always choose one shortest path to each vertex so that the union of the paths is a tree. If there are shortest paths from s to two vertices u and v that diverge, then meet, then diverge again, we can modify one of the paths without changing its length, so that the two paths only diverge once.

Figure 8.1. If $s \to a \to b \to c \to d \to v$ (solid) and $s \to a \to x \to y \to d \to u$ (dashed) are shortest paths, then $s \to a \to b \to c \to d \to u$ (along the top) is also a shortest path.

Although they are both optimal spanning trees, shortest-path trees and minimum spanning trees are very different creatures. Shortest-path trees are rooted and directed; minimum spanning trees are unrooted and undirected. Shortest-path trees are most naturally defined for directed graphs; minimum spanning trees are more naturally defined for undirected graphs. If edge weights are distinct, there is only one minimum spanning tree, but every source vertex induces a different shortest-path tree; moreover, it is possible for *every* shortest path tree to use a different set of edges from the minimum spanning tree.

♥8.2 Negative Edges

For most shortest-path problems, where the edge weights correspond to distance or length or time, it is natural to assume that all edge weights are non-negative, or even positive. However, for many applications of shortest-path algorithms, it is natural to consider edges with negative weight. For example, the weight

Figure 8.2. A minimum spanning tree and a shortest path tree of the same undirected graph.

of an edge might represent the *cost* of moving from one vertex to another, so negative-weight edges represent transitions with negative cost, or equivalently, transitions that earn a profit.

Negative edges are a thorn in the side of most shortest-path problems, because the presence of a negative *cycle* might imply that shortest paths may not be well-defiend. To be precise, a shortest path from s to t exists if and only if there is at least one path from s to t, but there is no path from s to t that touches a negative cycle. For *any* path from s to t that touches a negative cycle, there is a shorter path from s to t that goes around the cycle one more time.[2] Thus, if at least one path from s to t touches a negative cycle, there is no shortest path from s to t.

Figure 8.3. There is no shortest walk from s to t.

In part because we need to consider negative edge weights, this chapter explicitly considers *only* directed graphs. All of the algorithms described here also work for undirected graphs with essentially trivial modifications, *if and only if* negative edges are prohibited. Correctly handling negative edges in undirected graphs is considerably more subtle. We cannot simply replace every undirected edge with a pair of directed edges, because this would transform any negative edge into a short negative cycle. Subpaths of an *undirected* shortest path that contains a negative edge are *not* necessarily shortest paths; consequently, the set of all undirected shortest paths from a single source vertex may not define a tree, even if shortest paths are unique.

[2]Technically, we should be discussing shortest *walks* here, rather than shortest *paths*, but the abuse of terminology is standard. If s can reach t, there must be a shortest simple path from s to t; it's just NP-hard to compute (when there are negative cycles), by an easy reduction from the Hamiltonian path problem. On the other hand, if there is a shortest *walk* from s to t, that walk must be a simple path, and therefore must be the shortest simple path from s to t. Blerg.

Figure 8.4. An undirected graph where shortest paths from s are unique but do not define a tree.

A complete treatment of undirected graphs with negative edges is beyond the scope of this book. I will only mention, for people who want to follow up via Google, that a *single* shortest path in an undirected graph with negative edges can be computed in $O(VE + V^2 \log V)$ time, by a reduction to maximum weighted matching.

8.3 The Only SSSP Algorithm

Just like graph traversal and minimum spanning trees, many different SSSP algorithms can be described as special cases of a single generic algorithm, first proposed by Lester Ford in 1956 and independently described by George Dantzig in 1957[3] and again by George Minty in 1958. Each vertex v in the graph stores two values, which (inductively) describe a *tentative* shortest path from s to v.

- $dist(v)$ is the length of the tentative shortest $s \rightsquigarrow v$ path, or ∞ if there is no such path.
- $pred(v)$ is the predecessor of v in the tentative shortest $s \rightsquigarrow v$ path, or Null if there is no such vertex.

The predecessor pointers automatically define a tentative shortest-path *tree* rooted at s; these pointers are exactly the same as the parent pointers in our generic graph traversal algorithm. At the beginning of the algorithm, we initialize the distances and predecessors as follows:

```
InitSSSP(s):
  dist(s) ← 0
  pred(s) ← Null
  for all vertices v ≠ s
      dist(v) ← ∞
      pred(v) ← Null
```

During the execution of the algorithm, an edge $u \to v$ is ***tense*** if $dist(u) + w(u \to v) < dist(v)$. If $u \to v$ is tense, the tentative shortest path $s \rightsquigarrow v$ is clearly incorrect, because the path $s \rightsquigarrow u \to v$ is shorter. We can correct (or at least improve) this obvious overestimate by ***relaxing*** the edge as follows:

[3]Specifically, Dantzig showed that the shortest path problem can be phrased as a linear programming problem, and then described an interpretation of his simplex method in terms of the original graph. His description is (morally) equivalent to Ford's relaxation strategy.

```
RELAX(u→v):
  dist(v) ← dist(u) + w(u→v)
  pred(v) ← u
```

Now that everything is set up, Ford's generic algorithm has a simple one-line description:

Repeatedly relax tense edges, until there are no more tense edges.

```
FORDSSSP(s):
  INITSSSP(s)
  while there is at least one tense edge
      RELAX any tense edge
```

If FORDSSSP eventually terminates (because there are no more tense edges), then the predecessor pointers correctly define a shortest-path tree, and each value $dist(v)$ is the actual shortest-path distance from s to v. In particular, if s cannot reach v, then $dist(v) = \infty$, and if any negative cycle is reachable from s, then the algorithm never terminates.

The correctness of Ford's generic algorithm follows from the following series of simpler claims:

1. At any moment during the execution of the algorithm, for every vertex v, the distance $dist(v)$ is either ∞ or the length of a walk from s to v. This claim can be proved by induction on the number of relaxations.
2. If the graph has no negative cycles, then $dist(v)$ is either ∞ or the length of some *simple path* from s to v. Specifically, if $dist(v)$ is the length of a walk from s to v that contains a directed cycle, that cycle must have negative length. This claim implies that if G has no negative cycles, the relaxation algorithm eventually halts, because there are only a finite number of simple paths in G.
3. If no edge in G is tense, then for every vertex v, the distance $dist(v)$ is the length of the predecessor path $s \to \cdots pred(pred(v)) \to pred(v) \to v$. Specifically, if v violates this condition but its predecessor $pred(v)$ does not, the edge $pred(v) \to v$ is tense.
4. If no edge in G is tense, then for every vertex v, the path of predecessor edges $s \to \cdots \to pred(pred(v)) \to pred(v) \to v$ is in fact a shortest path from s to v. Specifically, if v violates this condition but its predecessor u *in some shortest path* does not, the edge $u \to v$ is tense. This claim also implies that if G has a negative cycle, then some edge is *always* tense, so the generic algorithm never halts.

So far I haven't said anything about how to find tense edges, or which tense edge(s) to relax if there is more than one. Just like whatever-first search, there

are several different instantiations of Ford's generic relaxation algorithm. Unlike whatever-first search, however, the efficiency and correctness of each search strategy depends on the structure of the input graph.

The rest of this chapter considers the four most common instantiations of Ford's algorithm, each of which is the best choice for a different class of input graphs. I'll leave the remaining details of the generic correctness proof as exercises, and instead give (more informative, self-contained) correctness proofs for each of these four specific algorithms.

8.4 Unweighted Graphs: Breadth-First Search

In the simplest special case of the shortest path problem, all edges have weight 1, and the length of a path is just the number of edges. This special case can be solved by a species of our generic graph-traversal algorithm called ***breadth-first search***. Breadth-first search is often attributed to Edward Moore, who described it in 1957 (as "Algorithm A") as the first published method to find the shortest path through a maze.[4] Especially in the context of VLSI wiring and robot path planning, breadth-first search is sometimes attributed to Chin Yang Lee, who described several applications of Moore's "Algorithm A" (with proper credit to Moore) in 1961. However, in 1945, more than a decade before Moore considered mazes, Konrad Zuse described an implementation of breadth-first search, as a method to count and label the components of a disconnected graph.[6]

[4]Moore was motivated by a weakness in Claude Shannon's maze-solving robot "Theseus", which Shannon designed and constructed in 1950. (Theseus used a memoized version of depth-first search, implemented using electromechanical relays; this was almost certainly the first *implementation* of depth-first search in graphs.) According to Moore, "When this machine was used with a maze which had more than one solution, a visitor asked why it had not been built to always find the shortest path. Shannon and I each attempted to find economical methods of doing this by machine. He found several methods suitable for analog computation,[5] and I obtained these algorithms."

[5]Analog methods for computing shortest paths through mazes have been proposed using ball bearings, fluid/plasma flow, chemical reaction waves, chemotaxis, resistor networks, electric circuits with LEDs, memristor networks, glow discharge in microfluidic chips, growing plants, slime mold, amoebas, ants, bees, nematodes, and tourists.

[6]Konrad Zuse was one of the early pioneers of computing; he designed and built his first programmable computer (later dubbed the Z1) in the late 1930s from metal strips and rods in his parents' living room; the Z1 and its original blueprints were destroyed by a British air raid in 1944. Zuse's 1945 PhD thesis describes the very first high-level programming language, called *Plankalkül*. The first complete example of a Plankalkül program in Zuse's thesis is an implementation of breadth-first search to count components, along with a pseudocode explanation and an illustrated step-by-step trace of the algorithm's execution on a disconnected graph with eight vertices. Due to the collapse of the Nazi government, Zuse was unable to submit his PhD thesis, and Plankalkül remained unpublished until 1972. The first Plankalkül compiler was finally implemented in 1975 by Joachim Hohmann.

Breadth-first search maintains a first-in-first-out queue of vertices, which initially contains only the source vertex s. At each iteration, the algorithm PULLS a vertex u from the front of the queue and examines each of its outgoing edges $u \to v$. Whenever the algorithm discovers an outgoing tense edge $u \to v$, it relaxes that edge and PUSHes vertex v onto the queue. The algorithm ends when the queue becomes empty.

```
BFS(s):
  InitSSSP(s)
  Push(s)
  while the queue is not empty
      u ← Pull()
      for all edges u→v
          if dist(v) > dist(u) + 1          ⟨⟨if u→v is tense⟩⟩
              dist(v) ← dist(u) + 1
              pred(v) ← u                   ⟨⟨relax u→v⟩⟩
              Push(v)
```

Breadth-first search is somewhat easier to analyze if we break its execution into *phases*, by introducing an imaginary *token*. Before we PULL any vertices, we PUSH the token into the queue. The current phase ends when we PULL the token out of the queue; we begin the next phase when we PUSH the token into the queue again. Thus, the first phase consists entirely of scanning the source vertex s. The algorithm ends when the queue contains *only* the token. The modified algorithm is shown in Figure 8.5, and Figure 8.6 shows an example of this algorithm in action. Let me emphasize that these modifications are merely a convenience for analysis; with or without the token, the algorithm PUSHes and PULLS vertices in the same order, scans edges in the same order, and outputs exactly the same distances and predecessors.

```
BFSWithToken(s):
  InitSSSP(s)
  Push(s)
  Push(✠)                  ⟨⟨start the first phase⟩⟩
  while the queue contains at least one vertex
      u ← Pull()
      if u = ✠
          Push(✠)          ⟨⟨start the next phase⟩⟩
      else
          for all edges u→v
              if dist(v) > dist(u) + 1          ⟨⟨if u→v is tense⟩⟩
                  dist(v) ← dist(u) + 1
                  pred(v) ← u                   ⟨⟨relax u→v⟩⟩
                  Push(v)
```

Figure 8.5. Breadth-first search with an end-of-phase token (✠); bold red lines are only for analysis.

Figure 8.6. A complete run of breadth-first search in a directed graph. Vertices are pulled from the queue in the order s ✠ b d ✠ c a g ✠ f e ✠ h ✠ ✠, where ✠ is the end-of-phase token. Bold vertices are in the queue at the end of each phase. Bold edges describe the evolving shortest path tree.

Let me emphasize that in the following lemma, $dist(v)$ is just a variable maintained by the algorithm. While $dist(v)$ *intuitively* represents a tentative shortest-path distance, we cannot assume (yet) that $dist(v)$ is ever actually equal to the true shortest-path distance from s to v. Don't worry; we'll get there.

Lemma 8.1. *For every integer $i \geq 0$ and every vertex v, at the end of the ith phase, either $dist(v) = \infty$ or $dist(v) \leq i$, and v is in the queue if and only if $dist(v) = i$.*

Proof: The proof proceeds by induction on i. The base case $i = 0$ is straightforward: At the start of the first phase ("at the end of the zeroth phase"), the queue contains only the start vertex s and the token ✠, and InitSSSP just set $dist(s) \leftarrow 0$ and $dist(v) \leftarrow \infty$ for all $v \neq s$.

So fix an integer $i > 0$. The inductive hypothesis implies that at the *start* of the ith phase, the queue contains every vertex u with $dist(u) = i - 1$, followed by the token ✠. In other words, the queue looks like this:

$$\rightarrow \quad ✠ \quad i-1 \quad i-1 \quad \cdots \quad i-1 \quad \rightarrow$$

Thus, before we Pull the token ✠ from the queue, ending the ith phase, we Pull *every* vertex u with $dist(u) = i - 1$.

For each such vertex u, we consider every outgoing edge $u \to v$. If $u \to v$ is tense, we set $dist(v) \leftarrow dist(u) + 1$, so that $dist(v) = i$, and then immediately

PUSH v into the queue. These are the only assignments to distance labels during the ith phase. Thus, by induction, during the entire ith phase, the queue contains some vertices with distance label $i-1$, followed by the token, followed by some vertices with distance label i:

$$\rightarrow \quad i \quad \cdots \quad i \quad \maltese \quad i-1 \quad \cdots \quad i-1 \quad \rightarrow$$

In particular, just before the ith phase ends, the queue contains the token, followed by some vertices with distance label i.

$$\rightarrow \quad i \quad i \quad \cdots \quad i \quad \maltese \quad \rightarrow$$

Moreover, vertex v appears in this final queue if and only if $dist(v)$ was changed during the ith phase. Thus, at the end of the ith phase, the queue contains *every* vertex v with $dist(v) = i$. □

Lemma 8.1 implies that the main body of BFS assigns distance labels in non-decreasing order; on the other hand, the distance label $dist(v)$ of each vertex v never increases. It follows that for each vertex v, the line "$dist(v) \leftarrow dist(u) + 1$" is executed *at most once*, during phase $dist(v)$. Similarly:

- Each predecessor pointer $pred(v)$ is changed at most once, during phase $dist(v)$.
- Each vertex v is PUSHed into the queue at most once, during phase $dist(v)$.
- Each vertex u is PULLed from the queue at most once, during phase $dist(u)+1$.
- For each edge $u \rightarrow v$, the comparison "is $dist(v) > dist(u) + 1$" is performed at most once, during phase $dist(u) + 1$.

Altogether, these observations imply that breadth-first search runs in $\boldsymbol{O(V+E)}$ ***time***. Intuitively, we can think of the vertices in the queue as a "wavefront" expanding monotonically outward from the source vertex s, passing over each vertex and edge of the graph at most once. This expanding wavefront analogy was already proposed by Chin Yang Lee in 1961, inspired by visualizations produced by his implementation of Moore's Algorithm A.

These observations also imply that we can replace the condition "if $dist(v) > dist(u) + 1$" by the (arguably) simpler test "if $dist(v) = \infty$". Then distances play the same role as the marks maintained by other graph-traversal algorithms, which ensure that each vertex is visited only once. Specifically, a vertex is "marked" if and only if its distance label is finite.

But we still need to prove that the final distance labels are correct!

Theorem 8.2. *When BFS ends, $dist(v)$ is the length of the shortest path in G from s to v, for every vertex v.*

Proof: Fix an arbitrary vertex v, and consider an arbitrary path $v_0 \to v_1 \to \cdots \to v_\ell$ in G, where $v_0 = s$ and $v_\ell = v$. I claim that $dist(v_j) \le j$ for each index j; in particular $dist(v) \le \ell$. We can prove this claim by induction on j as follows.

- Trivially $dist(v_0) = dist(s) = 0$.
- For any index $j > 0$, the induction hypothesis implies $dist(v_{j-1}) \le j-1$. Immediately after we Pull vertex v_{j-1} from the queue, either $dist(v_j) \le dist(v_{j-1}) + 1$ already, or we set $dist(v_j) \leftarrow dist(v_{j-1}) + 1$. In either case, we have $dist(v_j) \le dist(v_{j-1}) + 1 \le j$.

We just proved that $dist(v)$ is at most the length of an *arbitrary* path from s to v; it follows that $dist(v)$ is at most the length of the *shortest* path from s to‘v.

A similar induction proof implies that $dist(v)$ is the length of the predecessor path $s \to \cdots \to pred(pred(v)) \to pred(v) \to v$, so this must be the shortest path. □

8.5 Directed Acyclic Graphs: Depth-First Search

Shortest paths are also easy to compute in directed acyclic graphs, even when the edges are weighted, and in particular, even when some edges have negative weight. (We don't have to worry about negative cycles, because by definition, dags don't have *any* cycles!) Indeed, this is a completely standard dynamic programming algorithm.

Let G be a directed graph with weighted edges, and let s be the fixed start vertex. For any vertex v, let $dist(v)$ denote the length of the shortest path in G from s to v. This function satisfies the following simple recurrence:

$$dist(v) = \begin{cases} 0 & \text{if } v = s \\ \min\limits_{u \to v} \left(dist(u) + w(u \to v) \right) & \text{otherwise} \end{cases}$$

In fact, this identity holds for *all* directed graphs, but it is only a *recurrence* for directed acyclic graphs. If the input graph G contained a cycle, a recursive evaluation of this function would fall into an infinite loop; however, because G is a dag, each recursive call visits an earlier vertex in topological order.

The dependency graph for this recurrence is the reversal of the input graph G: subproblem $dist(v)$ depends on $dist(u)$ if and only if $u \to v$ is an edge in G. Thus, we compute the distance of every in $\boldsymbol{O(V+E)}$ ***time*** by performing a depth-first search in the reversal of G and considering vertices in postorder. Equivalently, we can consider the vertices in the original graph G in topological order, as shown in Figure 8.7.

The resulting dynamic-programming algorithm is another example of Ford's generic relaxation algorithm! To make this connection clearer, we can move the initialization $dist(v)$ outside the main loop and add computation of predecessor pointers, as shown in Figure 8.8. Figure 8.9 shows this algorithm in action.

```
DagSSSP(s):
  for all vertices v in topological order
      if v = s
          dist(v) ← 0
      else
          dist(v) ← ∞
          for all edges u→v
              if dist(v) > dist(u) + w(u→v)          ⟨⟨if u→v is tense⟩⟩
                  dist(v) ← dist(u) + w(u→v)         ⟨⟨relax u→v⟩⟩
```

Figure 8.7. Computing shortest paths in a dag using dynamic programming

```
DagSSSP(s):
  InitSSSP(s)
  for all vertices v in topological order
      for all edges u→v
          if u→v is tense
              Relax(u→v)
```

Figure 8.8. Computing shortest paths in a dag using Ford's algorithm. (These are the same algorithm.)

Figure 8.9. Computing shortest paths in a dag, by relaxing **incoming** edges in topological order. In each iteration, bold edges indicate predecessors, and the bold vertex is about to be scanned. Compare with Figure 8.10.

DagSSSP differs from breadth-first search and other instances of Ford's relaxation strategy in one minor respect. Whenever these other shortest-path algorithms consider a vertex, they attempt to relax each of its *outgoing* edges, intuitively *pushing* the wavefront forward from the source; whereas, DagSSSP attempts to relax each of the *incoming* edges of each vertex, intuitively *pulling* the wavefront forward.

However, if we modify DagSSSP to relax outgoing edges instead of incoming edges, we obtain another algorithm that computes shortest paths in dags in $O(V + E)$ ***time*** and that more closely resembles our other shortest-path algorithms.

```
PushDagSSSP(s):
  InitSSSP(s)
  for all vertices u in topological order
      for all outgoing edges u→v
          if u→v is tense
              Relax(u→v)
```

Figure 8.10 shows an execution of this modified algorithm on the same graph as Figure 8.9. The correctness of PushDagSSSP follows immediately from the correctness of Ford's general relaxation strategy, but it's not hard to prove correctness directly, by induction over the vertices in topological order.

8.6 Best-First: Dijkstra's Algorithm

If we replace the FIFO queue in breadth-first search with a priority queue, where the key of a vertex v is its tentative distance $dist(v)$, we obtain an algorithm first "published" in 1957 by a team of researchers at the Case Institute of Technology led by Michael Leyzorek, in an annual project report for the Combat Development Department of the US Army Electronic Proving Ground. The same algorithm was independently discovered by Edsger Dijkstra in 1956 (but not published until 1959), again by George Minty some time before 1960, and again by Peter Whiting and John Hillier in 1960. A nearly identical algorithm was also described by George Dantzig in 1958. Although several early sources called it "Minty's algorithm", this approach is now universally known as "Dijkstra's algorithm", in full accordance with Stigler's Law.[7] Pseudocode for this algorithm is shown in Figure 8.11.

An easy induction proof implies that, at all times during the execution of this algorithm, an edge $u \to v$ is tense if and only if vertex u is either in the priority

[7] I will follow this common convention, despite the historical inaccuracy, partly because I don't think anybody wants to read about the "Leyzorek-Gray-Johnson-Ladew-Meaker-Petry-Seitz-Dantzig-Dijkstra-Minty-Whiting-Hillier algorithm", and partly because papers that aren't *actually published* don't count.

Figure 8.10. Computing shortest paths in a dag, by relaxing **outgoing** edges in topological order. In each iteration, bold edges indicate predecessors, and the bold vertex is about to be scanned. Compare with Figure 8.9.

```
DIJKSTRA(s):
  INITSSSP(s)
  INSERT(s, 0)
  while the priority queue is not empty
      u ← EXTRACTMIN( )
      for all edges u→v
          if u→v is tense
              RELAX(u→v)
              if v is in the priority queue
                  DECREASEKEY(v, dist(v))
              else
                  INSERT(v, dist(v))
```

Figure 8.11. Dijkstra's algorithm.

queue or is the vertex most recently Extracted from the priority queue. Thus, Dijkstra's algorithm is an instance of Ford's general strategy, which implies that it correctly computes shortest paths, provided there are no negative cycles in G.

No Negative Edges

Dijkstra's algorithm is particularly well-behaved when the input graph has no negative-weight edges. In this setting, the algorithm intuitively expands a wavefront outward from the source vertex s, passing over vertices in increasing order of their distance from s, similarly to breadth-first search. Figure 8.12 shows the algorithm in action.

Figure 8.12. The first four iterations of Dijkstra's algorithm on a graph with no negative edges. In each iteration, bold edges indicate predecessors; shaded vertices are in the priority queue; and the bold vertex is about to be scanned. The remaining iterations do not change the distances or the shortest-path tree.

We can derive a self-contained proof of correctness for Dijkstra's algorithm in this setting by formalizing this wavefront intuition. For each integer i, let u_i denote the vertex returned by the ith call to ExtractMin, and let d_i be the value of $dist(u_i)$ just after this Extraction. In particular, we have $u_1 = s$ and $d_1 = 0$. We cannot assume at this point that the vertices u_i are distinct; in principle, the same vertex might be Extracted more than once.

Lemma 8.3. *If G has no negative-weight edges, then for all $i < j$, we have $d_i \leq d_j$.*

Proof: Assume G has no negative weight edges. Fix an arbitrary index i; to prove the lemma, it suffices to prove that $d_{i+1} \geq d_i$. There are two cases to consider.

- If G contains the edge $u_i \to u_{i+1}$, and this edge is relaxed during the ith iteration of the main loop, then at the end of the ith iteration, we have $dist(u_{i+1}) = dist(u_i) + w(u_i \to u_{i+1}) \geq dist(u_i)$, because all edge weights are non-negative.
- Otherwise, at the start of the ith iteration, u_{i+1} must already be in the priority queue, and it must have priority $dist(u_{i+1}) \geq dist(u_i)$, because u_i is the vertex returned by ExtractMin. Moreover, $dist(u_{i+1})$ does not change during the ith iteration.

In both cases, we conclude that $d_{i+1} \geq d_i$. The lemma now follows immediately by induction on i. □

Lemma 8.4. *If G has no negative-weight edges, each vertex of G is Extracted from the priority queue at most once.*

Proof: Suppose v is Extracted more than once. Specifically, suppose v is Extracted in the ith iteration of the main loop, reInserted during the jth iteration, and reExtracted during the kth iteration, for some indices $i < j < k$. Then in the notation of the previous proof, we have $v = u_i = u_k$.

The distance label $dist(v)$ never increases. Moreover, $dist(v)$ strictly decreases during the jth iteration, just before v is reInserted. It follows that $d_i > d_k$. Therefore, by the previous lemma, G has at least one negative-weight edge. □

Lemma 8.4 immediately implies that each vertex is scanned at most once, and thus that each edge is relaxed at most once. However, unlike in breadth-first search, each distance label $dist(v)$ can change multiple times. The first time $dist(v)$ changes from ∞, we Insert v into the priority queue; after that, each change to $dist(v)$ is followed by a call to DecreaseKey. After v is Extracted from the priority queue, its distance label never changes.

The rest of the correctness proof is almost identical to breadth-first search.

Theorem 8.5. *If G has no negative-weight edges, then when Dijkstra ends, $dist(v)$ is the length of the shortest path in G from s to v, for every vertex v.*

Proof: Fix an arbitrary vertex v, and consider an arbitrary path $v_0 \to v_1 \to \cdots \to v_\ell$ in G, where $v_0 = s$ and $v_\ell = v$. For any index j, let L_j denote the length of the subpath $v_0 \to v_1 \to \cdots \to v_j$. We prove by induction that $dist(v_j) \leq L_j$ for all j.

- Trivially $dist(v_0) = dist(s) = 0 = L_0$.
- For any index $j > 0$, the induction hypothesis implies $dist(v_{j-1}) \leq L_{j-1}$. Immediately after we Pull vertex v_{j-1} from the queue, either $dist(v_i) \leq dist(v_{j-1}) + w(v_{j-1} \to v_j)$ already, or we set $dist(v_i) \leftarrow dist(v_{j-1}) + w(v_{j-1} \to v_j)$. In either case, we have

$$dist(v_j) \;\leq\; dist(v_{j-1}) + w(v_{j-1} \to v_j) \;\leq\; L_{j-1} + w(v_{j-1} \to v_j) \;=\; L_j.$$

We just proved that $dist(v)$ is at most the length of *every* path from s to v; it follows that $dist(v)$ is at most the length of the *shortest* path from s to v.

On the other hand, a similar induction proof implies that $dist(v)$ is the length of the predecessor path $s \to \cdots \to pred(pred(v)) \to pred(v) \to v$. □

It remains only to bound the algorithm's running time. Altogether Dijkstra performs at most E DecreaseKey operations, and at most V Insert and ExtractMin operations. Thus, if we implement the underlying priority queue using a standard binary heap, which supports each operation in $O(\log V)$ time, Dijkstra runs in $\boldsymbol{O(E \log V)}$ ***time***.[8]

If we know in advance that our input graphs will *never* have negative edges, we can simplify Dijkstra's algorithm slightly, by Inserting every vertex into the priority queue in the initialization phase, and then only calling DecreaseKey in the main loop, as shown in Figure 8.13. This is the version of Dijkstra's algorithm presented by most algorithms textbooks, Wikipedia, and even Dijkstra's original paper; it's also the version of Dijkstra's algorithm that I described as "best-first search" in Chapter 5.

```
NonnegativeDijkstra(s):
  InitSSSP(s)
  for all vertices v
      Insert(v, dist(v))
  while the priority queue is not empty
      u ← ExtractMin( )
      for all edges u→v
          if u→v is tense
              Relax(u→v)
              DecreaseKey(v, dist(v))
```

Figure 8.13. Dijkstra's algorithm very slightly simplified for graphs without negative edges. Differences from Dijkstra are bold red.

♥Negative Edges

However, NonnegativeDijkstra does *not* correctly compute shortest paths in graphs with negative edges. Moreover, even when all edge weights are

[8]Shortest-path papers from the 1950s never mentioned priority queues. Dijkstra proposed a brute-force scan of all vertices on the wavefront at every iteration; his original algorithm runs in $\boldsymbol{O(V^2)}$ ***time***, which is actually faster than the binary-heap implementation when $E = \Omega(V^2)$! Minty proposed a brute-force scan of all *edges* $u \to v$ such that $dist(u)$ is finite but $dist(v)$ is not; thus, his original algorithm runs in $O(VE)$ time. The use of a priority queue, implemented as a binary heap, to obtain near-linear running time was proposed by Donald Johnson in 1977. The running time can be improved to $\boldsymbol{O(E + V \log V)}$ using a more complex priority queue data structure called a *Fibonacci heaps*. There are even faster algorithms, using even more sophisticated priority queues, for the special case of integer edge weights.

positive, NonnegativeDijkstra is no faster than Dijkstra (either in theory or in practice). For both of these reasons, I think Dijkstra is more deserving of the name "Dijkstra's algorithm" than NonnegativeDijkstra. Even Edsger Dijkstra would have agreed that a correct algorithm that is sometimes (and in practice, rarely) slow is better than a fast algorithm that doesn't always work!

Unfortunately, when the input graph has negative edges, the familiar "expanding wavefront" intuition is no longer accurate. The same vertex can be Extracted multiple times; the same edge can be relaxed multiple times; and distances might not be discovered in increasing order. Figure 8.15 shows an example execution where the top left vertex is Extracted six times, and the top three edges are each relaxed twice.

For graphs without negative cycles, but no other restrictions on edge weights, the worst-case running time of Dijkstra is actually exponential. Figure 8.14 shows particularly simple family of graphs (due to Douglas Shier and Christoph Witzgall) that forces Dijkstra to perform $\Theta(2^{V/2})$ relaxations.[9] A more complex family of graphs (which I'll leave as an exercise) forces $\Theta(2^V)$ relaxations, which is the worst possible. *In practice*, however, Dijkstra's algorithm is usually fast even for graphs with negative edges.

Figure 8.14. A directed graph with negative edges that forces Dijkstra to run in exponential time.

8.7 Relax ALL the Edges: Bellman-Ford

The simplest implementation of Ford's generic shortest-path algorithm was first sketched by Alfonso Shimbel in 1954, described in more detail by Edward Moore in 1957, and independently rediscovered by Max Woodbury and George Dantzig in 1957, by Richard Bellman in 1958, and by George Minty in 1958. (Neither Woodbury and Dantzig nor Minty published their algorithms.) In full compliance with Stigler's Law, the algorithm is almost universally known as ***Bellman-Ford***,[10] because Bellman explicitly used Ford's 1956 formulation of

[9] Amusingly, Shier and Witzgall's example is a dag with only $O(V)$ edges, which implies that shortest paths can be computed in only $O(V)$ time, even if we *didn't* already notice that the zig-zag path along the top *is* the shortest path tree.

[10] I will follow this common convention, despite the historical inaccuracy, partly because I don't think anyone really wants read about the "Shimbel / Moore / Woodbury-Dantzig / Bellman-Ford / Kalaba / Minty algorithm", and partly because I'm tired of people looking at me funny when I talk about "Shimbel's algorithm".

Figure 8.15. A complete run of Dijkstra's algorithm on a graph with negative edges. At each iteration, bold edges indicate predecessors; shaded vertices are in the priority queue; and the bold vertex is the next to be scanned. Compare with Figure 8.17.

relaxing edges, although some authors refer to "Bellman-Kalaba"[11] and a few early sources refer to "Bellman-Shimbel".

The Shimbel / Moore / Woodbury-Dantzig / Bellman-Ford / Kalaba / Minty / Brosh[12] algorithm can be summarized in one line:

BELLMAN-FORD: Relax ***ALL*** the tense edges, then recurse.

```
BellmanFord(s)
  InitSSSP(s)
  while there is at least one tense edge
      for every edge u→v
          if u→v is tense
              Relax(u→v)
```

The following lemma is the key to proving both correctness and efficiency of Bellman-Ford. For every vertex v and non-negative integer i, let $\boldsymbol{dist_{\leq i}(v)}$ denote the length of the shortest *walk* in G from s to v consisting of *at most* i edges. In particular, $dist_{\leq 0}(s) = 0$ and $dist_{\leq 0}(v) = \infty$ for all $v \neq s$.

Lemma 8.6. *For every vertex v and non-negative integer i, after i iterations of the main loop of* BELLMANFORD*, we have $dist(v) \leq dist_{\leq i}(v)$.*

Proof: The proof proceeds by induction on i. The base case $i = 0$ is trivial, so assume $i > 0$. Fix a vertex v, and let W be the shortest walk from s to v consisting of at most i edges (breaking ties arbitrarily). By definition, W has length $dist_{\leq i}(v)$. There are two cases to consider.

- Suppose W has no edges. Then W must be the trivial walk from s to s, so $v = s$ and $dist_{\leq i}(s) = 0$. We set $dist(s) \leftarrow 0$ in INITSSSP, and $dist(s)$ can never increase, so we always have $dist(s) \leq 0$.
- Otherwise, let $u \to v$ be the last edge of W. The induction hypothesis implies that after $i-1$ iterations, $dist(u) \leq dist_{\leq i-1}(u)$. During the ith iteration of the outer loop, when we consider the edge $u \to v$ in the inner loop, either $dist(v) < dist(u) + w(u \to v)$ already, or we set $dist(v) \leftarrow dist(u) + w(u \to v)$. In both cases, we have $dist(v) \leq dist_{\leq i-1}(u) + w(u \to v) = dist_{\leq i}(v)$. As usual, $dist(v)$ cannot increase (although $dist(v)$ might decrease further before the ith iteration of the outer loop ends).

[11] This name is most likely a reference to Richard Bellman and Robert Kalaba's 1965 monograph on dynamic programming and control theory, which describes Bellman's algorithm. Bellman and Kalaba also published an extension of Bellman's algorithm in 1960 that computes kth shortest paths, for any constant k.

[12] Go read everything in *Hyperbole and a Half* again. And then adopt another cat, so you can buy it another copy of the book.

In both cases, we conclude that $dist(v) \leq dist_{\leq i}(v)$ at the end of the ith iteration. □

If the input graph has no negative cycles, the shortest walk from s to any other vertex is a simple path with at most $V-1$ edges; it follows that BellmanFord halts with the correct shortest-path distances after at most $V-1$ iterations. Said differently, if any edge is still tense after $V-1$ iterations, then the input graph must contain a negative cycle! Thus, we can rewrite the algorithm more concretely as follows:

```
BellmanFord(s)
  InitSSSP(s)
  repeat V − 1 times
      for every edge u→v
          if u→v is tense
              Relax(u→v)
  for every edge u→v
      if u→v is tense
          return "Negative cycle!"
```

Each iteration of the inner loop trivially requires $O(E)$ time, so the overall algorithm runs in $\boldsymbol{O(VE)}$ ***time***. Thus, Bellman-Ford is *always* efficient, even if the graph has negative edges, and in fact even if the graph has negative *cycles*.

If all edge weights are non-negative, however, Dijkstra's algorithm is faster, at least in the worst case. (In practice, Dijkstra's algorithm is often faster than Bellman-Ford even for graphs with negative edges.)

Moore's Improvement

Neither Moore nor Bellman described the Bellman-Ford algorithm in the form I've presented here. Moore presented his version of the algorithm ("Algorithm D") in the same paper that proposed breadth-first search ("Algorithm A") for unweighted graphs; indeed, the two algorithms are nearly identical. Although Moore's algorithm has the same $O(VE)$ worst-case running time as BellmanFord, it is often significantly faster in practice, intuitively because it avoids checking edges that are "obviously" not tense.

Moore derived his weighted shortest-path algorithm by making two modifications to breadth-first search. First, replace each "+1" with "$+w(u\to v)$" in the innermost loop, to take the edge weights into account. Second, check whether a vertex is already in the FIFO queue before Insertring it, so that the queue always contains at most one copy of each vertex.[13]

[13]Moore's algorithm is still *correct* without this check, but the $O(VE)$ time bound is not.

Following our earlier analysis of breadth-first search, I'll introduce a "token" ✠ to break the execution of the algorithm into phases. Just like breadth-first search, each phase begins when the token is PUSHed into the queue, and ends when the token is PULLed out of the queue again. Just like BFS, the algorithm ends when the queue contains *only* the token. The resulting algorithm is shown in Figure 8.16.

```
MOORE(s):
  INITSSSP(s)
  PUSH(s)
  PUSH(✠)                    ⟨⟨start the first phase⟩⟩
  while the queue contains at least one vertex
      u ← PULL()
      if u = ✠
          PUSH(✠)            ⟨⟨start the next phase⟩⟩
      else
          for all edges u→v
              if u→v is tense
                  RELAX(u→v)
                  if v is not already in the queue
                      PUSH(v)
```

Figure 8.16. Moore's shortest-path algorithm. Bold red lines involving the token ✠ are only for analysis.

Because the queue contains at most one copy of each vertex at any time, each vertex is PULLed from the queue at most once in each phase, and therefore each edge $u \to v$ is checked for tenseness at most once in each phase. Moreover, every edge that is tense when a phase begins is relaxed during that phase. (Some edges that become tense during the phase might also be relaxed during that phase, and some relaxed edges might become tense again in the same phase.) Thus, MOORE can be viewed as a refinement of BELLMANFORD that uses a queue to maintain tense edges, rather than testing every edge by brute force. In particular, a similar inductive proof establishes the following analogue of Lemma 8.6:

Lemma 8.7. *For every vertex v and non-negative integer i, after i phases of* MOORE*, we have $dist(v) \leq dist_{\leq i}(v)$.*

Thus, if the input graph has no negative cycles, MOORE halts after at most $V - 1$ phases. In each phase, we scan each vertex at most once, so we relax each edge at most once, so the worst-case running time of a single phase is $O(E)$. Thus, the overall running time of MOORE is $\boldsymbol{O(VE)}$. In practice, however, MOORE often computes shortest paths considerably faster than BELLMANFORD, because it only scans an edge $u \to v$ if $dist(u)$ was changed in the previous phase.

If the input graph contains a negative cycle, MOORE never halts. Fortunately, like BELLMANFORD, it is easy to modify Moore's algorithm to report negative

Figure 8.17. A complete run of Moore's algorithm on a directed graph with negative edges. Nodes are pulled from the queue in the order s ✠ a b c ✠ d f ✠ d e ✠ d ✠ ✠, where ✠ is the end-of-phase token. At the start of each phase, bold edges indicate predecessors, and shaded vertices are in the vertex queue. Compare with Figures 8.6 and 8.15.

cycles if they exist. Perhaps the easiest modification is to *actually* maintain a token, and count the number of times the token is Pulled from the queue. Then the input graph contains a negative cycle if and only if the queue is non-empty immediately after the token is Pulled for the $(V-1)$th time.

Dynamic Programming Formulation

Like almost everything else with his name on it, Richard Bellman derived the "Bellman-Ford" shortest-path algorithm via dynamic programming. As usual, we need to start with a recursive definition of shortest path distances. It's tempting to use the same identity that we exploited for directed acyclic graphs:

$$dist(v) = \begin{cases} 0 & \text{if } v = s \\ \min\limits_{u \to v} \left(dist(u) + w(u \to v) \right) & \text{otherwise} \end{cases}$$

Unfortunately, if the input graph is not a dag, this recurrence doesn't work! Suppose the input graph contains the directed cycle $u \to v \to w \to u$. To compute $dist(w)$ we first need $dist(v)$, and to compute $dist(v)$ we first need $dist(u)$, but to compute $dist(u)$ we first need $dist(w)$. If the input graph has any directed cycles, we get stuck in an infinite loop!

To support a proper recurrence, we need to add an additional structural parameter to the distance function, which decreases monotonically at each recursive call, defined so that the function is trivial to evaluate when the parameter reaches 0. Bellman chose ***the maximum number of edges*** as this additional parameter.[14]

As in our earlier analysis, let $dist_{\leq i}(v)$ denote the length of the shortest walk from s to v consisting of at most i edges. Bellman observed that this function obeys the following ~~Bellman's equation~~ recurrence:

$$dist_{\leq i}(v) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = s \\ \infty & \text{if } i = 0 \text{ and } v \neq s \\ \min \left\{ \begin{array}{c} dist_{\leq i-1}(v) \\ \min\limits_{u \to v} \left(dist_{\leq i-1}(u) + w(u \to v) \right) \end{array} \right\} & \text{otherwise} \end{cases}$$

Let's assume that the graph has no negative cycles, so our goal is to compute $dist_{\leq V-1}(v)$ for every vertex v. Here is a straightforward dynamic-programming evaluation of this recurrence, where $dist[i, v]$ stores the value of $dist_{\leq i}(v)$. Correctness of the final shortest-path distances follows from the correctness of the recurrence, and the $O(VE)$ running time is obvious. This is essentially how Bellman presented his shortest-path algorithm.

```
BellmanFordDP(s)
  dist[0, s] ← 0
  for every vertex v ≠ s
      dist[0, v] ← ∞
  for i ← 1 to V − 1
      for every vertex v
          dist[i, v] ← dist[i − 1, v]
          for every edge u→v
              if dist[i, v] > dist[i − 1, u] + w(u→v)
                  dist[i, v] ← dist[i − 1, u] + w(u→v)
```

We can transform this dynamic programming algorithm into our original formulation of BellmanFord through a short series of minor optimizations. First, each iteration of the outermost loop considers each edge $u \to v$ exactly once, but the order in which we consider those edges doesn't actually matter. Thus, we can safely remove one level of indentation from the last three lines! The modified algorithm may consider edges in a different *order*, but it still correctly computes $dist_{\leq i}(v)$ for all i and v.

[14] As we'll see in the next chapter, this is not the only reasonable choice.

```
BellmanFordDP2(s)
  dist[0, s] ← 0
  for every vertex v ≠ s
      dist[0, v] ← ∞
  for i ← 1 to V − 1
      for every vertex v
          dist[i, v] ← dist[i − 1, v]
      for every edge u→v
          if dist[i, v] > dist[i − 1, u] + w(u→v)
              dist[i, v] ← dist[i − 1, u] + w(u→v)
```

Next we change the indices in the last two lines from $i-1$ to i. This change may cause the distances $dist[i,v]$ to approach the true shortest-path distances more quickly than before, but the algorithm correctly computes the true shortest path distances. Instead of $dist[i,v] = dist_{\leq i}(v)$, we now have $dist[i,v] \leq dist_{\leq i}(v)$ for all i and v, mirroring Lemmas 8.6 and 8.7.

```
BellmanFordDP3(s)
  dist[0, s] ← 0
  for every vertex v ≠ s
      dist[0, v] ← ∞
  for i ← 1 to V − 1
      for every vertex v
          dist[i, v] ← dist[i − 1, v]
      for every edge u→v
          if dist[i, v] > dist[i, u] + w(u→v)        ⟨⟨not i − 1!⟩⟩
              dist[i, v] ← dist[i, u] + w(u→v)       ⟨⟨not i − 1!⟩⟩
```

But this algorithm is a little silly. In the ith iteration of the outermost loop, we first copy the $(i-1)$th row of the array $dist[\cdot,\cdot]$ to the ith row, and then modify the elements of the ith row. So we really don't need a two-dimensional array at all; the iteration index i is completely redundant! In our final modification, we maintain only a one-dimensional array of tentative distances.

```
BellmanFordFinal(s)
  dist[s] ← 0
  for every vertex v ≠ s
      dist[v] ← ∞
  for i ← 1 to V − 1
      for every edge u→v
          if dist[v] > dist[u] + w(u→v)
              dist[v] ← dist[u] + w(u→v)
```

This final dynamic programming algorithm is almost identical to our original formulation of BellmanFord! The first three lines initialize the shortest path distances, and the last two lines relax the edge $u \to v$ if that edge is tense.

BellmanFordFinal is missing only two features of our earlier formulation: It does not maintain predecessor pointers or detect negative cycles. Fortunately, adding those features is straightforward.

Exercises

0. Let G be a directed graph with arbitrary edge weights (which may be positive, negative, or zero), possibly with negative cycles, and let s be an arbitrary vertex of G.

   (a) Suppose every vertex v stores a number $dist(v)$ (but no predecessor pointers). Describe and analyze an algorithm to determine whether $dist(v)$ is the shortest-path distance from s to v, for every vertex v.

   (b) Suppose instead that every vertex $v \neq s$ stores a pointer $pred(v)$ to another vertex in G (but no distances). Describe and analyze an algorithm to determine whether these predecessor pointers define a single-source shortest path tree rooted at s.

1. A *looped tree* is a weighted, directed graph built from a binary tree by adding an edge from every leaf back to the root. Every edge has non-negative weight.

A looped tree.

   (a) How much time would Dijkstra's algorithm require to compute the shortest path between two vertices u and v in a looped tree with n nodes?

   (b) Describe and analyze a faster algorithm.

2. Suppose we are given a directed graph G with weighted edges and two vertices s and t.

   (a) Describe and analyze an algorithm to find the shortest path from s to t when exactly one edge in G has negative weight. *[Hint: Modify Dijkstra's algorithm. Or don't.]*

(b) Describe and analyze an algorithm to find the shortest path from s to t when exactly k edges in G have negative weight. How does the running time of your algorithm depend on k?

3. Suppose we are given an undirected graph G in which every *vertex* has a positive weight.

   (a) Describe and analyze an algorithm to find a *spanning tree* of G with minimum total weight. (The total weight of a spanning tree is the sum of the weights of its vertices.)

   (b) Describe and analyze an algorithm to find a *path* in G from one given vertex s to another given vertex t with minimum total weight. (The total weight of a path is the sum of the weights of its vertices.)

   [Hint: One of these problems is trivial.]

4. For any edge e in any graph G, let $G \setminus e$ denote the graph obtained by deleting e from G. Suppose we are given a graph G and two vertices s and t. The *replacement paths* problem asks us to compute the shortest-path distance from s to t in $G \setminus e$, for *every* edge e of G. The output is an array of E distances, one for each edge of G.

   (a) Suppose G is a *directed* graph, and the shortest path from vertex s to vertex t passes through *every* vertex of G. Describe an algorithm to solve this special case of the replacement paths problem in $O(E \log V)$ time.

   ♥(b) Describe an algorithm to solve the replacement paths problem for arbitrary *undirected* graphs in $O(E \log V)$ time.

   In both subproblems, you may assume that all edge weights are non-negative. *[Hint: If we delete an edge of the original shortest path, how do the old and new shortest paths overlap?]*

5. Let $G = (V, E)$ be a connected directed graph with non-negative edge weights, let s and t be vertices of G, and let H be a subgraph of G obtained by deleting some edges. Suppose we want to reinsert exactly one edge from G back into H, so that the shortest path from s to t in the resulting graph is as short as possible. Describe and analyze an algorithm that chooses the best edge to reinsert, in $O(E \log V)$ time.

6. (a) Describe and analyze a modification of Bellman-Ford that actually returns a negative cycle if any such cycle is reachable from s, or a shortest-path tree if there is no such cycle. The modified algorithm should still run in $O(VE)$ time.

(b) Describe and analyze a modification of Bellman-Ford that computes the correct shortest path distances from s to every other vertex of the input graph, even if the graph contains negative cycles. Specifically, if any walk from s to v contains a negative cycle, your algorithm should end with $dist(v) = -\infty$; otherwise, $dist(v)$ should contain the length of the shortest path from s to v. The modified algorithm should still run in $O(VE)$ time.

♥(c) Repeat parts (a) and (b), but for Ford's generic relaxation algorithm. You may assume that the unmodified algorithm halts in $O(2^V)$ steps if there is no negative cycle; your modified algorithms should also run in $O(2^V)$ time.

7. Consider the following even looser variant of Ford's generic relaxation algorithm:

```
FellmanBored(s):
  InitSSSP(s)
  for i ← 1 to whatever, man, I don't care
      e_i ← any edge in G
      if e_i is tense
          Relax(e_i)
```

Prove that if FellmanBored examines the edges of any walk W starting from s, in order along W, then the last distance label in W is at most the length of W. More formally: If the edges of any walk $v_0 \to v_1 \to \cdots \to v_\ell$, where $v_0 = s$, define a *subsequence* of the edges $e_1, e_2, e_3, \ldots$ examined by FellmanBored, then we have $dist(v_\ell) \leq \sum_{i=1}^{\ell} w(v_{i-1} \to v_i)$. *[Hint: This property is almost easier to prove than it is to state correctly.]*

8. This problem considers several ways to detect negative cycles using Ford's generic relaxation algorithm.

(a) Prove that if $pred(s)$ ever changes after InitSSSP, then the input graph contains a negative cycle through s.

(b) Show that $pred(s)$ might never change after InitSSSP, even when the input graph contains a negative cycle through s.

(c) Let P denote the current graph of predecessor edges $pred(v) \to v$, and let X denote the set of all currently *tense* edges; both of these sets evolve as the algorithm executes. Prove that the input graph has no negative cycles if and only if $P \cup X$ is always a dag.

(d) Let R denote the set of all edges that have been relaxed so far; this set grows as the algorithm executes. Prove that the input graph has no negative cycles if and only if R is always a dag.

♥9. Prove that Dijkstra's algorithm performs $\Omega(2^V)$ relaxations in the worst case when edges are allowed to have negative weight, even if the underlying graph is acyclic. Specifically, for every positive integer n, construct a n-vertex dag G_n with weighted edges, such that Dijkstra's algorithm calls Relax $\Omega(2^n)$ times when G_n is the input graph. *[Hint: Binary counter.]*

♥10. Prove that Ford's generic relaxation algorithm (and therefore Dijkstra's algorithm) halts after at most $O(2^V)$ relaxations, unless the input graph contains a negative cycle. *[Hint: See Problem 8(d).]*

11. Suppose you are given a directed graph G in which ***every edge has negative weight***, and a source vertex s. Describe and analyze an efficient algorithm that computes the shortest-path distances from s to every other vertex in G. Specifically, for every vertex t:
    - If t is not reachable from s, your algorithm should report $dist(t) = \infty$.
    - If G has a cycle that is reachable from s, and t is reachable from that cycle, then the shortest-path distance from s to t is not well-defined, because there are paths (formally, walks) from s to t of arbitrarily large negative length. In this case, your algorithm should report $dist(t) = -\infty$.
    - If neither of the two previous conditions applies, your algorithm should report the correct shortest-path distance from s to t.

12. Although we typically speak of "the" shortest path between two nodes, single graph could contain several minimum-length paths with the same endpoints. Even for weighted graphs, it is often desirable to choose a minimum-weight path with the fewest edges; call this a ***best path*** from s to t. Suppose we are given a directed graph G with positive edge weights and a source vertex s in G. Describe and analyze an algorithm to compute *best* paths in G from s to every other vertex.

Figure 8.18. Four (of many) equal-length shortest paths. The first path is the "best" shortest path.

13. Describe and analyze an algorithm to determine the *number* of shortest paths from a source vertex s to a target vertex t in an arbitrary directed graph G with weighted edges. You may assume that all edge weights are positive and that all necessary arithmetic operations can be performed in

$O(1)$ time. *[Hint: Compute shortest path distances from s to every other vertex. Throw away all edges that cannot be part of a shortest path from s to another vertex. What's left?]*

14. You just discovered your best friend from elementary school on Twitbook. You both want to meet as soon as possible, but you live in two different cites that are far apart. To minimize travel time, you agree to meet at an intermediate city, and then you simultaneously hop in your cars and start driving toward each other. But where *exactly* should you meet?

    You are given a weighted graph $G = (V, E)$, where the vertices V represent cities and the edges E represent roads that directly connect cities. Each edge e has a weight $w(e)$ equal to the time required to travel between the two cities. You are also given a vertex p, representing your starting location, and a vertex q, representing your friend's starting location.

    Describe and analyze an algorithm to find the target vertex t that allows you and your friend to meet as quickly as possible.

15. You are hired as a cyclist for the Giggle Highway View project, which will provide street-level images along the entire US national highway system. As a pilot project, you are asked to ride the Giggle Highway-View Fixed-Gear Carbon-Fiber Bicycle from "the Giggleplex" in Portland, Oregon to "Gigglesburg" in Williamsburg, Brooklyn, New York.

    You are a hopeless caffeine addict, but like most Giggle employees you are also a coffee snob; you only drink independently roasted, hand-pulled, direct-trade, organic, shade-grown, single-origin espresso, unadulterated by milk or sugar, thank you *very* much. After each espresso shot, you can bike up to L miles before suffering a caffeine-withdrawal migraine.

    Giggle helpfully provides you with a map of the United States, in the form of an undirected graph G, whose vertices represent coffee shops that sell independently roasted hand-pulled direct-trade organic shade-grown single-origin espresso, and whose edges represent highway connections between them. Each edge e is labeled with the length $\ell(e)$ of the corresponding stretch of highway. Naturally, there are acceptable espresso stands at both Giggle offices, represented by two specific vertices s and t in the graph G.

    (a) Describe and analyze an algorithm to determine whether it is possible to bike from the Giggleplex to Gigglesburg without suffering a caffeine-withdrawal migraine.

    (b) You discover that by wearing a more expensive fedora, you can increase the distance L that you can bike between espresso shots. Describe and analyze and algorithm to find the minimum value of L that allows

you to bike from the Giggleplex to Gigglesburg without suffering a caffeine-withdrawal migraine.

(c) When you report to your supervisor (whom Giggle recently hired away from their competitor Ünter) that the ride is impossible, she demands to look at your map. "Oh, I see the problem; there are no *Starbucks* on this map!" As you look on in horror, she hands you an updated graph G' that includes a vertex for every Starbucks location in the United States, helpfully marked in Starbucks Green (Pantone® 3425 C).

Describe and analyze an algorithm to find the minimum number of Starbucks locations you must visit to bike from the Giggleplex to Gigglesburg without suffering a caffeine-withdrawal migraine. More formally, your algorithm should find the minimum number of green vertices on any path in G' from s to t that uses only edges of length at most L.

16. Suppose you are given a directed graph $G = (V, E)$ with non-negatively weighted edges and two vertices s and t. Describe and analyze an algorithm to find the shortest walk in G from s to t (possibly repeating vertices and/or edges) whose number of edges is divisible by 3.

For example, given the graph shown below, with the indicated vertices s and t, and with all edges having weight 1, your algorithm should return 6, which is the length of the walk $s \to w \to y \to x \to s \to w \to t$ has length 6.

17. Suppose you are given a directed graph G with non-negatively weighted edges, where some edges are red and the remaining edges are blue. Describe an algorithm to find the shortest walk in G from one vertex s to another vertex t in which no three consecutive edges have the same color. That is, if the walk contains two red edges in a row, the next edge must be blue, and if the walk contains two blue edges in a row, the next edge must be red.

For example, given the following graph as input, where every red edge has weight 1 and every blue edge has weight 2, your algorithm should return the integer 9, because the shortest legal walk from s to t is $s \to a \to b \Rightarrow d \to c \Rightarrow a \to b \to c$.

18. Consider a directed graph G, where each edge has a non-negative weight, and each edge is colored either red, white, or blue. A walk in G is called a *French flag walk* if its sequence of edge colors is red, white, blue, red, white, blue, and so on. More formally, a walk $v_0 \to v_1 \to \cdots \to v_k$ is a French flag walk if, for every integer i, the edge $v_i \to v_{i+1}$ is red if $i \bmod 3 = 0$, white if $i \bmod 3 = 1$, and blue if $i \bmod 3 = 2$.

    Describe an algorithm to find the *shortest* French flag walks from one starting vertex s to every other vertex in G.

19. There are n galaxies connected by m intergalactic teleport-ways. Each teleport-way joins two galaxies and can be traversed in both directions. Also, each teleport-way e has an associated cost of $c(e)$ dollars, where $c(e)$ is a positive integer. A teleport-way can be used multiple times, but the toll must be paid every time it is used.

    Judy wants to travel from galaxy s to galaxy t as cheaply as possible. However, she wants the total cost to be a multiple of five dollars, because carrying small change is not pleasant either.

    (a) Describe and analyze an algorithm to compute the minimum total cost of traveling from galaxy s to galaxy t, subject to the restriction that the total cost is a multiple of five dollars.

    (b) Solve part (a), but now assume that Judy has a coupon that allows her to use exactly one teleport-way for free.

20. After moving to a new city, you decide to choose a walking route from your home to your new office. To get a good daily workout, your route must consist of an uphill path (for exercise) followed by a downhill path (to cool down), or just an uphill path, or just a downhill path. (You'll walk the same path home, so you'll get exercise one way or the other.) But you also want the *shortest* path that satisfies these conditions, so that you actually get to work on time.

    Your input consists of an undirected graph G, whose vertices represent intersections and whose edges represent road segments, along with a start vertex s and a target vertex t. Every vertex v has an associated value $h(v)$, which is the height of that intersection above sea level, and each edge uv has an associated value $\ell(uv)$, which is the length of that road segment.

    (a) Describe and analyze an algorithm to find the shortest uphill–downhill walk from s to t. Assume all vertex heights are distinct.

    (b) Now suppose we allow some or all vertex heights to be equal. Describe and analyze an algorithm to find the shortest "uphill then downhill" walk from s to t; you may use flat edges in both the "uphill" and "downhill" portions of your walk.

(c) Finally, suppose you discover that there is no path from s to t with the structure you want. Describe an algorithm to find a path from s to t that alternates between "uphill" and "downhill" subpaths as few times as possible, and has minimum length among all such paths.

21. After graduating from Sham-Poobanana University you accept a job with Aerophobes-Я-Us, the leading traveling agency for people who hate to fly. Your job is to build a system to help customers plan airplane trips from one city to another. All of your customers are afraid of flying (and by extension, airports), so any trip you plan needs to be as short as possible. You know all the departure and arrival times of all the flights on the planet.

    Suppose one of your customers wants to fly from city X to city Y. Describe an algorithm to find a sequence of flights that minimizes the *total time in transit*—the length of time from the initial departure to the final arrival, including time at intermediate airports waiting for connecting flights.

22. In Exercise 20 from Chapter 5, you designed an algorithm to decide whether a given *acute-angle maze* is solvable. In this problem, you will design algorithms to find the *shortest* walk through a given acute-angle maze, for two different definitions of "length".

    Complete each angle maze below by tracing a path from start to finish that has only acute angles.

    Your input is a connected undirected graph G whose vertices are points in the plane and whose edges are line segments. Edges do not intersect, except at their endpoints. For example, a drawing of the letter X would have five vertices and four edges; the first maze above has 14 vertices and 21 edges. You are also given two vertices Start and Finish.

    A walk from Start to Finish in G is *valid* if it contains only acute angles, or more formally, for any two consecutive edges $u \to v \to w$, either $\angle uvw = \pi$ or $0 < \angle uvw < \pi/2$. Assume you can determine in $O(1)$ time whether the angle between two given segments is straight, obtuse, right, or acute.

    (a) Describe an algorithm to compute a valid walk from Start to Finish that traverses as few segments as possible. (If your walk traverses the same segment twice, count it twice.)

    (b) Describe an algorithm to compute a valid walk from Start to Finish that makes as few turns as possible. *[Hint: This is **not** the same as part (a).]*

(c) Describe an algorithm to compute a valid walk from Start to Finish whose total Euclidean length is as small as possible. (Assume you can also compute the length of any segment in $O(1)$ time.)

23. After a grueling midterm at the See-Bull Center for Fake News Detection, you decide to take the bus home. Since you planned ahead, you have a schedule that lists the times and locations of every stop of every bus in Sham-Poobanana. Unfortunately, no single bus visits both the See-Bull Center and your home; you must change buses at least once. There are exactly b different buses. Each bus starts at 12:00:01AM, makes exactly n stops, and finally stops running at 11:59:59PM. Buses always run exactly on schedule, and you have an accurate watch. Finally, you are far too tired to walk between bus stops.

    (a) Describe and analyze an algorithm to determine the sequence of bus rides that gets you home as early as possible. Your goal is to minimize your *arrival time*, not the time you spend traveling.

    (b) Oh, no! The midterm was held on Halloween, and the streets are infested with zombies! The Sham-Poobanana Mass Transit District doesn't have the funding to add additional buses or install zombie-proof bus stops, especially for only one night a year. Describe and analyze an algorithm to determine a sequence of bus rides that minimizes *the total time you spend waiting at bus stops*; you don't care how late you get home or how much time you spend on buses. (Assume you can wait inside the See-Bull Center until your first bus is just about to leave.)

24. The first morning after returning from a glorious spring break, Alice wakes to discover that her car won't start, so she has to get to her classes at Sham-Poobanana University by public transit. She has a complete transit schedule for Poobanana County. The bus routes are represented in the schedule by a directed graph G, whose vertices represent bus stops and whose edges represent bus routes between those stops. For each edge $u \to v$, the schedule records three positive real numbers:

    - $\ell(u \to v)$ is the length of the bus ride from stop u to stop v (in minutes)
    - $f(u \to v)$ is the first time (in minutes past 12am) that a bus leaves stop u for stop v.
    - $\Delta(u \to v)$ is the time between successive departures from stop u to stop v (in minutes).

    Thus, the first bus for this route leaves u at time $f(u \to v)$ and arrives at v at time $f(u \to v) + \ell(u \to v)$, the second bus leaves u at time $f(u \to v) + \Delta(u \to v)$ and arrives at v at time $f(u \to v) + \Delta(u \to v) + \ell(u \to v)$, the third bus leaves u at time

$f(u \to v) + 2 \cdot \Delta(u \to v)$ and arrives at v at time $f(u \to v) + 2 \cdot \Delta(u \to v) + \ell(u \to v)$, and so on.

Alice wants to leaves from stop s (her home) at a certain time and arrive at stop t (The See-Bull Center) as quickly as possible. If Alice arrives at a stop on one bus at the exact time that another bus is scheduled to leave, she can catch the second bus. Because she's a student at SPU, Alice can ride the bus for free, so she doesn't care how many times she has to change buses.

Describe and analyze an algorithm to find the earliest time Alice can reach her destination. Your input consists of the directed graph $G = (V, E)$, the vertices s and t, the values $\ell(e), f(e), \Delta(e)$ for each edge $e \in E$, and Alice's starting time (in minutes past 12am).

[Hint: In this rare instance, it may be easier to modify the algorithm, instead of modifying the input graph.]

25. Mulder and Scully have computed, for every road in the United States, the exact probability that someone driving on that road *won't* be abducted by aliens. Agent Mulder needs to drive from Langley, Virginia to Area 51, Nevada. What route should he take so that he has the least chance of being abducted?

More formally, you are given a directed graph $G = (V, E)$, where every edge e has an independent safety probability $p(e)$. The *safety* of a path is the product of the safety probabilities of its edges. Design and analyze an algorithm to determine the safest path from a given start vertex s to a given target vertex t. You may assume that all necessary arithmetic operations can be performed in $O(1)$ time.

For example, with the probabilities shown above, if Mulder tries to drive directly from Langley to Area 51, he has a 50% chance of getting there without being abducted. If he stops in Memphis, he has a $0.7 \times 0.9 = 63\%$ chance of arriving safely. If he stops first in Memphis and then in Las Vegas, he has a $1 - 0.7 \times 0.1 \times 0.5 = 96.5\%$ chance of being abducted! (That's how they got Elvis, you know.)

♣26. On an overnight camping trip in Sunnydale National Park, you are woken from a restless sleep by a scream. As you crawl out of your tent to investigate, a terrified park ranger runs out of the woods, covered in blood and clutching a crumpled piece of paper to his chest. As he reaches your tent, he gasps, "Get out. . . while. . . you. . . ", thrusts the paper into your hands, and falls to the ground. Checking his pulse, you discover that the ranger is stone dead.

You look down at the paper and recognize a map of the park, drawn as an undirected graph, where vertices represent landmarks in the park, and edges represent trails between those landmarks. (Trails start and end at landmarks and do not cross.) You recognize one of the vertices as your current location; several vertices on the boundary of the map are labeled EXIT.

On closer examination, you notice that someone (perhaps the poor dead park ranger) has written a real number between 0 and 1 next to each vertex and each edge. A scrawled note on the back of the map indicates that a number next to an edge is the probability of encountering a vampire along the corresponding trail, and a number next to a vertex is the probability of encountering a vampire at the corresponding landmark. (Vampires can't stand each other's company, so you'll never see more than one vampire on the same trail or at the same landmark.) The note warns you that stepping off the marked trails will result in a slow and painful death.

You glance down at the corpse at your feet. Yes, his death certainly looked painful. Wait, was that a twitch? Are his teeth getting longer? After driving a tent stake through the undead ranger's heart, you wisely decide to *immediately* leave the park as fast as possible.

Describe and analyze an efficient algorithm to find a path from your current location to an arbitrary EXIT node, such that the total *expected number* of vampires encountered along the path is as small as possible. Be sure to account for *both* the vertex probabilities *and* the edge probabilities. *[Hint: Even without the vertex probabilities, this is not the same as the previous problem!]*

The tree which fills the arms grew from the tiniest sprout;
the tower of nine storeys rose from a (small) heap of earth;
the journey of a thousand li commenced with a single step.

— Lao-Tzu, *Tao Te Ching*, chapter 64 (6th century BCE),
translated by James Legge (1891)

And I would walk five hundred miles,
And I would walk five hundred more,
Just to be the man who walks a thousand miles
To fall down at your door.

— The Proclaimers, "I'm Gonna Be (500 Miles)",
Sunshine on Leith (2001)

Almost there… Almost there…

— Red Leader [Drewe Henley], *Star Wars* (1977)

9

All-Pairs Shortest Paths

9.1 Introduction

In the previous chapter, we discussed several algorithms to find the shortest paths from a single source vertex s to every other vertex of the graph, by constructing a shortest path tree rooted at s. The shortest path tree specifies two pieces of information for each node v in the graph:

- $dist(v)$ is the length of the shortest path from s to v;
- $pred(v)$ is the second-to-last vertex in the shortest path from s to v.

In this chapter, we consider the more general *all pairs shortest path* problem, which asks for the shortest path from *every* possible source to every possible destination. For every pair of vertices u and v, we want to compute the following information:

- $dist(u, v)$ is the length of the shortest path from u to v;
- $pred(u, v)$ is the second-to-last vertex on the shortest path from u to v.

These intuitive definitions exclude a few boundary cases, all of which we already saw in the previous chapter.

- If there is no path from u to v, then there is no *shortest* path from u to v; in this case, we define $dist(u, v) = \infty$ and $pred(u, v) = \text{Null}$.
- If there is a negative cycle between u and v, then there are paths[1] from u to v with arbitrarily negative length; in this case, we define $dist(u, v) = -\infty$ and $pred(u, v) = \text{Null}$.
- Finally, if u does not lie on a negative cycle, then the shortest path from u to itself has no edges, and therefore doesn't have a last edge; in this case, we define $dist(u, u) = 0$ and $pred(u, u) = \text{Null}$.

The desired output of the all-pairs shortest path problem is a pair of $V \times V$ arrays, one storing all V^2 shortest-path distances,[2] the other storing all V^2 predecessors. In this chapter, I'll focus almost exclusively on computing the distance array. The predecessor array, from which we can compute the actual shortest paths, can be computed with only minor modifications (hint, hint).

9.2 Lots of Single Sources

The most obvious solution to the all-pairs shortest path problem is to run a single-source shortest path algorithm V times, once for each possible source vertex. Specifically, to fill the one-dimensional subarray $dist[s, \cdot]$, we invoke a single-source algorithm starting at the source vertex s.

```
ObviousAPSP(V, E, w):
  for every vertex s
      dist[s, ·] ← SSSP(V, E, w, s)
```

The running time of this algorithm obviously depends on which single-source shortest path algorithm we use. Just as in the single-source setting, there are four natural options, depending on the structure of the graph and its edge weights:

- If the edges of the graph are unweighted, breadth-first search gives us an overall running time of $\boldsymbol{O(VE)} = O(V^3)$.
- If the graph is acyclic, scanning the vertices in topological order also gives us an overall running time of $\boldsymbol{O(VE)} = O(V^3)$.

[1] formally, walks

[2] Back when road maps used to be printed on paper and had to be searched manually, it was fairly common for them to include a triangular "distance table". To find the distance from Champaign to Columbus, for example, you would look in the row labeled "Champaign" and the column labeled "Columbus".

- If all edge weights are non-negative, Dijkstra's algorithm gives us a running time to $\boldsymbol{O(VE \log V)} = O(V^3 \log V)$.[3]
- Finally, in the most general setting, the Bellman-Ford algorithm gives us an overall running time of $\boldsymbol{O(V^2E)} = O(V^4)$.

9.3 Reweighting

Negative edges slow us down significantly; can we get rid of them? One simple idea that occurs to many people is increasing the weights of all the edges by the same amount so that all the weights become positive, so that we can use Dijkstra's algorithm instead of Bellman-Ford. Unfortunately, this simple idea doesn't work, intuitively because our two natural notions of "length" are incompatible—paths with more edges can have smaller total weight than paths with fewer edges. If we increase all edge weights at the same rate, paths with more edges get longer faster than paths with fewer edges; as a result, the shortest path between two vertices might change.

Figure 9.1. Increasing all the edge weights by 2 changes the shortest path from s to t.

However, there is a more subtle method for reweighting edges that does preserve shortest paths. This reweighting method is often attributed to Donald Johnson, who described its application to shortest path algorithms in 1973. But in fact, Johnson attributed the method to a 1972 paper of Jack Edmonds and Richard Karp. The same method was also described by Nobuaki Tomizawa in 1971, and in a slightly different form by Delbert Fulkerson in 1961.

Suppose each vertex v has some associated *price* $\pi(v)$, which might be positive, negative, or zero. We can define a new weight function w' as follows:

$$w'(u \to v) = \pi(u) + w(u \to v) - \pi(v)$$

To give some intuition, imagine that when we leave vertex u, we have to pay an exit tax of $\pi(u)$, and when we enter v, we get $\pi(v)$ as an entrance gift.

It's not hard to show that shortest paths with the new weight function w' are exactly the same as shortest paths with the original weight function w. In

[3]Again, if we replace the binary heap in our implementation of Dijkstra's algorithm with an unsorted array, the overall running time becomes $O(V^3)$ (no matter how many edges the graph has), and if we replace the binary heap with a Fibonacci heap, the running time drops to $O(V(E + V \log V)) = O(VE + V^2 \log V) = O(V^3)$.

fact, for *any* path $u \rightsquigarrow v$ from one vertex u to another vertex v, we have

$$w'(u \rightsquigarrow v) = \pi(u) + w(u \rightsquigarrow v) - \pi(v).$$

We pay $\pi(u)$ in exit fees, plus the original weight of of the path, minus the $\pi(v)$ entrance gift. At every intermediate vertex x on the path, we get $\pi(x)$ as an entrance gift, but then immediately pay it back as an exit tax! Since all paths from u to v change length by exactly the same amount, the shortest path from u to v does not change. (Paths between different pairs of vertices could change lengths by different amounts, so their order could change.)

9.4 Johnson's Algorithm

Johnson's all-pairs shortest path algorithm computes a cost $\pi(v)$ for each vertex, so that the new weight of every edge is non-negative, and then computes shortest paths with respect to the new weights using Dijkstra's algorithm.

First, suppose the input graph has a vertex s that can reach *all* the other vertices. Johnson's algorithm computes the shortest paths from s to the other vertices, using Bellman-Ford (which doesn't care if the edge weights are negative), and then reweights the graph using the price function $\pi(v) = dist(s, v)$. The new weight of every edge is

$$w'(u \to v) = dist(s, u) + w(u \to v) - dist(s, v).$$

These new weights are non-negative *because* Bellman-Ford halted! Recall that an edge $u \to v$ is *tense* if $dist(s, u) + w(u \to v) < dist(s, v)$, and that single-source shortest path algorithms eliminate all tense edges. (If Bellman-Ford detects a negative cycle, Johnson's algorithm aborts, because shortest paths are not well-defined.)

If there is no suitable vertex s that can reach everything, then no matter where we start Bellman-Ford, some of the resulting vertex prices will be infinite. To avoid this issue, we *always* add a new vertex s to the graph, with zero-weight edges from s to the other vertices, but *no* edges going back into s. This addition doesn't change the shortest paths between any pair of original vertices, because there are no paths into s.

Complete pseudocode for Johnson's algorithm is shown in Figure 9.2. The running time of this algorithm is dominated by the calls to Dijkstra's algorithm. Specifically, we spend $O(VE)$ time running BellmanFord once, $O(VE \log V)$ time running Dijkstra V times, and $O(V + E)$ time doing other bookkeeping. Thus, the overall running time is $\boldsymbol{O(VE \log V)} = O(V^3 \log V)$.[4] Negative edges don't slow us down after all!

[4]. . . assuming the default binary-heap implementation; see the previous footnote.

```
JohnsonAPSP(V, E, w):
  ⟨⟨Add an artificial source⟩⟩
  add a new vertex s
  for every vertex v
      add a new edge s→v
      w(s→v) ← 0
  ⟨⟨Compute vertex prices⟩⟩
  dist[s, ·] ← BellmanFord(V, E, w, s)
  if BellmanFord found a negative cycle
      fail gracefully
  ⟨⟨Reweight the edges⟩⟩
  for every edge u→v ∈ E
      w′(u→v) ← dist[s, u] + w(u→v) − dist[s, v]
  ⟨⟨Compute reweighted shortest path distances⟩⟩
  for every vertex u
      dist′[u, ·] ← Dijkstra(V, E, w′, u)
  ⟨⟨Compute original shortest-path distances⟩⟩
  for every vertex u
      for every vertex v
          dist[u, v] ← dist′[u, v] − dist[s, u] + dist[s, v]
```

Figure 9.2. Johnson's all-pairs shortest paths algorithm

9.5 Dynamic Programming

We can also solve the all-pairs shortest path problem directly using dynamic programming, instead of invoking a single-source algorithm. For *dense* graphs, where $E = \Omega(V^2)$, the dynamic programming approach eventually yields an algorithm that is both simpler and (slightly) faster than Johnson's algorithm. **For the rest of this chapter, I will assume that the input graph contains no negative cycles.**

As usual for dynamic programming algorithms, we first need a recurrence. Just as in the single-source setting, the "obvious" recursive definition

$$dist(u,v) = \begin{cases} 0 & \text{if } u = v \\ \min\limits_{x \to v} \big(dist(u,x) + w(x \to v) \big) & \text{otherwise} \end{cases}$$

only works when the input graph is a dag; any directed cycles drive the recurrence into an infinite loop.

We can break this infinite loop by introducing as an additional parameter, exactly as we did for Bellman-Ford; let $dist(u, v, \ell)$ denote the length of the shortest path from u to v that uses *at most* ℓ *edges*. The shortest path between any two vertices traverses at most $V - 1$ edges, so the true shortest-path distance is $dist(u, v, V - 1)$. Bellman's single-source recurrence adapts to this setting

immediately:

$$dist(u,v,\ell) = \begin{cases} 0 & \text{if } \ell = 0 \text{ and } u = v \\ \infty & \text{if } \ell = 0 \text{ and } u \neq v \\ \min \left\{ \begin{array}{c} dist(u,v,\ell-1) \\ \min\limits_{x \to v} \left(dist(u,x,\ell-1) + w(x \to v) \right) \end{array} \right\} & \text{otherwise} \end{cases}$$

Turning this recurrence into a dynamic programming algorithm is straightforward; the resulting algorithm runs in $\boldsymbol{O(V^2E)} = O(V^4)$ time.

```
ShimbelAPSP(V, E, w):
  for all vertices u
      for all vertices v
          if u = v
              dist[u, v, 0] ← 0
          else
              dist[u, v, 0] ← ∞
  for ℓ ← 1 to V − 1
      for all vertices u
          for all vertices v ≠ u
              dist[u, v, ℓ] ← dist[u, v, ℓ − 1]
              for all edges x→v
                  if dist[u, v, ℓ] > dist[u, x, ℓ − 1] + w(x→v)
                      dist[u, v, ℓ] ← dist[u, x, ℓ − 1] + w(x→v)
```

This algorithm was first sketched by Alfonso Shimbel in 1954.[5] Just like Bellman's formulation of Bellman-Ford, we don't need the inner loop over vertices v or the iteration index ℓ. The modified algorithm is shown below.

```
AllPairsBellmanFord(V, E, w):
  for all vertices u
      for all vertices v
          if u = v
              dist[u, v] ← 0
          else
              dist[u, v] ← ∞
  for ℓ ← 1 to V − 1
      for all vertices u
          for all edges x→v
              if dist[u, v] > dist[u, x] + w(x→v)
                  dist[u, v] ← dist[u, x] + w(x→v)
```

[5]Shimbel assumed the input was a complete $V \times V$ matrix of distances, so his original algorithm actually runs in $O(V^4)$ time no matter how many edges the graph has.

Given how we derived it, it should come as no surprise that the resulting algorithm is exactly the same as interleaving V different executions of Bellman-Ford, each with a different source vertex. In particular, for all vertices u and v, after the ℓth iteration of the main for-loop, $dist[u, v]$ is *at most* the length of the shortest path from u to v containing at most ℓ edges.

9.6 Divide and Conquer

But we can make a more significant improvement, suggested by Michael Fischer and Albert Meyer in 1971. Bellman's recurrence breaks the shortest path into a slightly shorter path and a single edge, by considering all possible predecessors of the target vertex. Instead, let's break the shortest paths into two shorter shortest paths at the *middle* vertex. This idea gives us a different recurrence for the same function $dist(u, v, \ell)$. Here we need to stop at the base case $\ell = 1$ instead of $\ell = 0$, because a path with at most one edge has no "middle" vertex. To simplify the recurrence slightly, let's define $w(v \to v) = 0$ for every vertex v.

$$dist(u, v, \ell) = \begin{cases} w(u \to v) & \text{if } i = 1 \\ \min\limits_{x} \big(dist(u, x, \ell/2) + dist(x, v, \ell/2) \big) & \text{otherwise} \end{cases}$$

As stated, this recurrence only works when ℓ is a power of 2, since otherwise we might try to find the shortest path with (at most) a fractional number of edges! But that's not really a problem; $dist(u, v, \ell)$ is the true shortest-path distance from u to v *for all* $\ell \geq V - 1$; in particular, we can use $\ell = 2^{\lceil \lg V \rceil} < 2V$.

Once again, a dynamic programming solution is straightforward. Even before we write down the algorithm, we can tell the running time is $\boldsymbol{O(V^3 \log V)}$—we need to consider V possible values of u, v, and x, but only $\lceil \lg V \rceil$ possible values of ℓ. In the following pseudocode for Fischer and Meyer's algorithm, the array entry $dist[u, v, i]$ stores the value of $dist(u, v, 2^i)$.

```
FischerMeyerAPSP(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v, 0] ← w(u→v)

  for i ← 1 to ⌈lg V⌉            ⟨⟨ℓ = 2^i⟩⟩
      for all vertices u
          for all vertices v
              dist[u, v, i] ← ∞
              for all vertices x
                  if dist[u, v, i] > dist[u, x, i−1] + dist[x, v, i−1]
                      dist[u, v, i] ← dist[u, x, i−1] + dist[x, v, i−1]
```

Unlike our earlier algorithms, FischerMeyerAPSP is *not* the same as V invocations of any single-source shortest-path algorithm; in particular, the

innermost loop does *not* simply relax tense edges. Nevertheless, we can still remove the last dimension of the table, using $dist[u, v]$ everywhere in place of $dist[u, v, i]$, just as we did in Bellman-Ford and our earlier dynamic programming algorithm; this reduces the space from $O(V^3)$ to $O(V^2)$. This more polished algorithm was described by Leyzorek *et al.* in 1957, in the same paper where they describe Dijkstra's algorithm.

```
LeyzorekAPSP(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v] ← w(u→v)
  for i ← 1 to ⌈lg V⌉            ⟨⟨ℓ = 2^i⟩⟩
      for all vertices u
          for all vertices v
              for all vertices x
                  if dist[u, v] > dist[u, x] + dist[x, v]
                      dist[u, v] ← dist[u, x] + dist[x, v]
```

9.7 Funny Matrix Multiplication

There is a very close connection (first observed by Shimbel, and later independently by Bellman) between computing shortest paths in a directed graph and computing powers of a square matrix. Compare the following algorithm for squaring an $n \times n$ matrix A with the inner loop of FischerMeyerAPSP. (I've slightly modified the notation in the second algorithm to make the similarity clearer.)

```
MatrixSquare(A):
  for i ← 1 to n
      for j ← 1 to n
          A′[i, j] ← 0
          for k ← 1 to n
              A′[i, j] ← A′[i, j] + A[i, k] · A[k, j]
```

```
FischerMeyerInnerLoop(D):
  for all vertices u
      for all vertices v
          D′[u, v] ← ∞
          for all vertices x
              D′[u, v] ← min {D′[u, v], D[u, x] + D[x, v]}
```

The *only* difference between these two algorithms is that the second algorithm uses addition instead of multiplication, and minimization instead of addition. For this reason, the shortest path inner loop is sometimes referred to as "min-plus" or "distance" or "funny" matrix multiplication.

Our slower algorithm ShimbelAPSP is the standard iterative algorithm for computing the $(V-1)$th "min-plus power" of the weight matrix w. The first set of loops sets up the min-plus identity matrix, with 0s on the main diagonal and ∞ everywhere else, and each iteration of the second main loop computes the next "min-plus power". FischerMeyerAPSP replaces this iterative method for computing powers with repeated squaring, exactly as we saw at the end of Chapter 1. Once again, we see the influence of ancient Egyptian *ἀρπεδονάπται*!

There are faster divide-and-conquer algorithms for (standard) matrix multiplication, similar to Karatsuba's divide-and-conquer algorithm for multiplying integers. The first such algorithm, described by Volker Strassen in 1969, reduces the problem of multiplying two $n \times n$ matrices to *seven* instances of multiplying two $n/2 \times n/2$ matrices; Strassen's algorithm runs in $O(n^{\lg 7}) = O(n^{2.807355})$. Strassen's algorithm has been improved many times over the last fifty years; as of 2018, the fastest matrix-multiplication algorithm known runs in $O(n^{2.372864})$ time.[6] Unfortunately, *all* of these faster algorithms use subtraction, and there's no "funny" equivalent of subtraction. (What's the inverse operation for min?) So at least for general graphs, there's no obvious way to speed up the inner loop of our dynamic programming algorithms.

But "not obvious" does not mean "impossible"! In fact, there are several significantly faster algorithms for *special cases* of the all-pairs shortest paths problem. One of the nicest is a simple randomized algorithm discovered in 1991 by Zvi Galil and Oded Margalit, and further simplified in 1992 by Raimund Seidel, that computes all-pairs shortest path *distances* in *unweighted, undirected* graphs in $O(M(V)\log V)$ *expected* time, where $M(n) = O(n^{2.372864})$ is the time required to (seriously) multiply two $n \times n$ integer matrices.[7] Galil, Margalit, and Seidel's approach has since been extended to compute actual shortest paths, deterministically, in directed graphs, with small integer edge weights, in strongly subcubic time.

On the other hand, despite considerable progress in the small-integer-weight setting, nobody knows how to compute all-pairs shortest paths for more general edge weights in $O(V^{2.999999})$ time, for any number of 9s. Moreover, there is some evidence that such an algorithm is impossible! So maybe "not obvious" does mean "impossible" after all.

[6]Determining the minimum time required to multiply two arbitrary $n \times n$ matrices is a long-standing open problem; many people believe there is an undiscovered algorithm that runs in $O(n^{2+\varepsilon})$ time for any $\varepsilon > 0$, or possibly even in $O(n^2)$ time.

[7]Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400-403, 1995. This is one of the few algorithms papers where (in the 1992 conference version at least) the algorithm is completely described and analyzed *in the abstract* of the paper. See also: Noga Alon, Zvi Galil, Oded Margalit*. On the exponent of the all pairs shortest path problem. *Journal of Computer and System Sciences* 54(2):255–262, 1997.

9.8 (Kleene-Roy-)Floyd-Warshall(-Ingerman)

Our fast dynamic programming algorithm is still a factor of $O(\log V)$ slower in the worst case than the standard implementation of Johnson's algorithm. A different formulation of shortest paths that removes this logarithmic factor was proposed twice in 1962, first by Robert Floyd and later independently by Peter Ingerman, both slightly generalizing an algorithm of Stephen Warshall published earlier in the same year. In fact, Warshall's algorithm was previously discovered by Bernard Roy in 1959, and the underlying recursion pattern was used by Stephen Kleene[8] in 1951.

Warshall's (and Roy's and Kleene's) insight was to use a different third parameter in the dynamic programming recurrence. Instead of considering paths with a limited number of edges, they considered paths that can pass through only certain vertices. Here, "pass through" means "both enter and leave"; for example, the path $w \to x \to y \to z$ starts at w, *passes through* x and y, and ends at z.

Number the vertices arbitrarily from 1 to V. For every pair of vertices u and v and every integer r, we define a path $\pi(u, v, r)$ as follows:

> $\pi(u, v, r)$ is the shortest path (if any) from u to v that passes through only vertices numbered at most r.

In particular, $\pi(u, v, V)$ is the true shortest path from u to v. Kleene and Roy and Warshall all observed that these paths have a simple recursive structure.

Figure 9.3. Recursive structure of the restricted shortest path $\pi(u, v, r)$.

- The path $\pi(u, v, 0)$ can't pass through any intermediate vertices, so it must be the edge (if any) from u to v.
- For any integer $r > 0$, either $\pi(u, v, r)$ passes through vertex r or it doesn't.
  - If $\pi(u, v, r)$ passes through vertex r, it consists of a subpath from u to r, followed by a subpath from r to v. Both of those subpaths pass through only vertices numbered at most $r - 1$; moreover, those subpaths are as short as possible with this restriction. So the two subpaths must be $\pi(u, r, r - 1)$ and $\pi(r, v, r - 1)$.

[8] Pronounced "clay knee", not "clean" or "clean-ee" or "clay-nuh" or "dimaggio". Specifically, Kleene described an inductive proof that every finite automata has an equivalent regular expression; Kleene's induction pattern is essentially identical to the Floyd-Warshall recurrence.

– On the other hand, if $\pi(u, v, r)$ does not pass through vertex r, then it passes through only vertices numbered at most $r-1$, and it must be the *shortest* path with this restriction. So in this case, we must have $\pi(u, v, r) = \pi(u, v, r-1)$.

Now let $dist(u, v, r)$ denote the *length* of the path $\pi(u, v, r)$. The recursive structure of $\pi(u, v, r)$ immediately implies the following recurrence:

$$dist(u, v, r) = \begin{cases} w(u \to v) & \text{if } r = 0 \\ \min \left\{ \begin{array}{c} dist(u, v, r-1) \\ dist(u, r, r-1) + dist(r, v, r-1) \end{array} \right\} & \text{otherwise} \end{cases}$$

Our goal is to compute $dist(u, v, V)$ for all vertices u and v. Once again, this recurrence can be evaluated by a straightforward dynamic programming algorithm in $\boldsymbol{O(V^3)}$ ***time***.

```
KleeneAPSP(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v, 0] ← w(u→v)

  for r ← 1 to V
      for all vertices u
          for all vertices v
              if dist[u, v, r−1] < dist[u, r, r−1] + dist[r, v, r−1]
                  dist[u, v, r] ← dist[u, v, r−1]
              else
                  dist[u, v, r] ← dist[u, r, r−1] + dist[r, v, r−1]
```

Like all our previous dynamic programming algorithms for shortest paths, we can simplify KleeneAPSP by removing the third dimension of the memoization table. Also, because we chose the vertex numbering arbitrarily, there's no reason to refer to it explicitly in the pseudocode. We finally arrive at Floyd's improvement of Warshall's algorithm:

```
FloydWarshall(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v] ← w(u→v)

  for all vertices r
      for all vertices u
          for all vertices v
              if dist[u, v] > dist[u, r] + dist[r, v]
                  dist[u, v] ← dist[u, r] + dist[r, v]
```

It's interesting to compare FloydWarshall with our earlier, slightly slower dynamic programming algorithm LeyzorekAPSP. Instead of $O(\log V)$ passes

through all triples of vertices, FloydWarshall requires only a single pass, but only because it uses a different nesting order for the three loops!

Exercises

1. (a) Describe a modification of LeyzorekAPSP that returns an array of predecessor pointers, in addition to the array of shortest path distances, still in $O(V^3 \log V)$ time.

   (b) Describe a modification of FloydWarshall that returns an array of predecessor pointers, in addition to the array of shortest path distances, still in $O(V^3)$ time.

2. All of the algorithms discussed in this chapter fail if the graph contains a negative cycle. Johnson's algorithm detects the negative cycle in the initialization phase (via Bellman-Ford) and aborts; the dynamic programming algorithms just return incorrect results. However, *all* of these algorithms can be modified to return correct shortest-path distances, even in the presence of negative cycles. Specifically, for all vertices u and v:

   - If u cannot reach v, the algorithm should return $dist[u, v] = \infty$.
   - If u can reach a negative cycle that can reach v, the algorithm should return $dist[u, v] = -\infty$.
   - Otherwise, there is a shortest path from u to v, so the algorithm should return its length.

   (a) Describe how to modify Johnson's algorithm to return the correct shortest-path distances, even if the graph has negative cycles.

   (b) Describe how to modify LeyzorekAPSP to return the correct shortest-path distances, even if the graph has negative cycles.

   (c) Describe how to modify Floyd-Warshall to return the correct shortest-path distances, even if the graph has negative cycles.

3. The algorithms described in this chapter can also be modified to return an explicit description of some negative cycle in the input graph G, if one exists, instead of only reporting whether or not G contains a negative cycle.

   (a) Describe how to modify Johnson's algorithm to return either the array of all shortest-path distances or a negative cycle.

   (b) Describe how to modify LeyzorekAPSP to return either the array of all shortest-path distances or a negative cycle.

   (c) Describe how to modify Floyd-Warshall to return either the array of all shortest-path distances or a negative cycle.

In all cases, if the input graph contains more than one negative cycle, your algorithms may choose one arbitrarily.

4. Let $G = (V, E)$ be a directed graph with weighted edges; edge weights can be positive, negative, or zero, but there are no negative cycles.
   (a) Describe an efficient algorithm that either finds a cycle of length zero in G, or correctly reports that no such cycle exists.
   (b) Describe an efficient algorithm that constructs a subgraph H of G with the following properties:
      - Every vertex of G is a vertex of H.
      - Every directed cycle in H has length 0.
      - Every directed cycle of length 0 in G is also a cycle in H.

      In particular, if there are no zero-cycles in G, then H has no edges.

5. Let $G = (V, E)$ be a directed graph with weighted edges; edge weights can be positive, negative, or zero. Suppose the vertices of G are partitioned into k disjoint subsets $V_1, V_2, \ldots, V_k$; that is, every vertex of G belongs to exactly one subset V_i. For each i and j, let $\delta(i, j)$ denote the minimum shortest-path distance between vertices in V_i and vertices in V_j:

$$\delta(i, j) = \min \left\{ dist(v_i, v_j) \;\middle|\; v_i \in V_i \text{ and } v_j \in V_j \right\}.$$

   Describe an algorithm to compute $\delta(i, j)$ for all i and j. For full credit, your algorithm should run in $O(VE + kV \log V)$ time.

6. In this problem we will discover how you, yes ***you***, can be employed by Wall Street and cause a major economic collapse! The *arbitrage* business is a money-making scheme that takes advantage of differences in currency exchange. In particular, suppose 1 US dollar buys 120 Japanese yen, 1 yen buys 0.01 euros, and 1 euro buys 1.2 US dollars. Then, a trader starting with \$1 can convert their money from dollars to yen, then from yen to euros, and finally from euros back to dollars, ending with \$1.44! The cycle of currencies \$ → ¥ → € → \$ is called an ***arbitrage cycle***. Of course, finding and exploiting arbitrage cycles before the prices are corrected requires extremely fast algorithms.

   Suppose n different currencies are traded in your currency market. You are given the matrix $Exch[1..n, 1..n]$ of exchange rates between every pair of currencies; for each i and j, one unit of currency i can be traded for $Exch[i, j]$ units of currency j. (Do *not* assume that $Exch[i, j] \cdot Exch[j, i] = 1$.)
   (a) Describe an algorithm that returns an array $MaxAmt[1..n]$, where $MaxAmt[i]$ is the maximum amount of currency i that you can obtain

by trading, starting with one unit of currency 1, assuming there are no arbitrage cycles.

(b) Describe an algorithm to determine whether the given matrix of currency exchange rates creates an arbitrage cycle.

(c) Modify your algorithm from part (b) to actually return an arbitrage cycle, if it exists.

7. Morty needs to retrieve a stabilized plumbus from the Clackspire Labyrinth. He must enter the labyrinth using Rick's interdimensional portal gun, traverse the Labyrinth to a plumbus, then take that plumbus through the Labyrinth to a fleeb to be stabilized, and finally take the stabilized plumbus back to the original portal to return home. Plumbuses are stabilized by fleeb juice, which any fleeb will release immediately after being removed from its fleebhole. An unstabilized plumbus will explode if it is carried more than 137 flinks from its original storage unit. The Clackspire Labyrinth smells like farts, so Morty wants to spend as little time there as possible.

   Rick has given Morty a detailed map of the Clackspire Labyrinth, which consist of a directed graph $G = (V, E)$ with non-negative edge weights (indicating distance in flinks), along with two disjoint subsets $P \subset V$ and $F \subset V$, indicating the plumbus storage units and fleebholes, respectively. Morty needs to identify a start vertex s, a plumbus storage unit $p \in P$, and a fleebhole $f \in F$, such that the shortest-path distance from p to f is at most 137 flinks long, and the length of the shortest walk $s \rightsquigarrow p \rightsquigarrow f \rightsquigarrow s$ is as short as possible.

   Describe and analyze an algo(burp)rithm to so(burp)olve Morty's problem. You can assume that it is in fact possible for Morty to succeed.

8. Let $G = (V, E)$ be a directed graph with weighted edges; edge weights could be positive, negative, or zero.

   (a) How would we delete an arbitrary vertex v from this graph, without changing the shortest-path distance between any other pair of vertices? Describe an algorithm that constructs a directed graph $G' = (V \setminus \{v\}, E')$ with weighted edges, such that the shortest-path distance between any two vertices in G' is equal to the shortest-path distance between the same two vertices in G, in $O(V^2)$ time.

   (b) Now suppose we have already computed all shortest-path distances in G'. Describe an algorithm to compute the shortest-path distances in the original graph G from v to every other vertex, and from every other vertex to v, all in $O(V^2)$ time.

(c) Combine parts (a) and (b) into another all-pairs shortest path algorithm that runs in $O(V^3)$ time. (The resulting algorithm is *almost* the same as Floyd-Warshall!)

9. Suppose A and B are *boolean* $n \times n$ matrices. The *boolean* or *and-or* product of A and B is the $n \times n$ matrix C defined as follows:

$$C[i,j] := \bigvee_k \big(A[i,k] \wedge B[k,j]\big)$$

(a) Reduce boolean matrix multiplication to min-plus matrix multiplication. That is, given a subroutine MinPlusMultiply that computes the min-plus product of two $n \times n$ matrices in $T(n)$ time, describe and analyze an algorithm BooleanMatrixMultiply that multiplies two boolean matrices in $O(T(n))$ time.

(b) Reduce boolean matrix multiplication to standard matrix multiplication. That is, given a subroutine MatrixMultiply that computes the standard product of two $n \times n$ matrices in $T(n)$ time, describe and analyze an algorithm BooleanMatrixMultiply that multiplies two boolean matrices in $O(T(n))$ time.

10. The *transitive closure* of a directed graph G contains an edge $u \to v$ if and only if there is a directed path from u to v in G. For this problem, assume we can multiply two $n \times n$ boolean matrices in $O(n^\omega)$ time, for some constant $2 \le \omega < 3$. (Problem 9(b) implies $\omega \le 2.372864$.)

(a) Describe an algorithm to compute the transitive closure of an n-vertex directed graph in $O(n^\omega \log n)$ time.

(b) Now suppose G is a directed *acyclic* graph. Describe an algorithm to compute the transitive closure of G in $O(n^\omega)$ time. *[Hint: Do what you always do with dags, and then divide and conquer. Use the fact that $\omega \ge 2$.]*

(c) Finally, describe an algorithm to compute the transitive closure of an *arbitrary* directed graph in $O(n^\omega)$ time. *[Hint: Do what you always do to turn an arbitrary directed graph into a dag.]*

(d) Now let's reverse the previous reduction. Given a subroutine TransitiveClosure that computes the transitive closure of an n-vertex directed graph in $O(n^\alpha)$ time, for some constant $2 \le \alpha < 3$, describe and analyze an algorithm for boolean matrix multiplication that runs in $O(n^\alpha)$ time.

11. Prove that the following recursive algorithm correctly computes all-pairs shortest-path distances in $O(n^3)$ time. For simplicity, you may assume n is a power of 2. As usual, the array D is passed *by reference* to the helper

function RecAPSP. *[Hint: This is a jumbled version of Floyd-Warshall, with significantly better cache behavior.[9]]*

```
RecursiveAPSP(V, E, w):
  n ← |V|
  for i ← 1 to n
      for j ← 1 to n
          if i = j
              D[i, j] ← 0
          if i→j ∈ E
              D[i, j] ← w(i→j)
          else
              D[i, j] ← ∞
  RecAPSP(D, n, 1, 1, 1)
  return D[1..n, 1..n]
```

```
RecAPSP(D, n, i, j, k):
  if n = 1
      D[i, j] ← min{D[i, j], D[i, k] + D[j, k]}
  else
      m ← n/2
      RecAPSP(D, n/2, i,     j,     k     )
      RecAPSP(D, n/2, i,     j,     k + m)
      RecAPSP(D, n/2, i,     j + m, k     )
      RecAPSP(D, n/2, i,     j + m, k + m)
      RecAPSP(D, n/2, i + m, j,     k     )
      RecAPSP(D, n/2, i + m, j,     k + m)
      RecAPSP(D, n/2, i + m, j + m, k     )
      RecAPSP(D, n/2, i + m, j + m, k + m)
```

♥12. Let $G = (V, E)$ be an undirected, unweighted, connected, n-vertex graph, represented by an adjacency matrix $A[1..n, 1..n]$. In this problem, we will derive Seidel's sub-cubic algorithm to compute the $n \times n$ matrix $D[1..n, 1..n]$ of shortest-path distances in G using fast matrix multiplication. Assume that we have a subroutine MatrixMultiply that computes the standard product of two $n \times n$ matrices in $O(n^\omega)$ time, for some unknown constant $\omega \geq 2$.

(a) Let G^2 denote the graph with the same vertices as G, where two vertices are connected by a edge if and only if they are connected by a path of length at most 2 in G. Describe an algorithm to compute the adjacency matrix of G^2 using a single call to MatrixMultiply and $O(n^2)$ additional time.

(b) Suppose we discover that G^2 is a complete graph. Describe an algorithm to compute the matrix D of shortest path distances in G in $O(n^2)$ additional time.

(c) Suppose we recursively compute the matrix D^2 of shortest-path distances in G^2. Prove that the shortest-path distance in G from node i to node j is either $2 \cdot D^2[i, j]$ or $2 \cdot D^2[i, j] - 1$.

(d) Now suppose G^2 is *not* a complete graph. Let $X = D^2 \cdot A$, and let $\deg(i)$ denote the degree of vertex i in the original graph G. Prove that the shortest-path distance from node i to node j in G is $2 \cdot D^2[i, j]$ if and only if $X[i, j] \geq D^2[i, j] \cdot \deg(i)$.

[9] Joon-Sang Park, Michael Penner, and Viktor K. Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Trans. Parallel and Distributed Systems* 15(9):769–782, 2004. For a significant generalization to a wider class of dynamic programming problems, see Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-oblivious dynamic programming. *Proc. 17th SODA* 591–600, 2006.

(e) Describe an algorithm to compute the matrix D of shortest-path distances in G in $O(n^\omega \log n)$ time.

13. Gideon Yuval proposed the following reduction from min-plus matrix multiplication to standard matrix multiplication in 1976. Suppose we are given two integers $n \times n$ matrices A and B of integers, each of whose entries is between 0 and M, and we want to compute their min-plus product matrix C, defined by setting

$$C[i,k] = \min_j (A[i,j] + B[j,k])$$

for all indices i and k. Define two new $n \times n$ matrices A' and B', where

$$A'[i,j] = n^{M-A[i,j]} \qquad \text{and} \qquad B'[i,j] = n^{M-B[i,j]}.$$

Finally, let C' be the (standard) product of A' and B', defined by setting $C'[i,k] = \sum_j A'[i,j] \cdot B'[j,k]$.

(a) Describe an algorithm to construct A' from A using only standard integer arithmetic operations $(+, -, \times)$.

(b) Describe an algorithm to extract the min-plus product C from C', using only standard integer arithmetic operations $(+, -, \times)$.[10]

(c) Suppose we can compute the standard product of two $n \times n$ integer matrices using $O(n^\omega)$ arithmetic operations, for some constant $2 \le \omega < 3$. How many arithmetic operations does Yuval's algorithm need to compute the min-plus product C?

(d) Given a single $n \times n$ integer matrix A, how many arithmetic operations are required to compute the nth "funny" power of A using Yuval's algorithm? (Recall that if A is the weighted adjacency matrix of a graph, then the nth "funny" power of A is the matrix of shortest-path distances.)

(e) Why doesn't Yuval's algorithm imply an all-pairs shortest path algorithm that is faster than Floyd-Warshall for *arbitrary* edge weights? How are we cheating?

[10] In particular, do *not* use logarithms or division or the floor function $\lfloor x \rfloor$. Trust me—this is a can of worms you do *not* want to open.

A process cannot be understood by stopping it. Understanding must move with the flow of the process, must join it and flow with it.

— The First Law of Mentat, in Frank Herbert's *Dune* (1965)

Contrary to expectation, flow usually happens not during relaxing moments of leisure and entertainment, but rather when we are actively involved in a difficult enterprise, in a task that stretches our mental and physical abilities.... Flow is hard to achieve without effort. Flow is not "wasting time."

— Mihaly Csíkszentmihályi, *Flow: The Psychology of Optimal Experience* (1990)

There's a difference between knowing the path and walking the path.

— Morpheus [Laurence Fishburne], *The Matrix* (1999)

10

Maximum Flows & Minimum Cuts

In the mid-1950s, U. S. Air Force researcher Theodore E. Harris and retired U. S. Army general Frank S. Ross wrote a classified report studying the rail network that linked the Soviet Union to its satellite countries in Eastern Europe. The network was modeled as a graph with 44 vertices, representing geographic regions, and 105 edges, representing links between those regions in the rail network. Each edge was given a weight, representing the rate at which material could be shipped from one region to the next. Essentially by trial and error, they determined both the maximum amount of stuff that could be moved from Russia into Europe, as well as the cheapest way to disrupt the network by removing links (or in less abstract terms, blowing up train tracks), which they called "the bottleneck". Their report, which included the drawing of the network in Figure 10.1, was only declassified in 1999.[1]

[1] I learned this story from Alexander Schrijver's fascinating survey "On the history of combinatorial optimization (till 1960)"; the Harris-Ross report was declassified at Schrijver's request. Ford and Fulkerson (who we will meet shortly) credit Harris for formulating the

Figure 10.1. Harris and Ross's map of the Warsaw Pact rail network. (See Image Credits at the end of the book.)

This one of the first recorded applications of the ***maximum flow*** and ***minimum cut*** problems. For both problems, the input is a directed graph $G = (V, E)$ with two special vertices s and t, called the *source* and *target*. As in previous chapters, I will write $\boldsymbol{u \to v}$ to denote the directed edge from vertex u to vertex v. Intuitively, the maximum flow problem asks for the maximum rate at which some resource can be moved from s to t; the minimum cut problem asks for the minimum damage needed to separate s from t.

10.1 Flows

An ***(s, t)-flow*** (or just a ***flow*** if the source and target vertices are clear from context) is a function $f : E \to \mathbb{R}$ that satisfies the following ***conservation constraint*** at every vertex v except possibly s and t:

$$\sum_u f(u \to v) = \sum_w f(v \to w).$$

In English, the total flow into v is equal to the total flow out of v. To keep the notation simple, we define $f(u \to v) = 0$ if there is no edge $u \to v$ in the graph.

The ***value*** of the flow f, denoted $|\boldsymbol{f}|$, is the total net flow out of the source vertex s:

$$|f| := \sum_w f(s \to w) - \sum_u f(u \to s).$$

maximum-flow problem, although the precise chronology is somewhat muddled; Harris and Ross thank George Dantzig "for assistance in formulating the problem".

It's not hard to prove that $|f|$ is also equal to the total net flow *into* the target vertex t, as follows. To simplify notation, let $\boldsymbol{\partial f(v)}$ denote the total net flow out of any vertex v:

$$\partial f(v) := \sum_u f(u \to v) - \sum_w f(v \to w).$$

The conservation constraint implies that $\partial f(v) = 0$ or every vertex v except s and t, so

$$\sum_v \partial f(v) = \partial f(s) + \partial f(t).$$

On the other hand, any flow that leaves one vertex must enter another vertex, so we must have $\sum_v \partial f(v) = 0$. It follows immediately that $|f| = \partial f(s) = -\partial f(t)$.

Now suppose we have another function $c\colon E \to \mathbb{R}_{\geq 0}$ that assigns a non-negative ***capacity*** $c(e)$ to each edge e. We say that a flow f is ***feasible*** (with respect to c) if $0 \leq f(e) \leq c(e)$ for every edge e. Most of the time we consider only flows that are feasible with respect to some fixed capacity function c. We say that a flow f ***saturates*** edge e if $f(e) = c(e)$, and ***avoids*** edge e if $f(e) = 0$. The ***maximum flow problem*** is to compute a feasible (s,t)-flow in a given directed graph, with a given capacity function, whose value is as large as possible.

Figure 10.2. A feasible (s,t)-flow with value 10. Each edge is labeled with its flow/capacity.

10.2 Cuts

An $\boldsymbol{(s,t)}$***-cut*** (or just ***cut*** if the source and target vertices are clear from context) is a partition of the vertices into disjoint subsets S and T—meaning $S \cup T = V$ and $S \cap T = \varnothing$—where $s \in S$ and $t \in T$.

If we have a capacity function $c\colon E \to \mathbb{R}_{\geq 0}$, the ***capacity*** of a cut is the sum of the capacities of the edges that start in S and end in T:

$$\boldsymbol{\|S,T\|} := \sum_{v \in S} \sum_{w \in T} c(v \to w).$$

(Again, if $v \to w$ is not an edge in the graph, we assume $c(v \to w) = 0$.) Notice that the definition is asymmetric; edges that start in T and end in S are unimportant.

The ***minimum cut problem*** is to compute an (s,t)-cut whose capacity is as small as possible.

Figure 10.3. An (s,t)-cut with capacity 15. Each edge is labeled with its capacity.

Intuitively, the minimum cut is the cheapest way to disrupt all flow from s to t. Indeed, it is not hard to show the following relationship between flows and cuts:

Lemma 10.1. *Let f be **any** feasible (s,t)-flow, and let (S,T) be **any** (s,t)-cut. The value of f is at most the capacity of (S,T). Moreover, $|f| = \|S,T\|$ if and only if f saturates every edge from S to T and avoids every edge from T to S.*

Proof: Choose your favorite flow f and your favorite cut (S,T), and then follow the bouncing inequalities:

$$
\begin{aligned}
|f| &= \partial f(s) && \text{[by definition]}\\
&= \sum_{v\in S} \partial f(v) && \text{[conservation constraint]}\\
&= \sum_{v\in S}\sum_{w} f(v\to w) - \sum_{v\in S}\sum_{u} f(u\to v) && \text{[math, definition of } \partial\text{]}\\
&= \sum_{v\in S}\sum_{w\notin S} f(v\to w) - \sum_{v\in S}\sum_{u\notin S} f(u\to v) && \text{[removing edges from } S \text{ to } S\text{]}\\
&= \sum_{v\in S}\sum_{w\in T} f(v\to w) - \sum_{v\in S}\sum_{u\in T} f(u\to v) && \text{[definition of cut]}\\
&\le \sum_{v\in S}\sum_{w\in T} f(v\to w) && \text{[because } f(u\to v)\ge 0\text{]}\\
&\le \sum_{v\in S}\sum_{w\in T} c(v\to w) && \text{[because } f(v\to w)\le c(v\to w)\text{]}\\
&= \|S,T\| && \text{[by definition]}
\end{aligned}
$$

In the second step, we are just adding zeros, because $\partial f(v) = 0$ for every vertex $v \in S \setminus \{s\}$. In the fourth step, we are removing flow values $f(x\to y)$ where

both x and y are in S, because they appear in both sums: positively when $v = x$ and $w = y$, and negatively when $v = y$ and $u = x$.

The first inequalities in this derivation is actually an equality if and only if f avoids every edge from T to S. Similarly, the second inequality is actually an equality if and only if f saturates every edge from S to T. □

This lemma immediately implies that if $|f| = \|S, T\|$, then f must be a maximum flow, and (S, T) must be a minimum cut.

10.3 The Maxflow-Mincut Theorem

Surprisingly, in every flow network, there is a feasible (s, t)-flow f and an (s, t)-cut (S, T) such that $|f| = \|S, T\|$. This is the famous *Maxflow-Mincut Theorem*, first proved by Lester Ford (of shortest-path fame) and Delbert Fulkerson in 1954 and independently by Peter Elias, Amiel Feinstein, and Claude Shannon (of information-theory and maze-solving-robot fame) in 1956.

The Maxflow-Mincut Theorem. *In every flow network with source s and target t, the value of the maximum (s, t)-flow is* equal *to the capacity of the minimum (s, t)-cut.*

Ford and Fulkerson proved this theorem as follows. Fix a graph G, vertices s and t, and a capacity function $c : E \to \mathbb{R}_{\geq 0}$. The proof will be easier if we assume that G is ***reduced***, meaning there is at most one edge between any two vertices u and v. In particular, either $c(u \to v) = 0$ or $c(v \to u) = 0$. This assumption is easy to enforce: Subdivide each edge $u \to v$ in G with a new vertex x, replacing $u \to v$ with a path $u \to x \to v$, and define $c(u \to x) = c(x \to v) = c(u \to v)$. The modified graph has the same maximum flow value and minimum cut capacity as the original graph.

Figure 10.4. Enforcing the one-direction assumption.

Let f be an arbitrary feasible (s, t)-flow in G. We define a new capacity function $c_f : V \times V \to \mathbb{R}$, called the ***residual capacity***, as follows:

$$c_f(u \to v) = \begin{cases} c(u \to v) - f(u \to v) & \text{if } u \to v \in E \\ f(v \to u) & \text{if } v \to u \in E \\ 0 & \text{otherwise} \end{cases}$$

Intuitively, the residual capacity of an edge indicates how much *more* flow can be pushed through that edge. Because $f \geq 0$ and $f \leq c$, these residual capacities are always non-negative. It is possible to have $c_f(u{\to}v) > 0$ even if $u{\to}v$ is not an edge in the original graph G. Thus, we define the ***residual graph*** $G_f = (V, E_f)$, where E_f is the set of edges whose residual capacity is positive. Most residual graphs are *not* reduced; in particular, if $0 < f(u{\to}v) < c(u{\to}v)$, then the residual graph G_f contains both $u{\to}v$ and its reversal $v{\to}u$.

Figure 10.5. A flow f in a weighted graph G and the corresponding residual graph G_f.

Now we have two cases to consider: Either there is a directed path from the source vertex s to the target vertex t in the residual graph G^f, or there isn't.

First suppose the residual graph G_f contains a directed path P from s to t; we call P an ***augmenting path***. Let $F = \min_{u\to v \in P} c_f(u{\to}v)$ denote the maximum amount of flow that we can push through P. We define a new flow $f' \colon E \to \mathbb{R}$ (in the original graph) as follows:

$$f'(u{\to}v) = \begin{cases} f(u{\to}v) + F & \text{if } u{\to}v \in P \\ f(u{\to}v) - F & \text{if } v{\to}u \in P \\ f(u{\to}v) & \text{otherwise} \end{cases}$$

Figure 10.6. An augmenting path with value $F = 5$ and the resulting augmented flow f'.

I claim that this new flow f' is feasible with respect to the original capacities c, meaning $f' \geq 0$ and $f' \leq c$ everywhere. Consider an edge $u{\to}v$ in the original graph G. There are three cases to consider.

- If the augmenting path P contains $u{\to}v$, then

$$f'(u{\to}v) \;=\; f(u{\to}v) + F \;>\; f(u{\to}v) \;\geq\; 0$$

because f is feasible, and

$$\begin{aligned} f'(u{\to}v) &= f(u{\to}v) + F && \text{by definition of } f' \\ &\le f(u{\to}v) + c_f(u{\to}v) && \text{by definition of } F \\ &= f(u{\to}v) + c(u{\to}v) - f(u{\to}v) && \text{by definition of } c_f \\ &= c(u{\to}v) && \text{Duh.} \end{aligned}$$

- If the augmenting path P contains the reversed edge $v{\to}u$, then

$$f'(u{\to}v) \;=\; f(u{\to}v) - F \;<\; f(u{\to}v) \;\le\; c(u{\to}v),$$

again because f is feasible, and

$$\begin{aligned} f'(u{\to}v) &= f(u{\to}v) - F && \text{by definition of } f' \\ &\ge f(u{\to}v) - c_f(v{\to}u) && \text{by definition of } F \\ &= f(u{\to}v) - f(u{\to}v) && \text{by definition of } c_f \\ &= 0 && \text{Duh.} \end{aligned}$$

- Finally, if neither $u{\to}v$ nor $v{\to}u$ is in the augmenting path, then $f'(u{\to}v) = f(u{\to}v)$, and therefore $0 \le f'(u{\to}v) \le c(u{\to}v)$, because f is feasible.

So f is indeed feasible.

Finally, only the first edge in the augmenting path leaves s, which implies $|f'| = |f| + F > |f|$. Thus, f' is a feasible flow with larger value than f. We conclude that if there is a path from s to t in the residual graph G_f, then f is *not* a maximum flow.

On the other hand, suppose the residual graph G_f does *not* contain a directed path from s to t. Let S be the set of vertices that are reachable from s in G_f, and let $T = V \setminus S$. The partition (S, T) is clearly an (s, t)-cut. For every vertex $u \in S$ and $v \in T$, we have

$$c_f(u{\to}v) \;=\; (c(u{\to}v) - f(u{\to}v)) + f(v{\to}u) \;=\; 0.$$

The feasibility of f implies $c(u{\to}v) - f(u{\to}v) \ge 0$ and $f(v{\to}u) \ge 0$, so in fact we must have $c(u{\to}v) - f(u{\to}v) = 0$ and $f(v{\to}u) = 0$. In other words, our flow f saturates every edge from S to T and avoids every edge from T to S. Lemma 10.1 now implies that $|f| = \|S, T\|$, which means f is a maximum flow and (S, T) is a minimum cut.

This completes the proof! □

10.4 Ford and Fulkerson's augmenting-path algorithm

Ford and Fulkerson's proof of the Maxflow-Mincut Theorem immediately suggests an algorithm to compute maximum flows: Starting with the zero flow, repeatedly augment the flow along **any** path from s to t in the residual graph, until there is no such path.

This algorithm has an important but straightforward corollary:

Integrality Theorem. *If all capacities in a flow network are integers, then there is a maximum flow such that the flow through every edge is an integer.*

Proof: We argue by induction that after each iteration of the augmenting path algorithm, all flow values and all residual capacities are integers.

- Before the first iteration, all flow values are 0 (which is an integer), and all residual capacities are the original capacities, which are integers by definition.
- In each later iteration, the induction hypothesis implies that the capacity F of the augmenting path is an integer, so augmenting changes the flow on each edge, and therefore the residual capacity of each edge, by an integer.

In particular, each iteration of the augmenting path algorithm increases the value of the flow by a positive integer. It follows that the algorithm eventually halts and returns a maximum flow. □

If every edge capacity is an integer, then conservatively, the Ford-Fulkerson algorithm halts after at most $|f^*|$ iterations, where f^* is the actual maximum flow. In each iteration, we can build the residual graph G_f and perform a whatever-first-search to find an augmenting path in $O(E)$ time. Thus, in this setting, the algorithm runs in $\boldsymbol{O(E|f^*|)}$ time in the worst case.

Jack Edmonds and Richard Karp observed that this running time analysis is essentially tight. Consider the 4-node network in Figure 10.7, where X is some large integer. The maximum flow in this network is clearly $2X$. However, Ford-Fulkerson might alternate between pushing one unit of flow along the augmenting path $s \to u \to v \to t$ and then pushing one unit of flow along the augmenting path $s \to v \to u \to t$, leading to a running time of $\Theta(X) = \Omega(|f^*|)$.

Figure 10.7. Edmonds and Karp's bad example for the Ford-Fulkerson algorithm.

Ford and Fulkerson's algorithm is usually fast in practice, and it is always fast when the maximum flow value $|f^*|$ is small, but without further constraints on the augmenting paths, this is *not* an efficient algorithm in worst case. Edmonds and Karp's bad example network can be described using only $O(\log X)$ bits; thus, the running time of Ford-Fulkerson is actually *exponential* in the input size.

♥Irrational Capacities

But what if the capacities are *not* integers? If we multiply all the capacities by the same (positive) constant, the maximum flow increases everywhere by the same constant factor. It follows that if all the edge capacities are *rational*, then the Ford-Fulkerson algorithm eventually halts, although still in exponential time (in the number of bits used to described the input).

However, if we allow *irrational* capacities, the algorithm can actually loop forever, always finding smaller and smaller augmenting paths. Worse yet, this infinite sequence of augmentations may not even converge to the maximum flow, or even to a significant fraction of the maximum flow! The smallest network that exhibits this bad behavior was discovered by Uri Zwick in 1993.[2]

Consider the six-node network shown in Figure 10.8. Six of the nine edges have some large integer capacity X, two have capacity 1, and one has capacity $\phi = (\sqrt{5}-1)/2 \approx 0.618034$, chosen so that $1-\phi = \phi^2$. To prove that the Ford-Fulkerson algorithm can get stuck, we can watch the residual capacities of the three horizontal edges as the algorithm progresses. (The residual capacities of the other six edges will always be at least $X-3$.)

Figure 10.8. Uri Zwick's non-terminating flow example, and three augmenting paths.

Suppose the Ford-Fulkerson algorithm starts by choosing the central augmenting path, shown at the top of Figure 10.8. The three horizontal edges, in

[2]In 1962, Ford and Fulkerson described a more complex network, with 10 vertices and 48 edges, with the same bad behavior.

order from left to right, now have residual capacities 1, 0, and ϕ. Suppose inductively that the horizontal residual capacities are ϕ^{k-1}, 0, and ϕ^k for some non-negative integer k.

1. Augment along path B, adding ϕ^k to the flow; the residual capacities are now ϕ^{k+1}, ϕ^k, and 0.
2. Augment along path C, adding ϕ^k to the flow; the residual capacities are now ϕ^{k+1}, 0, and ϕ^k.
3. Augment along path B, adding ϕ^{k+1} to the flow; the residual capacities are now 0, ϕ^{k+1}, and ϕ^{k+2}.
4. Augment along path A, adding ϕ^{k+1} to the flow; the residual capacities are now ϕ^{k+1}, 0, and ϕ^{k+2}.

It follows by induction that after $4n+1$ augmentation steps, the horizontal edges have residual capacities ϕ^{2n-2}, 0, and ϕ^{2n-1}. As the number of augmentations grows to infinity, the value of the flow converges to

$$1 + 2\sum_{i=1}^{\infty} \phi^i = 1 + \frac{2}{1-\phi} = 4 + \sqrt{5} < 7,$$

even though the maximum flow value is clearly $2X + 1 \gg 7$.

Practically-minded readers might wonder why anyone should care about irrational capacities; after all, computers can't represent anything but (small) integers or (small dyadic) rationals exactly. Good question! The mathematician's answer is that the restriction to integer capacities is literally *artificial*; it's an *artifact* of digital computational hardware (or perhaps the otherwise irrelevant laws of physics), not an inherent feature of the abstract computational problem. But a more practical reason is that the behavior of the algorithm with irrational inputs tells us something about its worst-case behavior *in practice* with floating-point capacities—terrible! Even with very reasonable capacities, a careless implementation of Ford-Fulkerson could enter an infinite loop, simply because of round-off error, without ever coming close to the correct answer.

10.5 Combining and Decomposing Flows

Flows are normally defined as functions on the edges of a graph satisfying certain constraints at the vertices. However, flows have a second characterization that is more natural and useful in certain contexts.

Consider an arbitrary graph G with source vertex s and target vertex t. Fix any two (s,t)-flows f and g and any two real numbers α and β, and consider the function $h\colon E \to \mathbb{R}$ defined by setting

$$h(u{\to}v) := \alpha \cdot f(u{\to}v) + \beta \cdot g(u{\to}v)$$

for every edge $u \to v$; we can write this definition more simply as $h = \alpha f + \beta g$. Straightforward definition-chasing implies that h is also an (s,t)-flow with value $|h| = \alpha|f| + \beta|g|$. More generally, any linear combination of (s,t)-flows is also an (s,t)-flow.

It turns out that any (s,t)-flow can be written as a weighted sum of flows with a special structure. For any directed path P from s to t, we define a corresponding ***path flow*** as follows:

$$P(u \to v) = \begin{cases} 1 & \text{if } u \to v \in P, \\ -1 & \text{if } v \to u \in P, \\ 0 & \text{otherwise.} \end{cases}$$

Straightforward definition-chasing implies that the function $P : E \to \mathbb{R}$ is indeed an (s,t)-flow with value 1. I am deliberately overloading the variable P to mean both the path (a sequence of vertices and directed edges) and the unit flow along that path.

Similarly, for any directed cycle C, we define a corresponding ***cycle flow*** by setting

$$C(u \to v) = \begin{cases} 1 & \text{if } u \to v \in C, \\ -1 & \text{if } v \to u \in C, \\ 0 & \text{otherwise.} \end{cases}$$

Again, it is easy to verify that $C : E \to \mathbb{R}$ is an (s,t)-flow with value zero.

Our earlier argument implies that any linear combination of path flows and cycle flows is another flow; this weighted sum is called a ***flow decomposition***. Moreover, *every* non-negative flow has a flow decomposition with the following special structure.

Flow Decomposition Theorem. *Every non-negative (s,t)-flow f can be written as a positive linear combination of directed (s,t)-paths and directed cycles. Moreover, a directed edge $u \to v$ appears in at least one of these paths or cycles if and only if $f(u \to v) > 0$, and the total number of paths and cycles is at most the number of edges in the network.*

Proof: We prove the theorem by induction on the number of edges carrying non-zero flow, intuitively by running the Ford-Fulkerson algorithm backward. As long as at least one edge in the graph carries positive flow, we can find either an (s,t)-path or a directed cycle that carries flow. Subtracting as much flow as possible from that path or cycle empties at least one edge, so the Recursion Fairy can give us the rest of the decomposition.

To formalize this argument, we first consider the special case of ***circulations***; these are flows with value 0, where flow is conserved at *every* vertex. Fix an

Figure 10.9. Decomposing a circulation into weighted directed cycles.

arbitrary circulation f in an arbitrary flow network, and let $\#f$ denote the number of edges $u{\to}v$ such that $f(u{\to}v) > 0$. We prove that f can be decomposed into a positive linear combination of at most $\max\{0, \#f - 1\}$ cycles, by induction on $\#f$. There are three cases to consider:

- If $\#f = 0$, then f is vacuously a linear combination of zero cycles.
- Suppose $f(u{\to}v) > 0$ for a single directed cycle of edges $u{\to}v$. Then $\#f \geq 2$, and f is trivially a linear combination of one cycle.
- Otherwise, pick an arbitrary edge $u{\to}v$ with $f(u{\to}v) > 0$. Consider an arbitrary walk $v_0{\to}v_1{\to}v_2{\to}\cdots$ with $v_0 = u$ and $v_1 = v$, such that $f(v_{i-1}{\to}v_i) > 0$ for every index i. The conservation constraint implies that every vertex with incoming flow also has outgoing flow, so we can make this walk arbitrarily long; in particular, the walk must eventually visit some vertex more than once. Let k be the smallest index such that $v_j = v_k$ for some index $j < k$. The subwalk $v_j{\to}v_{j-1}{\to}\cdots{\to}v_k$ is a simple directed cycle C.

  Define $F := \min_{e \in C} f(e)$, and consider the function $f' := f - F \cdot C$, or more verbosely,

$$f'(u{\to}v) := \begin{cases} f(u{\to}v) - F & \text{if } u{\to}v \in C, \\ f(u{\to}v) & \text{otherwise.} \end{cases}$$

  Straightforward definition-chasing shows that f' is another feasible circulation in G. There is at least one edge $e \in C$ such that $f(e) = F$, and therefore $f'(e) = 0$, which implies $\#f' \leq \#f - 1$. Since fewer edges carry flow in f' than in f, the Recursion Fairy can decompose f' into at most $\#f' - 1 \leq \#f - 2$ cycles. Adding F units of flow around cycle C gives us a flow decomposition for f; more succinctly: $f = f' + F \cdot C$.

Now let f be an arbitrary (s,t)-flow in an arbitrary flow network, such that $|f| > 0$. Add an edge $t \to s$ to the network, and define a circulation f' by setting $f'(t \to s) = |f|$ and $f'(u \to v) = f(u \to v)$ for every original edge $u \to v$; observe that $\#f' = \#f + 1 \geq 2$. The previous argument implies that the circulation f' is a positive linear combination of at most $\#f' - 1$ directed cycles. Deleting the edge $t \to s$ gives us a decomposition of the original flow f into at most $\#f' - 1 = \#f$ paths and cycles. Specifically, cycles in f' that include $t \to s$ become (s,t)-paths in f, and cycles in f' that do not include $t \to s$ remain cycles in f. □

The proof of the Flow Decomposition Theorem implies stronger results in two interesting special cases.

- Any circulation can be decomposed into a weighted sum of cycles; no paths are necessary.
- Any ***acyclic*** (s,t)-flow can be decomposed into a weighted sum of (s,t)-paths; no cycles are necessary.

Moreover, by canceling flow cycles until no more remain, we can transform any flow into an acyclic flow with the same value. In particular, every flow network supports a maximum (s,t)-flow that is acyclic.

The proof also immediately translates directly into an algorithm, similar to Ford-Fulkerson, to decompose any (s,t)-flow into paths and cycles. The algorithm repeatedly seeks either a directed (s,t)-path or a directed cycle in the remaining flow, and then subtracts as much flow as possible along that path or cycle, until the flow is empty. We can find a flow path or cycle in $O(V)$ time as follows:

- If any edge leaving s has positive flow, follow an arbitrary walk from s in the flow graph until it either reaches t (giving us a flow path) or reaches some vertex for the second time (giving us a flow cycle).
- If no edge leaving s has positive flow, find any other vertex v with positive outflow, and follow an arbitrary walk from v in the flow graph until it reaches some vertex for the second time (giving us a flow cycle).

In both cases, the conservation constraint implies that this algorithm will never get stuck. Each iteration takes $O(V)$ time and removes at least one edge from the flow graph; thus, the entire decomposition algorithm runs in $\boldsymbol{O(VE)}$ ***time***.

Flow decompositions provide a natural lower bound on the running time of any maximum-flow algorithm that builds the flow one path or cycle at a time. Every flow can be decomposed into at most E paths and cycles, each of which uses at most V edges, so the overall complexity of the flow decomposition is $O(VE)$. Moreover, it is easy to construct flows for which *every* flow decomposition has complexity $\Omega(VE)$. Thus, any maximum-flow algorithm that explicitly constructs a flow one path or cycle at a time—in particular, any implementation

of Ford and Fulkerson's augmenting path algorithm—must take $\Omega(VE)$ time in the worst case.

10.6 Edmonds and Karp's Algorithms

Ford and Fulkerson's algorithm does not specify which path in the residual graph to augment; the poor worst-case behavior of the algorithm can be blamed on poor choices for the augmenting path. In the early 1970s, Jack Edmonds and Richard Karp published two natural rules for choosing augmenting paths, both of which led to more efficient algorithms.

Fattest Augmenting Paths

Edmonds and Karp's first rule is essentially a greedy algorithm:

Choose the augmenting path with largest bottleneck value.

It's not hard to show that the maximum-bottleneck (s,t)-path in a directed graph can be computed in $O(E \log V)$ time using a "best-first" traversal, similar to Jarník's minimum-spanning-tree algorithm or Dijkstra's shortest-path algorithm. The algorithm grows a directed tree T, rooted at s, one vertex at a time, by repeatedly adding the highest-capacity edge leaving T to T, until T contains a path from s to t. Alternately, one could emulate Kruskal's algorithm—insert edges one at a time in decreasing capacity order until there is a path from s to t—although this approach is less efficient, at least when the graph is directed.

To complete the running-time analysis of the flow algorithm, we need an upper bound on the number of iterations before the algorithm halts. In fact, for arbitrary real capacities, the algorithm may *never* halt; see Exercise 18. For integer capacities, however, we can bound the number of iterations as a function of the maximum flow value $|f^*|$, as follows.

Let f be any flow in G, and let f' be the maximum flow *in the current residual graph* G_f. (At the beginning of the algorithm, $G_f = G$ and $f' = f^*$.) We have already proved that f' can be decomposed into at most E paths and cycles. A simple averaging argument implies that at least one of the paths in this decomposition must carry at least $|f'|/E$ units of flow. It follows immediately that the *fattest* (s,t)-path in G_f carries at least $|f'|/E$ units of flow.

Thus, augmenting f along the maximum-bottleneck path in G_f multiplies the value of the remaining maximum flow in G_f by a factor of at most $1-1/E$. In other words, the residual maximum flow value *decays exponentially* with the number of iterations. After $E \cdot \ln|f^*|$ iterations, the maximum flow value in G_f is at most

$$|f^*| \cdot (1-1/E)^{E \cdot \ln|f^*|} < |f^*|\, e^{-\ln|f^*|} = 1.$$

(That's Euler's constant e, not the edge e. Sorry.) In particular, after $E \cdot \ln|f^*|$ iterations, the residual maximum flow value is less than 1. *If all capacities are integers*, the residual maximum flow value is also an integer, so it must be 0; in other words, f is a maximum flow!

We conclude that for graphs with integer capacities, the Edmonds-Karp "fattest path" algorithm runs in $\boldsymbol{O(E^2 \log E \log|f^*|)}$ time. Unlike the worst-case running time of raw Ford-Fulkerson, this time bound is actually a polynomial function of the input size.

Just like the original Ford-Fulkerson algorithm, the "fattest path" algorithm can get stuck in an infinite loop in networks with arbitrary real capacities. However, our analysis implies that even if the algorithm never halts, it maintains a flow f that approaches a maximum flow in the limit.

Shortest Augmenting Paths

The second Edmonds-Karp rule was actually proposed as a practical heuristic by Ford and Fulkerson in their original maximum-flow paper; a variant of this rule was independently proposed in 1970 by the Russian mathematician Yefim Dinitz.[3]

Choose the augmenting path with the smallest number of edges.

The shortest augmenting path can be found in $O(E)$ time by running breadth-first search in the residual graph. Surprisingly, the resulting algorithm halts after a polynomial number of iterations, independent of the actual edge capacities!

The proof of this polynomial upper bound relies on two observations about the evolution of the residual graph. Let f_i be the current flow after i augmentation steps, let G_i be the corresponding residual graph. In particular, f_0 is zero everywhere and $G_0 = G$. For each vertex v, let $\boldsymbol{level_i(v)}$ denote the unweighted shortest-path distance from s to v in G_i, or equivalently, the *level* of v in a breadth-first search tree of G_i rooted at s. In particular, if there is no path from s to v in G_i, then $level_i(v) = \infty$ (because $\min \varnothing = \infty$).

Our first observation is that the level of a vertex can only increase over time.

Lemma 10.2. *$level_i(v) \geq level_{i-1}(v)$ for all vertices v and all integers $i > 0$.*

Proof: Fix an arbitrary positive integer $i > 0$ and an arbitrary vertex v. We prove the claim by induction on $level_i(v)$ (and *not* on the integer i). As an inductive hypothesis, assume for every vertex u such that $level_i(u) < level_i(v)$, that $level_i(u) \geq level_{i-1}(u)$. There are three cases to consider.

[3] Specifically, Dinitz discovered a more complex maximum-flow algorithm, while he was a student in an algorithms class taught by Georgy Adelson-Velsky (the "AV" in AVL trees), in response to an in-class exercise. Dinitz's algorithm also pushes flows along shortest paths, but with additional bookkeeping to reduce the running time from $O(VE^2)$ to $O(V^2E)$.

- If $v = s$, we immediately have $level_i(s) = level_{i-1}(s) = 0$.
- If there is no path from s to v in G_i, then $level_i(v) = \infty \geq level_{i-1}(v)$.
- Otherwise, let $s \to \cdots \to u \to v$ be any unweighted shortest path from s to v in the graph G_i. Because this is a shortest path, we have $level_i(v) = level_i(u) + 1$, so the inductive hypothesis implies $level_i(u) \geq level_{i-1}(u)$. To complete the proof, we need to show that $level_{i-1}(u) \geq level_{i-1}(v) - 1$. We have two subcases to consider.
  - If $u \to v$ is an edge in G_{i-1}, then $level_{i-1}(v) \leq level_{i-1}(u) + 1$, because the levels are defined by breadth-first traversal.
  - On the other hand, if $u \to v$ is not an edge in G_{i-1}, then its reversal $v \to u$ must be an edge in the ith augmenting path, which by definition is the shortest path from s to t in G_{i-1}. It follows that $level_{i-1}(v) = level_{i-1}(u) - 1 \leq level_{i-1}(u) + 1$.

  In both subcases, we conclude that $level_i(v) = level_i(u) + 1 \geq level_{i-1}(u) + 1 \geq level_{i-1}(v)$. □

Whenever we augment the flow, the bottleneck edge in the augmenting path disappears from the residual graph, and some edges in the *reversal* of the augmenting path may (re-)appear. Our second observation is that an edge cannot appear or disappear too many times.

Lemma 10.3. *During the execution of the Edmonds-Karp shortest-augmenting-path algorithm, each edge $u \to v$ disappears from the residual graph G_f at most $V/2$ times.*

Proof: Suppose $u \to v$ is in two residual graphs G_i and G_{j+1}, but not in any of the intermediate residual graphs $G_{i+1}, \ldots, G_j$, for some $i < j$. Then $u \to v$ must be in the ith augmenting path, so $level_i(v) = level_i(u) + 1$, and $v \to u$ must be on the jth augmenting path, so $level_j(v) = level_j(u) - 1$. The previous lemma implies that

$$level_j(u) \;=\; level_j(v) + 1 \;\geq\; level_i(v) + 1 \;=\; level_i(u) + 2.$$

In other words, between the disappearance and reappearance of $u \to v$, the distance from s to u increased by at least 2. Because every level is either less than V or infinite, the number of disappearances is at most $V/2$. □

Now we can derive an upper bound on the number of iterations. Because each edge disappears at most $V/2$ times, there are at most $EV/2$ edge disappearances overall. But at least one edge disappears on each iteration, so the algorithm must halt after at most $EV/2$ iterations. Finally, each iteration requires $O(E)$ time, so the overall algorithm runs in $\boldsymbol{O(VE^2)}$ ***time***.

10.7 Further Progress

This is nowhere near the end of the story for maximum-flow algorithms. Decades of further research have led to several faster algorithms, some of which are summarized in Figure 10.10.[4] All the listed algorithms listed compute a maximum flow in several iterations. Most of these algorithms have two variants: a simpler version that performs each iteration by brute force, and a faster variant that uses sophisticated data structures to maintain a spanning tree of the flow network, so that each iteration can be performed (and the spanning tree updated) in logarithmic time. There is no reason to believe that the best algorithms known so far are optimal; indeed, maximum flows are still a very active area of research.

Technique	Direct	With dynamic trees	Source(s)
Blocking flow	$O(V^2E)$	$O(VE\log V)$	[Dinitz; Karzanov; Even and Itai; Sleator and Tarjan]
Network simplex	$O(V^2E)$	$O(VE\log V)$	[Dantzig; Goldfarb and Hao; Goldberg, Grigoriadis, and Tarjan]
Push-relabel (generic)	$O(V^2E)$	—	[Goldberg and Tarjan]
Push-relabel (FIFO)	$O(V^3)$	$O(VE\log(V^2/E))$	[Goldberg and Tarjan]
Push-relabel (highest label)	$O(V^2\sqrt{E})$	—	[Cheriyan and Maheshwari; Tunçel]
Push-relabel-add games	—	$O(VE\log_{E/(V\log V)} V)$	[Cheriyan and Hagerup; King, Rao, and Tarjan]
Pseudoflow	$O(V^2E)$	$O(VE\log V)$	[Hochbaum]
Pseudoflow (highest label)	$O(V^3)$	$O(VE\log(V^2/E))$	[Hochbaum and Orlin]
Incremental BFS	$O(V^2E)$	$O(VE\log(V^2/E))$	[Goldberg, Held, Kaplan, Tarjan, and Werneck]
Compact networks	—	$O(VE)$	[Orlin]

Figure 10.10. Several purely combinatorial maximum-flow algorithms and their running times.

The fastest known (purely combinatorial) maximum-flow algorithm, announced by James Orlin in 2012, runs in $\boldsymbol{O(VE)}$ ***time***, exactly matching the worst-case complexity of a flow decomposition. The details of Orlin's algorithm are far beyond the scope of this book; in addition to his own new techniques, Orlin uses several older algorithms and data structures as black boxes, most of which are themselves quite complicated. In particular, Orlin's algorithm does *not* construct an explicit flow decomposition; in fact, for graphs with only $O(V)$ edges, an extension of his algorithm actually runs in only $O(V^2/\log V)$ time! Nevertheless, for purposes of analyzing algorithms that *use* maximum flows,

[4]To keep this table short, I have deliberately omitted algorithms whose running time depends on edge capacities or the maximum flow value. Even with this restriction, the list is embarrassingly incomplete!

this is the time bound you should cite. So write the following sentence on your cheat sheets and cite it in your homeworks:

Maximum flows can be computed in $O(VE)$ time.

Finally, faster maximum-flow algorithms are known for *unit-capacity* networks, where every edge has capacity 1. In 1973, Alexander Karzanov proved that Dinitz's blocking-flow algorithm—the first algorithm listed in the table above—runs in $O(\min\{V^{2/3}, E^{1/2}\}E)$ time in this setting. (This time bound appears to break the $\Omega(VE)$ flow decomposition barrier, but in fact Karzanov's analysis implies that any flow in a unit-capacity network can be decomposed into paths with total complexity $O(\min\{V^{2/3}, E^{1/2}\}E)$.) This was the fastest algorithm known in this setting for four decades. Karzanov's record was finally broken in 2013, when Aleksander Mądry announced a truly remarkable algorithm that computes maximum flows in unit-capacity networks in $O(E^{10/7}\operatorname{polylog} E)$ time. Again, the details of Mądry's algorithm are far beyond the scope of this book, or indeed the expertise of its author.

Exercises

0. Suppose you are given a directed graph $G = (V, E)$, two vertices s and t, a capacity function $c : E \to \mathbb{R}^+$, and a second function $f : E \to \mathbb{R}$. Describe an algorithm to determine whether f is a maximum (s, t)-flow in G.

1. Let f and f' be two feasible (s, t)-flows in a flow network G, such that $|f'| > |f|$. Prove that there is a feasible (s, t)-flow with value $|f'| - |f|$ in the residual network G_f.

2. Let $u{\to}v$ be an arbitrary edge in an arbitrary flow network G. Prove that if there is a minimum (s, t)-cut (S, T) such that $u \in S$ and $v \in T$, then there is *no* minimum cut (S', T') such that $u \in T'$ and $v \in S'$.

3. Let (S, T) and (S', T') be minimum (s, t)-cuts in some flow network G. Prove that $(S \cap S', T \cup T')$ and $(S \cup S', T \cap T')$ are also minimum (s, t)-cuts in G.

4. Let G be a flow network that contains an opposing pair of edges $u{\to}v$ and $v{\to}u$, both with positive capacity. Let G' be the flow network obtained from G by decreasing the capacities of both of these edges by $\min\{c(u{\to}v), c(v{\to}u)\}$. In other words:

   - If $c(u{\to}v) > c(v{\to}u)$, change the capacity of $u{\to}v$ to $c(u{\to}v) - c(v{\to}u)$ and delete $v{\to}u$.

- If $c(u \to v) < c(v \to u)$, change the capacity of $v \to u$ to $c(v \to u) - c(u \to v)$ and delete $u \to v$.
- Finally, if $c(u \to v) = c(v \to u)$, delete both $u \to v$ and $v \to u$.

Figure 10.11. Enforcing the one-direction assumption.

(a) Prove that every maximum (s, t)-flow in G' is also a maximum (s, t)-flow in G. (Thus, by simplifying *every* opposing pair of edges in G, we obtain a new reduced flow network with the same maximum flow value as G.)

(b) Prove that every minimum (s, t)-cut in G is also a minimum (s, t)-cut in G' *and vice versa*.

(c) Prove that there is at least one maximum (s, t)-flow in G that is ***not*** a maximum (s, t)-flow in G'.

5. (a) Describe an efficient algorithm to determine whether a given flow network contains a *unique* maximum (s, t)-flow.

(b) Describe an efficient algorithm to determine whether a given flow network contains a *unique* minimum (s, t)-cut.

(c) Describe a flow network that contains a unique maximum (s, t)-flow but does not contain a unique minimum (s, t)-cut.

(d) Describe a flow network that contains a unique minimum (s, t)-cut but does not contain a unique maximum (s, t)-flow.

6. An (s, t)-flow in a network G is *acyclic* if there are no directed cycles where every edge has a positive flow value; that is, the subgraph of edges with positive flow value is a dag.

(a) Describe and analyze an algorithm to compute an *acyclic* maximum (s, t)-flow in a given flow network. Your algorithm should have the same asymptotic running time as Ford-Fulkerson.

(b) Describe and analyze an algorithm to determine whether *every* maximum (s, t)-flow in a given flow network is acyclic.

7. Let $G = (V, E)$ be a flow network in which every edge has capacity 1 and the shortest-path distance from s to t is at least d.

(a) Prove that the value of the maximum (s,t)-flow is at most E/d.

(b) Now suppose that G is *simple*, meaning that for all vertices u and v, there is at most one edge from u to v. (Flow networks can have parallel edges.) Prove that the value of the maximum (s,t)-flow is at most $O(V^2/d^2)$. *[Hint: How many nodes are in the average level of a BFS tree rooted at s?]*

8. Suppose we are given a flow network $G = (V,E)$ in which every edge has capacity 1, together with an integer k. Describe and analyze an algorithm to identify k edges in G such that after deleting those k edges, the value of the maximum (s,t)-flow in the remaining graph is as small as possible.

9. The analysis in our proof of the Flow Decomposition Theorem can be tightened. Let $G = (V,E)$ be an arbitrary flow network, and let f be an arbitrary (s,t)-flow in G.

(a) Prove that if $|f| = 0$, then f is the weighted sum of at most $E - V + 1$ directed cycles, where $f(e) > 0$ for every edge e in each of these cycles.

(b) Prove that if $|f| > 0$, then f is the weighted sum of at most $E - V + 2$ directed paths and directed cycles, where $f(e) > 0$ for every edge e in each of these paths and cycles.

(c) Prove that both of the previous upper bounds are tight: There are graphs in which some circulations cannot be decomposed into less than $E-V+1$ cycles, and some flows cannot be decomposed into less than $E - V + 2$ paths and cycles. *[Hint: This is easy.]*

♣10. Our observation that any linear combination of (s,t)-flows is itself an (s,t)-flow implies that the set of all (not necessarily feasible) (s,t)-flows in any graph actually define a real *vector space*, which we can call the ***flow space*** of the graph.

(a) Prove that the flow space of any connected graph $G = (V,E)$ has dimension $E - V + 2$.

(b) Let T be any spanning tree of G. Prove that the following collection of paths and cycles define a basis for the flow space:

- The unique path in T from s to t;
- The unique cycle in $T \cup \{e\}$, for every edge $e \notin T$.

(c) Let T be any spanning tree of G, and let F be the forest obtained by deleting any single edge in T. Prove that the following collection of paths and cycles define a basis for the flow space:

- The unique path in $F \cup \{e\}$ from s to t, for every edge $e \notin F$ that has one endpoint in each component of F;

- The unique cycle in $F \cup \{e\}$, for every edge $e \notin F$ with both endpoints in the same component of F.

(d) Prove or disprove the following claim: Every connected flow network has a flow basis that consists entirely of simple paths from s to t.

11. Cuts are sometimes defined as subsets of the edges of the graph, instead of as partitions of its vertices. In this problem, you will prove that these two definitions are *almost* equivalent.

We say that a subset X of (directed) edges *separates s and t* if every directed path from s to t contains at least one (directed) edge in X. For any subset S of *vertices*, let $\boldsymbol{\delta S}$ denote the set of directed edges leaving S; that is, $\delta S := \{u \to v \mid u \in S, v \notin S\}$.

(a) Prove that if (S, T) is an (s, t)-cut, then δS separates s and t.

(b) Let X be an arbitrary subset of edges that separates s and t. Prove that there is an (s, t)-cut (S, T) such that $\delta S \subseteq X$.

(c) Let X be a *minimal* subset of edges that separates s and t. (Such a set of edges is sometimes called a ***bond***.) Prove that there is an (s, t)-cut (S, T) such that $\delta S = X$.

12. Suppose instead of capacities, we consider networks where each edge $u \to v$ has a non-negative ***demand*** $d(u \to v)$. Now an (s, t)-flow f is *feasible* if and only if $f(u \to v) \geq d(u \to v)$ for every edge $u \to v$. (Feasible flow values can now be arbitrarily large.) A natural problem in this setting is to find a feasible (s, t)-flow of *minimum* value.

(a) Describe an efficient algorithm to compute a feasible (s, t)-flow, given the graph, the demand function, and the vertices s and t as input. *[Hint: Find a flow that is non-zero everywhere, and then scale it up to make it feasible.]*

(b) Suppose you have access to a subroutine MaxFlow that computes *maximum* flows in networks with edge capacities. Describe an efficient algorithm to compute a *minimum* flow in a given network with edge demands; your algorithm should call MaxFlow exactly once.

(c) State and prove an analogue of the max-flow min-cut theorem for this setting. (Do minimum flows correspond to maximum cuts?)

13. For any flow network G and any vertices u and v, let $bottleneck_G(u, v)$ denote the maximum, over all paths π in G from u to v, of the minimum-capacity edge along π.

(a) Describe and analyze an algorithm to compute $bottleneck_G(s,t)$ in $O(E \log V)$ time. This is the amount of flow that the Edmonds-Karp fattest-augmenting-paths algorithm pushes in the first iteration.

(b) Now suppose the flow network G is undirected; equivalently, suppose $c(u{\to}v) = c(v{\to}u)$ for every pair of vertices u and v. Describe and analyze an algorithm to compute $bottleneck_G(s,t)$ in $O(V+E)$ time. *[Hint: Find the median edge capacity.]* Why doesn't this speedup work for directed graphs?

♥(c) Again, suppose the flow network G is undirected. Describe and analyze an algorithm to construct a spanning tree T of G such that $bottleneck_T(u,v) = bottleneck_G(u,v)$ for *all* vertices u and v. (Edges in T inherit their capacities from G.) For full credit, your algorithm should run in $O(E)$ time.

14. Suppose you are given a flow network G with ***integer*** edge capacities and an ***integer*** maximum flow f^* in G. Describe algorithms for the following operations:

(a) Increment(e): Increase the capacity of edge e by 1 and update the maximum flow.

(b) Decrement(e): Decrease the capacity of edge e by 1 and update the maximum flow.

Both algorithms should modify f^* so that it is still a maximum flow, more quickly than recomputing a maximum flow from scratch.

15. Let G be a network with integer edge capacities. An edge in G is *upper-binding* if increasing its capacity by 1 also increases the value of the maximum flow in G. Similarly, an edge is *lower-binding* if decreasing its capacity by 1 also decreases the value of the maximum flow in G.

(a) Does every network G have at least one upper-binding edge? Prove your answer is correct.

(b) Does every network G have at least one lower-binding edge? Prove your answer is correct.

(c) Describe an algorithm to find all upper-binding edges in G, given both G and a maximum flow in G as input, in $O(E)$ time.

(d) Describe an algorithm to find all lower-binding edges in G, given both G and a maximum flow in G as input, in $O(EV)$ time.

16. A given flow network G may have more than one minimum (s,t)-cut. Let's define the ***best*** minimum (s,t)-cut to be any minimum cut (S,T) with the smallest number of edges crossing from S to T.

(a) Describe an efficient algorithm to find the best minimum (s, t)-cut when the capacities are integers.

(b) Describe an efficient algorithm to find the best minimum (s, t)-cut for *arbitrary* edge capacities.

(c) Describe an efficient algorithm to determine whether a given flow network contains a unique *best* minimum (s, t)-cut.

17. A new assistant professor, teaching maximum flows for the first time, suggests the following greedy modification to the generic Ford-Fulkerson augmenting path algorithm. Instead of maintaining a residual graph, just[5] reduce the capacity of edges along the augmenting path! In particular, whenever we saturate an edge, just remove it from the graph. Who needs all that residual graph nonsense?

```
GreedyFlow(G, c, s, t):
  for every edge e in G
      f(e) ← 0

  while there is a path from s to t
      π ← an arbitrary path from s to t
      F ← minimum capacity of any edge in π
      for every edge e in π
          f(e) ← f(e) + F
          if c(e) = F
              remove e from G
          else
              c(e) ← c(e) − F

  return f
```

(a) Show that GreedyFlow does not always compute a maximum flow.

(b) Show that GreedyFlow is not even guaranteed to compute a good approximation to the maximum flow. That is, for any constant $\alpha > 1$, there is a flow network G such that the value of the maximum flow is more than α times the value of the flow computed by GreedyFlow. *[Hint: Assume that GreedyFlow chooses the worst possible path π at each iteration.]*

18. In 1980 Maurice Queyranne published an example of a flow network, shown below, where Edmonds and Karp's "fattest path" heuristic does not halt. As in Zwick's bad example for the original Ford-Fulkerson algorithm, ϕ denotes the inverse golden ratio $(\sqrt{5}-1)/2$. The three vertical edges play essentially the same role as the horizontal edges in Zwick's example.

[5] The adverb *just* is almost always subconscious shorthand for "I'm too lazy to figure out the details, but you should believe me anyway", or more succinctly, "This is probably wrong." See also *merely*, *simply*, *clearly*, and *obviously*.

Figure 10.12. Queyranne's network, and a sequence of "fattest path" augmentations.

(a) Show that the following infinite sequence of path augmentations is a valid execution of the Edmonds-Karp "fattest path" algorithm. (See the bottom of Figure 10.12.)

```
QueyranneFatPaths:
  for i ← 1 to ∞
      push ϕ^(3i−2) units of flow along s→a→f→g→b→h→c→d→t
      push ϕ^(3i−1) units of flow along s→f→a→b→g→h→c→t
      push ϕ^(3i)   units of flow along s→e→f→a→g→b→c→h→t
```

(b) Describe a sequence of $O(1)$ path augmentations that yields a maximum flow in Queyranne's network.

♥19. An **(s,t)-*series-parallel*** graph is a directed acyclic graph with two distinguished vertices s and t and with one of the following structures:

- **Base case:** A single directed edge from s to t.
- **Series:** The union of an (s,u)-series-parallel graph and a (u,t)-series-parallel graph that share a common vertex u but no other vertices or edges.
- **Parallel:** The union of two smaller (s,t)-series-parallel graphs with the same source s and target t, but with no other vertices or edges in common.

Every (s,t)-series-parallel graph G can be represented by a ***decomposition tree***, which is a binary tree with three types of nodes: leaves (which corresponding to edges in G), series nodes (which correspond to vertices other than s and t), and parallel nodes. The same series-parallel graph could be represented by many different decomposition trees.

(a) Suppose you are given a directed graph G with two special vertices s and t. Describe and analyze an algorithm that either builds a decomposition tree for G or correctly reports that G is not (s,t)-series-parallel. *[Hint: Build the tree from the bottom up.]*

Figure 10.13. A series-parallel graph and a corresponding decomposition tree. Squares in the decomposition tree are leaves; diamonds are parallel nodes.

(b) Describe and analyze an algorithm to compute a maximum (s, t)-flow in a given (s, t)-series-parallel flow network with arbitrary edge capacities. *[Hint: In light of part (a), you can assume that you are actually given the decomposition tree. First compute the maximum-flow value, then compute an actual maximum flow.]*

20. We can speed up the Edmonds-Karp "fattest path" algorithm, at least for networks with small integer capacities, by relaxing our requirements for the next augmenting path. Instead of finding the augmenting path with maximum bottleneck capacity, we find a path whose bottleneck capacity is at least half of maximum, using the following ***capacity scaling*** algorithm. (This algorithm was actually proposed by Edmonds and Karp.)

Assume all the edge capacities are positive integers less than $U = 2^k$ for some integer k. The scaling algorithm maintains a bottleneck threshold Δ; initially, we set $\Delta \leftarrow U$. In each *phase*, the algorithm augments along paths from s to t in which every edge has residual capacity at least Δ. When there is no such path, the phase ends, we set $\Delta \leftarrow \lfloor \Delta/2 \rfloor$, and the next phase begins. The algorithm ends when $\Delta = 0$.

(a) How many phases will this algorithm execute in the worst case?

(b) Let f be the flow at the end of a phase for a particular value of Δ. Prove that the capacity of a minimum cut in the residual graph G_f is at most $E \cdot \Delta$.

(c) Prove that in each phase of the scaling algorithm, there are at most $2E$ augmentations.

(d) What is the overall running time of the capacity scaling algorithm?

For a long time it puzzled me how something so expensive, so leading edge, could be so useless, and then it occurred to me that a computer is a stupid machine with the ability to do incredibly smart things, while computer programmers are smart people with the ability to do incredibly stupid things. They are, in short, a perfect match.

— Bill Bryson, *Notes from a Big Country* (1999)

Shortly after the "iron curtain" fell in 1990, an American and a Russian, who had both worked on the development of weapons, met. The American asked: "When you developed the Bomb, how were you able to perform such an enormous amount of computing with your weak computers?" The Russian responded: "We used better algorithms."

— Yefim Dinitz, in "Dinitz' Algorithm: The Original Version and Even's Version" (2006)

So I'm the bad guy now I hear, because I don't go with the flow.
Don't ever go with the flow, be the flow.

— Jay-Z, "Stream of Consciousness", May 16, 2015

11

Applications of Flows and Cuts

11.1 Edge-Disjoint Paths

One of the easiest applications of maximum flows is computing the maximum number of edge-disjoint paths between two specified vertices s and t in a directed graph G using maximum flows. A set of paths in G is *edge-disjoint* if each edge in G appears in at most one of the paths; several edge-disjoint paths may pass through the same vertex, however.

If we give each edge capacity 1, then the maximum flow from s to t pushes either 0 or 1 units of flow along each edge. The flow-decomposition theorem implies that the subgraph S of saturated edges is the union of several edge-disjoint paths and cycles. Moreover, the number of paths in this decomposition is exactly equal to the value of the flow. Extracting the actual paths from S is straightforward—follow any directed path in S from s to t, remove that path from S, and recurse.

Conversely, we can transform any collection of k edge-disjoint paths into a

flow by pushing one unit of flow along each path from s to t; the value of the resulting flow is exactly k. It follows that any maximum flow algorithm actually computes the largest possible set of edge-disjoint paths.

If we use Orlin's algorithm to compute maximum flows, we can compute edge-disjoint paths in $O(VE)$ time, but Orlin's algorithm is overkill for this simple application. The cut $(\{s\}, V \setminus \{s\})$ has capacity at most $V-1$, so the maximum flow has value at most $V-1$. Thus, Ford and Fulkerson's original augmenting path algorithm already runs in $O(|f^*|E) = \boldsymbol{O(VE)}$ ***time***.

The same algorithm can also be used to find edge-disjoint paths in *undirected* graphs. First, replace every undirected edge uv in G with a pair of directed edges $u{\to}v$ and $v{\to}u$, each with unit capacity, and call the resulting directed graph G'. Next, compute a maximum (s,t)-flow f^* in G' using Ford-Fulkerson. For any edge uv in G, if f^* saturates both directed edges $u{\to}v$ and $v{\to}u$ in G', we can remove *both* edges from the flow without changing its value. (More generally, we can find an *acyclic* maximum flow in G' by canceling all cycles in f^*, not only cycles of length 2.) Thus, without loss of generality, f^* assigns a unique direction to each saturated edge. Finally, we can extract the edge-disjoint paths by searching the subgraph of directed edges saturated by f^*.

11.2 Vertex Capacities and Vertex-Disjoint Paths

Now suppose the vertices of the input graph G have capacities, not just the edges. In addition to our other constraints, for each vertex v other than s and t, we require the total flow into v (and therefore the total flow out of v) to be at most some non-negative value $c(v)$:

$$\sum_{u\to v} f(u{\to}v) \le c(v).$$

Can we still compute maximum flows with these new vertex constraints?

In 1962, Ford and Fulkerson proposed the following reduction to a flow network $\bar{G}$ with only edge capacities. Replace every vertex v with two vertices v_{in} and v_{out}, connected by an edge $v_{\text{in}}{\to}v_{\text{out}}$ with capacity $c(v)$, and then replace every directed edge $u{\to}v$ with the edge $u_{\text{out}}{\to}v_{\text{in}}$ (keeping the same capacity). Routine definition-chasing implies that every feasible $(s_{\text{out}}, t_{\text{in}})$-flow in $\bar{G}$ is equivalent to a feasible (s,t)-flow with the same value in the original graph G, and vice versa. In particular, every maximum flow in $\bar{G}$ is equivalent to a maximum flow in G. The reduction from G to $\bar{G}$ takes $O(E)$ time, after which we can compute the maximum flow in $\bar{G}$ using Orlin's algorithm. Altogether, computing the maximum flow in G requires $\boldsymbol{O(VE)}$ ***time***.

It is now easy to compute the maximum number of *vertex*-disjoint paths from s to t in $O(VE)$ time: Assign capacity 1 to every vertex and compute a maximum flow!

Figure 11.1. Reducing vertex-disjoint paths in G to edge-disjoint paths in $\bar{G}$.

11.3 Bipartite Matching

Another natural application of maximum flows is finding maximum ***matchings*** in bipartite graphs. A matching is a subgraph in which every vertex has degree at most one, or equivalently, a collection of edges such that no two share a vertex. The problem is to find a matching with the maximum number of edges.

For example, suppose we have a set of doctors who are looking for jobs, and a set of hospitals who are looking for doctors. Each doctor lists all hospitals where they are willing to work, and each hospital lists all doctors they are willing to hire. Our task is to find the largest subset of doctor-hospital hires that everyone is willing to accept.[1] This problem is equivalent to finding a maximum matching in a bipartite graph whose vertices are the doctors and hospitals, and there is an edge between a doctor and a hospital if and only if each find the other acceptable.

We can solve this problem by reducing it to a maximum flow problem, as follows. Let G be the given bipartite graph with vertex set $L \cup R$, such that every edge joins a vertex in L to a vertex in R. We create a new *directed* graph G' by (1) orienting each edge from L to R, (2) adding a new source vertex s with edges to every vertex in L, and (3) adding a new target vertex t with edges from every vertex in R. Finally, we assign every edge in G' a capacity of 1.

Any matching M in G can be transformed into a flow f_M in G' as follows: For each edge uw in M, push one unit of flow along the path $s \to u \to w \to t$. These paths are disjoint except at s and t, so the resulting flow satisfies the capacity constraints. Moreover, the value of the resulting flow is equal to the number of edges in M.

Conversely, consider any (s,t)-flow f in G', computed using the Ford-Fulkerson augmenting path algorithm. Because the edge capacities are integers, the Ford-Fulkerson algorithm assigns an integer flow to every edge. (This is easy to verify by induction, hint, hint.) Moreover, since each edge has *unit* capacity, the computed flow either saturates ($f(e) = 1$) or avoids ($f(e) = 0$) every edge

[1]This problem is very different from the stable matching problem we saw in Chapter 4, because we aren't trying to make each doctor and hospital as happy as possible.

in G'. Finally, since at most one unit of flow can enter any vertex in U or leave any vertex in W, the saturated edges from U to W form a matching in G. The size of this matching is exactly $|f|$.

Thus, the size of the maximum matching in G is equal to the value of the maximum flow in G', and provided we compute the maxflow using augmenting paths, we can convert the actual maxflow into a maximum matching in $O(E)$ time. We can compute the maximum flow in $\boldsymbol{O(VE)}$ ***time*** using either Orlin's algorithm or off-the-shelf Ford-Fulkerson.

Figure 11.2. A maximum matching in a bipartite graph G, and the corresponding maximum flow in G'.

It is enlightening to interpret the behavior of Ford-Fulkerson in G' in terms of the original bipartite graph G. The algorithm maintains a matching M in G, which is initially empty; the edges of M correspond to edges in G' that carry flow. Call a vertex of G *matched* if is an endpoint of some edge in M, and *unmatched* otherwise. In each iteration of the algorithm, we look for an *alternating path* in G—a path from an unmatched vertex of L to an unmatched vertex in R that alternates between edges in M and edges not in M. (Alternating paths in G correspond exactly to augmenting paths for the current flow in G'.) If we find an augmenting path P, we update M to the symmetric difference $M \oplus P$, which increases the number of edges in M by 1, and continue to the next iteration. If there is no alternating path, the maxflow-mincut theorem implies that M is a maximum matching, so the algorithm ends. Finding a single alternating path requires $O(E)$ time, and the algorithm halts after at most V iterations, so the overall algorithm runs in $\boldsymbol{O(VE)}$ ***time***. Figure 11.3 shows this algorithm in action.

This characterization of maximum bipartite matchings in terms of alternating paths was proved by Claude Berge in 1957 (independently of the maxflow-mincut theorem), although it was already implicit in algorithms described by Harald Kuhn in 1955, by Dénes Kőnig in 1916, and by Carl Jacobi around 1836.

A more sophisticated algorithm, proposed by John Hopcroft and Richard Karp in 1973, computes maximum matchings in bipartite graphs in only $O(\sqrt{V}E)$ time, by finding several disjoint alternating paths in each iteration.

Figure 11.3. An increasing sequence of matchings connected by alternating paths.

11.4 Tuple Selection

The bipartite maximum matching problem is the simplest example of a broader class of problems that I call ***tuple selection***.[2] The input to a tuple selection problem consists of several finite sets $X_1, X_2, \ldots, X_d$, each representing a different discrete resource. Our task is to select the largest possible set of d-tuples, each containing exactly one element from each set X_i, subject to several *capacity* constraints of the following form:

- For each index i, Each element $x \in X_i$ can appear in at most $c(x)$ selected tuples.
- For each index i, any two elements $x \in X_i$ and $y \in X_{i+1}$ can appear in at most $c(x, y)$ selected tuples.

Each of the upper bounds $c(x)$ and $c(x, y)$ is either a (typically small) non-negative integer or ∞.

In the maximum-matching problem, we have $d = 2$ resources, each element x has capacity $c(x) = 1$, and each pair (x, y) has capacity $c(x, y) = 1$ or $c(x, y) = 0$, depending on whether or not xy is an edge in the underlying bipartite graph.

[2] I couldn't find a standard name for these problems, so I made up my own. These are sometimes called "assignment problems", but it's more common for the phrase "the assignment problem" to refer to the problem of finding a maximum-*weight* bipartite matching in an edge-weighted bipartite graph.

Because the resources are linearly ordered, and only pairs of objects in *adjacent* subsets X_i and X_{i+1} are constrained,[3] the tuple selection problem can be reduced to a maximum-flow problem in a directed graph G defined as follows:

- G contains a vertex for each element of each set X_i, as well as a source vertex s and a target vertex t. Each vertex x (except s and t) has capacity $c(x)$.
- G contains an edge $s \to w$ for each element $w \in X_1$, an edge $z \to t$ for each element $z \in X_d$, and an edge $x \to y$ with capacity $c(x, y)$ for each pair of elements $x \in X_i$ and $y \in X_{i+1}$, for all i. (Optionally, we can omit edges $x \to y$ with $c(x, y) = 0$.)

Every path from s to t in G corresponds to (or "is") a d-tuple that we *could* select; conversely, every selectable d-tuple that satisfies the stated constraints corresponds to (or "is") a path from s to t in G.

Figure 11.4. The flow network for a tuple selection problem.

More generally, let f be an arbitrary feasible integer (s, t)-flow in G. Because all capacities are integers or ∞, the Flow Decomposition Theorem implies that f is the sum of $|f|$ paths from s to t, each carrying exactly one unit of flow. Straightforward definition-chasing implies that the resulting set of tuples satisfies all the capacity constraints. Conversely, for any set of k tuples that satisfies the capacity constraints, the sum of the k corresponding paths is a feasible integer (s, t)-flow with value k.

Thus, we can select the maximum number of tuples that satisfy the given capacity constraints by computing a maximum (s, t)-flow f^* in G and then computing a flow decomposition of f^*. Because all finite capacities in G are integers, we can assume without loss of generality that f^* is an integer flow, and therefore (by the previous paragraph) corresponds to a valid set of $|f^*|$ tuples.

Exam Scheduling

The following "real world" scheduling problem might help clarify our general reduction.

[3] If pairs of objects from even one non-adjacent pair of subsets (X_i and X_j where $j > i + 1$) are also constrained, the problem becomes NP-hard, by a straightforward reduction from Exact-3DimensionalMatching. We'll discuss NP-hardness in the next chapter.

Sham-Poobanana University has hired you to write an algorithm to schedule their final exams. There are n different classes, each of which needs to schedule a final exam in one of r rooms during one of t different time slots. At most one class's final exam can be scheduled in each room during each time slot; conversely, classes cannot be split into multiple rooms or multiple times. Moreover, each exam must be overseen by one of p proctors.[4] Each proctor can oversee at most one exam at a time; each proctor is available for only certain time slots; and no proctor is allowed oversee more than 5 exams total. The input to the scheduling problem consists of three arrays:

- An integer array $E[1..n]$ where $E[i]$ is the number of students enrolled in the ith class.
- An integer array $S[1..r]$, where $S[j]$ is the number of seats in the jth room. The ith class's final exam *can* be held in the jth room if and only if $E[i] \leq S[j]$.
- A boolean array $A[1..t, 1..p]$ where $A[k, \ell] = \textsc{True}$ if and only if the ℓth proctor is available during the kth time slot.[5]

let $N = n + r + tp$ denote the total size of the input. Your job is to design an algorithm that either schedules a room, a time slot, and a proctor for every class's final exam, or correctly reports that no such schedule is possible.

This is a standard tuple-selection problem with four resources: classes, rooms, time slots, and proctors. To solve this problem, we construct a flow network G with six types of vertices—a source vertex s', a vertex c_i for each class, a vertex r_j for each room, a vertex t_k for each time slot, a vertex p_ℓ for each proctor, and a target vertex t'—and five types of edges, as shown in Figure 11.5:

Figure 11.5. A flow network for the exam scheduling problem.

- An edge $s' \to c_i$ with capacity 1 for each class i. ("Each class can hold at most one final exam.")

[4] or as they are better known outside the US, invigilators

[5] Arguably, this information is better represented as a graph, but I thought that would make the reduction more confusing.

- An edge $c_i \to r_j$ with capacity ∞ for each class i and room j such that $E[i] \le S[j]$. ("Class i can hold exams in room j if and only if the room has enough seats.") This is the only place where the enrollments $E[i]$ and seat numbers $S[j]$ are used.
- An edge $r_j \to t_k$ with capacity 1 for each room j and time slot k. ("At most one exam can be held in room j at time k.")
- An edge $t_k \to p_\ell$ with capacity 1 for time slot k and proctor ℓ such that $A[\ell, k] = \text{True}$. ("A proctor can oversee at most one exam at any time, and only during times that they are available.")
- An edge $p_\ell \to t'$ with capacity 5 for each proctor ℓ. ("Each proctor can oversee at most 5 exams.")

(I'm calling the source and target vertices s' and t' instead of s and t only because the problem statement already uses the variable t to denote the number of time slots.) Altogether, G has $n + r + t + p + 2 = O(N)$ vertices and $O(nr + rt + tp) = O(N^2)$ edges.

Each path from s' to t' in G represents a unique valid choice of class, room, time, and proctor for one final exam; specifically, the class fits into the room, and the proctor is available at that time. Conversely, for each valid choice (class, room, time, proctor), there is a corresponding path from s' to t' in G. Thus, we can construct a valid schedule for the maximum possible number of exams by computing an maximum (s', t')-flow f^* in G, decomposing f^* into paths from s' to t', and then transcribing each path into a class-room-time-proctor assignment. If $|f^*| = n$, we can return the resulting schedule; otherwise, we can correctly report that scheduling all n final exams is impossible.

Constructing G from the given input data by brute force takes $O(E)$ time. We can compute the maximum flow in $O(VE)$ time using either Ford-Fulkerson (because $|f^*| \le n < V$) or Orlin's algorithm, and we can compute the flow decomposition in $O(VE)$ time. Thus, the overall algorithm runs in $O(VE) =$ ***$O(N^3)$ time***.

11.5 Disjoint-Path Covers

A ***path cover*** of a directed graph G is a collection of directed paths in G such that every vertex of G lies on *at least* one path. A ***disjoint-***path cover of G is a path cover such that every vertex of G lies on *exactly* one path. Every directed graph has a trivial disjoint-path cover consisting of several paths of length zero, but that's boring. Instead, let's look for disjoint-path covers that contain as few paths as possible. This problem is NP-hard in general—a graph has a disjoint-path cover of size 1 if and only if it contains a Hamiltonian path—but there is an efficient flow-based algorithm for directed *acyclic* graphs.

To solve this problem for a given directed acyclic graph $G = (V, E)$, we construct a new bipartite graph $G' = (V', E')$ as follows.

- H contains two vertices $v^\flat$ and $v^\sharp$ for every vertex v of G.
- H contains an undirected edge $u^\flat v^\sharp$ for every directed edge $u \to v$ in G.

(If G is represented as an adjacency matrix, then G' is the bipartite graph represented by the same adjacency matrix!)

Figure 11.6. Reducing minimum disjoint-path cover of a dag to maximum bipartite matching; squares are flat$^\flat$ and diamonds are sharp$^\sharp$.

Now I claim that G can be covered by k disjoint paths if and only if the new graph G' has a matching of size $V - k$. As usual, we prove the equivalence in two stages:

$\Leftarrow$ Suppose G has a disjoint path cover P with k paths; think of P as a subgraph of G. Every vertex in P has in-degree either 0 or 1; moreover, there is exactly one vertex with in-degree 0 in each path in P. It follows that P has exactly $V - k$ edges. Now define a subset M of the edges of G' as follows:

$$M := \left\{ u^\flat v^\sharp \in E' \;\middle|\; u \to v \in P \right\}.$$

By definition of disjoint-path cover, every vertex of G has at most one incoming edge in P and at most one outgoing edge in P. We conclude that every vertex of G' is incident to at most one edge in M; that is, M is a matching of size $V - k$.

$\Rightarrow$ Suppose G' has a matching M of size $V - k$. We project M' back to G by defining a subgraph $P = (V, M')$, where

$$M' := \left\{ u \to v \in E \;\middle|\; u^\flat v^\sharp \in M \right\}.$$

By definition of matching, every vertex of G has at most one incoming edge in P and at most one outgoing edge in P. It follows that P is a collection of disjoint directed paths in G; since P includes every vertex, P defines an disjoint path cover with $V - k$ edges. The number of paths in P is equal to the number of vertices in G that have no incoming edge in M'. We conclude that P contains exactly k paths.

It follows immediately that we can find a minimum disjoint-path cover in G by computing a maximum matching in G', using Ford-Fulkerson's maximum-flow algorithm, in $O(V'E') = \boldsymbol{O(VE)}$ ***time***.

Despite its formulation in terms of dags and paths, this is really a maximum matching problem: We want to *match* as many vertices as possible to distinct *successors* in the graph. The number of paths required to cover the dag is equal to the number of vertices with no successor. (And of course, every bipartite maximum matching problem is really a flow problem.)

Minimal Faculty Hiring

Let's go back to Sham-Poobanana University for another "real-world" scheduling problem.[6] SPU offers several thousand courses every day. Due to extreme budget cuts, the university needs to significantly reduce the size of its faculty. However, because students pay tuition (and the university cannot afford lawyers), the university must retain enough professors to guarantee that every class advertised in the course catalog is actually taught. How few professors can SPU get away with? Each remaining faculty member will be assigned a sequence of classes to teach on any given day. The classes assigned to each professor must not overlap; moreover, there must be enough slack in each professor's schedule for them to walk from one class to the next. For purposes of this problem, let's assume that every professor is capable of teaching every class, and that professors will not have office hours, lunches, or bathroom breaks.[7]

Concretely, suppose there are n classes offered in m different locations. The input to our problem consists of the following data:

- An array $C[1..n]$ of classes, where each class $C[i]$ has three fields: the starting time $C[i].start$, the ending time $C[i].end$, and the location $C[i].loc$.
- A two-dimensional array $T[1..m, 1..m]$, where $T[u, v]$ is the time required to walk from location u to location v.

We want to find the minimum number of professors that can collectively teach every class, such that whenever a professor is assigned to teach two classes i and j where $C[j].start \geq C[i].start$, we actually have

$$C[j].start \;\geq\; C[i].end + T\big[C[i].loc, C[j].loc\big].$$

We can solve this problem by reducing it to a disjoint-path cover problem as follows. We construct a dag $G = (V, E)$ whose vertices are classes and whose edges represent pairs of classes that are scheduled far enough apart to be taught

[6]For a somewhat more realistic (and less depressing) formulation of this problem, consider airplanes and flights, or buses and bus routes, instead of professors and classes.

[7]They will, however, be expected to answer student emails as they walk between classes.

by the same professor. Specifically, a directed edge $i \to j$ indicates that the same professor can teach class i and then class j. It is easy to construct this dag in $O(n^2)$ time by brute force. Then we find a disjoint-path cover of G using the matching algorithm described above; each directed path in G represents a legal class schedule for one professor. The entire algorithm runs in $O(n^2 + VE) = \boldsymbol{O(n^3)}$ ***time***.[8]

Despite its initial description in terms of intervals and distances, this is really a maximum matching problem (which means it's *really* really a maximum-flow problem). Specifically, we want to *match* as many classes as possible to the *next* class taught by the same professor. The number of professors we need is equal to the number of classes with no assigned successor; each class without an assigned successor is the last class that some professor teaches.

11.6 Baseball Elimination

Every year millions of American baseball fans eagerly watch their favorite team, hoping they will win a spot in the playoffs, and ultimately the World Series. Sadly, most teams are "mathematically eliminated" days or even weeks before the regular season ends. Often, it is easy to spot when a team is eliminated—they can't win enough games to catch up to the current leader in their division. But sometimes the situation is more subtle. For example, here are the actual standings from the American League East on August 30, 1996.

Team	**Won–Lost**	**Left**	NYY	BAL	BOS	TOR	DET
New York Yankees	75–59	28		3	8	7	3
Baltimore Orioles	71–63	28	3		2	7	4
Boston Red Sox	69–66	27	8	2		0	0
Toronto Blue Jays	63–72	27	7	7	0		0
Detroit Tigers	49–86	27	3	4	0	0	

Detroit is clearly behind, but some die-hard Tigers fans may hold out hope that their team can still win. After all, if Detroit wins all 27 of their remaining games, they will end the season with 76 wins, more than any other team has now. So as long as every other team loses every game. . . but that's not possible,

[8] If we assume that every time interval $T[u, v]$ is equal,[9] this scheduling problem can actually be solved in $O(n \log n)$ time using a simple greedy algorithm.

[9] Many American universities schedule ten-minute breaks between classes, under the remarkable belief that a human being can walk from any classroom to any other classroom on the same campus in ten minutes. I invite anyone who thinks this belief is realistic to visit my campus and walk from one Siebel Center to the other.

because some of those other teams still have to play each other. Here is one complete argument:[10]

> By winning all of their remaining games, Detroit can finish the season with a record of 76 and 86. If the Yankees win just 2 more games, then they will finish the season with a 77 and 85 record which would put them ahead of Detroit. So, let's suppose the Tigers go undefeated for the rest of the season and the Yankees fail to win another game.
>
> The problem with this scenario is that New York still has 8 games left with Boston. If the Red Sox win all of these games, they will end the season with at least 77 wins putting them ahead of the Tigers. Thus, the only way for Detroit to even have a chance of finishing in first place, is for New York to win exactly one of the 8 games with Boston and lose all their other games. Meanwhile, the Sox must lose all the games they play against teams other than New York. This puts them in a 3-way tie for first place. . . .
>
> Now let's look at what happens to the Orioles and Blue Jays in our scenario. Baltimore has 2 games left with with Boston and 3 with New York. So, if everything happens as described above, the Orioles will finish with at least 76 wins. So, Detroit can catch Baltimore only if the Orioles lose all their games to teams other than New York and Boston. In particular, this means that Baltimore must lose all 7 of its remaining games with Toronto. The Blue Jays also have 7 games left with the Yankees and we have already seen that for Detroit to finish in first place, Toronto must will all of these games. But if that happens, the Blue Jays will win at least 14 more games giving them at final record of 77 and 85 or better which means they will finish ahead of the Tigers. So, no matter what happens from this point in the season on, Detroit can not finish in first place in the American League East.

There has to be a better way to figure this out!

Here is a more abstract formulation of the problem. Our input consists of two arrays $W[1..n]$ and $G[1..n, 1..n]$, where $W[i]$ is the number of games team i has already won, and $G[i, j]$ is the number of upcoming games between teams i and j. We want to determine whether team n can end the season with the most wins (possibly tied with other teams).[11]

In the mid-1960s, Benjamin Schwartz observed that this question can be modeled as a maximum flow problem; about 20 years later, Dan Gusfield, Charles Martel, and David Fernández-Baca simplified Schwartz's flow formulation to a pair selection problem. Specifically, we want to know whether it is possible to ***select*** *a winner for each game*, so that team n comes in first place. Let $R[i] = \sum_j G[i, j]$ denote the number of remaining games for team i. We will assume that team n wins all $R[n]$ of its remaining games. Then team n can come in first place if and only if every other team i wins at most $W[n] + R[n] - W[i]$ of its $R[i]$ remaining games.

Since we want to ***select*** a winning team for each game, we start by building a bipartite graph, whose nodes represent the games and the teams. We have

[10] Both the example and this argument are taken from Eli Olinick's web site https://s2.smu.edu/~olinick/riot/detroit.html, which is based on Olinick's joint research with Ilan Adler, Alan Erera, and Dorit Hochbaum.

[11] We are implicitly assuming that no game ends in a tie and that every game is actually played. Both assumptions are consistent with Major League Baseball rules, at least for games that affect postseason standing, barring wars, natural disasters, or swarms of bees.

$\binom{n}{2}$ *game* nodes $g_{i,j}$, one for each pair $1 \le i < j < n$, and $n-1$ *team* nodes t_i, one for each $1 \le i < n$. For each pair i, j, we add edges $g_{i,j} \to t_i$ and $g_{i,j} \to t_j$ with *infinite* capacity. We add a source vertex s and edges $s \to g_{i,j}$ with capacity $G[i,j]$ for each pair i, j. Finally, we add a target node t and edges $t_i \to t$ with capacity $W[n] - W[i] + R[n]$ for each team i.

Figure 11.7 shows the graph derived from the 1996 American League East standings, where "team n" is the Detroit Tigers. All unlabeled edges have infinite capacity.

Figure 11.7. Cubs win! Cubs win!

Theorem. *Team n can end the season in first place if and only if there is a feasible flow in this graph that saturates every edge leaving s.*

Proof: Suppose it is possible for team n to end the season in first place. Then every team $i < n$ wins at most $W[n] + R[n] - W[i]$ of the remaining games. For each game between team i and team j that team i wins, add one unit of flow along the path $s \to g_{i,j} \to t_i \to t$. Because there are exactly $G[i,j]$ games between teams i and j, every edge leaving s is saturated. Because each team i wins at most $W[n] + R[n] - W[i]$ games, the resulting flow is feasible.

Conversely, let f be a feasible flow that saturates every edge out of s. Suppose team i wins exactly $f(g_{i,j} \to t_i)$ games against team j, for all i and j. Then teams i and j play $f(g_{i,j} \to t_i) + f(g_{i,j} \to t_j) = f(s \to g_{i,j}) = G[i,j]$ games, so every upcoming game is played. Moreover, each team i wins a total of $\sum_j f(g_{i,j} \to t_i) = f(t_i \to t) \le W[n] + R[n] - W[i]$ upcoming games, and therefore at most $W[n] + R[n]$ games overall. Thus, if team n win all their upcoming games, they end the season in first place. □

In summary, to decide whether our favorite team can win, we construct the flow network, compute a maximum flow, and report whether than maximum flow saturates every edge out of s. For example, in the graph in Figure 11.7, the total capacity of the edges leaving s is 27 (because there are 27 remaining games). On the other hand, the total capacity of the edges entering t is only 26,

which implies that the maximum flow value is at most 26. We conclude that Detroit is mathematically eliminated.[12]

The flow network has $O(n^2)$ vertices and $O(n^2)$ edges, and it can be constructed in $O(n^2)$ time. Using Orlin's algorithm, we can compute the maximum flow in $O(VE) = \boldsymbol{O(n^4)}$ ***time***.

This is not the fastest algorithm for the baseball elimination problem. In 2001, Kevin Wayne proved that one can determine ***all*** teams that are mathematically eliminated in only $\boldsymbol{O(n^3)}$ ***time***, essentially using a single maximum-flow computation.

11.7 Project Selection

In our final example, suppose we are given a set of n projects that we could possibly perform. Some projects cannot be started until certain other projects are completed. The projects and their dependencies are described by a directed acyclic graph G whose vertices are the projects, where each edge $u \to v$ indicates that project u cannot be performed before project v. (This is exactly the form of dependency graphs we considered in Chapter 6.4.) Finally, each project v has an associated *profit* $\$(v)$ which will be given to us if the project is completed; some projects have negative profits, which we interpret as positive *costs*. We can choose to finish any subset X of the projects that includes all its dependents; that is, for every project $x \in X$, every project that x depends on is also in X. Our goal is to find a valid subset of the projects whose total profit is as large as possible. In particular, if all of the jobs have negative profit, the correct answer is to do nothing.

Figure 11.8. A dependency graph for a set of eight projects. Diamonds indicate profitable projects; squares indicate costly projects. Each edge $u \to v$ means u depends on v.

At a high level, our task to partition the projects into two subsets S and T, the jobs we *Select* and the jobs we *Turn down*. So intuitively, we'd like to model our problem as a minimum cut problem in a certain graph. But in which graph? How do we enforce prerequisites? We want to *maximize* profit, but we only know how to find *minimum* cuts. And how do we convert negative profits into positive capacities?

[12]We got (un)lucky here; it is possible for a team to be eliminated even if the total capacity of all edges into t is no smaller than the total capacity of edges out of s.

To transform our given constraint graph G into a flow network G', we add a source vertex s and a target vertex t to the dependency graph, with an edge $s \to v$ for every profitable job v (with $\$(v) > 0$), and an edge $u \to t$ for every costly job u (with $\$(u) < 0$). Intuitively, we can think of s as a new job ("Sleep!") with profit/cost 0 that we must perform last. We assign capacities to the edges of G' as follows:

- $c(s \to v) = \$(v)$ for every profitable job v;
- $c(u \to t) = -\$(u)$ for every costly job u;
- $c(u \to v) = \infty$ for every dependency edge $u \to v$.

All edge-capacities are positive, so this is a valid input to the maximum cut problem.

Now consider an arbitrary (s,t)-cut (S,T) in G'. For any edge $u \to v$ in the original dependency graph, if $u \in S$ and $v \in T$, then $\|S,T\| = \infty$. Thus, we can legally select the jobs in S if and only if the capacity of the cut (S,T) is finite.

Figure 11.9. The flow network for the example dependency graph, along with its minimum cut. The cut has capacity 13 and $P = 15$, so the total profit for the selected jobs is 2.

In fact, it turns out that cuts with smaller capacity correspond to job selections with higher profit. Specifically, I claim that selecting the jobs in S earns a total profit of $P - \|S,T\|$, where P is the sum of all the positive profits:

$$P = \sum_{v} \max\{0, \$(v)\} = \sum_{\$(v)>0} \$(v).$$

We can prove this claim by straightforward definition-chasing, as follows. For any subset X of projects, we define three values. (Here, as usual, we define $c(u \to v) = 0$ when $u \to v$ is not an edge.)

$$cost(X) := \sum_{\substack{u \in X \\ \$(u)<0}} -\$(u) = \sum_{u \in X} c(u \to t)$$

$$yield(X) := \sum_{\substack{v \in X \\ \$(v)>0}} \$(v) = \sum_{v \in X} c(s \to v)$$

$$profit(X) := \sum_{v \in X} \$(v) = yield(X) - cost(X).$$

By definition, $P = yield(V) = yield(S) + yield(T)$. Because the cut (S, T) has finite capacity, only edges of the form $s \to v$ and $u \to t$ can cross the cut. By construction, every edge $s \to v$ points to a profitable job and each edge $u \to t$ points from a costly job. Thus, $\|S, T\| = cost(S) + yield(T)$. We immediately conclude that $P - \|S, T\| = yield(S) - cost(S) = profit(S)$, as claimed.

It follows immediately that we can *maximize* our total profit by computing a *minimum* cut in G'. We can easily construct G' from G in $O(V + E)$ time, and we can compute the minimum (s, t)-cut in G' in $O(VE)$ time using Orlin's algorithm. We conclude that the entire project-selection algorithm runs in $\boldsymbol{O(VE)}$ ***time***.

Exercises

1. Let $G = (V, E)$ be a directed graph where for each vertex v, the in-degree and out-degree of v are equal. Suppose G contains k edge-disjoint paths from some vertex u to another vertex v. Under these conditions, must G also contain k edge-disjoint paths from v to u? Give a proof or a counterexample with explanation.

2. Given an undirected graph $G = (V, E)$, with three vertices u, v, and w, describe and analyze an algorithm to determine whether there is a path from u to w that passes through v. *[Hint: If G were a directed graph, this problem would be NP-hard!]*

3. Consider a directed graph $G = (V, E)$ with several source vertices $s_1, s_2, \ldots, s_\sigma$ and target vertices $t_1, t_1, \ldots, t_\tau$, where no vertex is both a source and a target. A *multi-terminal flow* is a function $f : E \to \mathbb{R}_{\geq 0}$ that satisfies the flow conservation constraint at every vertex that is neither a source nor a target. The value $|f|$ of a multi-terminal flow is the total excess flow out of *all* the source vertices:

$$|f| := \sum_{i=1}^{\sigma} \left(\sum_{w} f(s_i \to w) - \sum_{u} f(u \to s_i) \right)$$

As usual, we are interested in finding flows with maximum value, subject to capacity constraints on the edges. (In particular, we don't care how much flow moves from any particular source to any particular target.)

(a) Consider the following algorithm for computing multi-terminal flows. The variables f and f' represent flow functions. The subroutine MaxFlow(G, s, t) solves the standard maximum flow problem with source s and target t.

```
MaxMultiFlow(G, s[1..σ], t[1..τ]):
  f ← 0                                  ⟨⟨Initialize the flow⟩⟩
  for i ← 1 to σ
      for j ← 1 to τ
          f′ ← MaxFlow(G_f, s[i], t[j])
          f ← f + f′                     ⟨⟨Update the flow⟩⟩
  return f
```

Prove that this algorithm correctly computes a maximum multi-terminal flow in G.

(b) Describe a more efficient algorithm to compute a maximum multi-terminal flow in G.

4. The Island of Sodor is home to a large number of towns and villages, connected by an extensive rail network. Recently, several cases of a deadly contagious disease (either swine flu or zombies; reports are unclear) have been reported in the village of Skarloey. The controller of the Sodor railway plans to close down certain railway stations to prevent the disease from spreading to Tidmouth, his home town. No trains can pass through a closed station. To minimize expense (and public notice), he wants to close down as few stations as possible. However, he cannot close the Skarloey station, because that would expose him to the disease, and he cannot close the Tidmouth station, because then he couldn't visit his favorite pub.

   Describe and analyze an algorithm to find the minimum number of stations that must be closed to block all rail travel from Skarloey to Tidmouth. The Sodor rail network is represented by an undirected graph, with a vertex for each station and an edge for each rail connection between two stations. Two special vertices s and t represent the stations in Skarloey and Tidmouth.

   For example, given the following input graph, your algorithm should return the integer 2.

5. An $n \times n$ grid is an undirected graph with n^2 vertices organized into n rows and n columns. We denote the vertex in the ith row and the jth column by (i, j). Every vertex (i, j) has exactly four neighbors $(i-1, j)$, $(i+1, j)$, $(i, j-1)$, and $(i, j+1)$, except the *boundary* vertices, for which $i = 1$, $i = n$, $j = 1$, or $j = n$.

   Let $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ be distinct vertices, called *terminals*, in the $n \times n$ grid. The ***escape problem*** is to determine whether there are m

vertex-disjoint paths in the grid that connect the terminals to any m distinct boundary vertices.

Figure 11.10. A positive instance of the escape problem, and its solution.

(a) Describe and analyze an efficient algorithm to solve the escape problem. The running time of your algorithm should be a small polynomial function of n.

(b) Now suppose the input to the escape problem consists of a single integer n and the list of m terminal vertices. If m is very small, the previous running time is actually *exponential* in the input size! Describe and analyze an algorithm to solve the escape problem in time *polynomial in* m.

♥(c) Modify the previous algorithm to output an explicit description of the escape paths (if they exist), still in time polynomial in m.

6. The SPU Commuter Silence Department is installing a mini-golf course in the basement of the See-Bull Center! The playing field is a closed polygon bounded by m horizontal and vertical line segments, meeting at right angles. The course has n *starting points* and n *holes*, in one-to-one correspondence. It is always possible hit the ball along a straight line directly from each starting point to the corresponding hole, without touching the boundary of the playing field. (Players are not allowed to bounce golf balls off the walls; too much glass.) The n starting points and n holes are all at distinct locations.

Sadly, the architect's computer crashed just as construction was about to begin. Thanks to the herculean efforts of their sysadmins, they were able to recover the *locations* of the starting points and the holes, but all information about which starting points correspond to which holes was lost!

Describe and analyze an algorithm to compute a one-to-one correspondence between the starting points and the holes that meets the straight-line requirement, or to report that no such correspondence exists. The input consists of the x- and y-coordinates of the m corners of the playing field, the n starting points, and the n holes. Assume you can determine in constant time whether two line segments intersect, given the x- and y-coordinates of their endpoints.

Figure 11.11. A mini-golf course with five starting points (⋆) and holes (∘), and a legal correspondence between them.

7. A ***cycle cover*** of a given directed graph $G = (V, E)$ is a set of vertex-disjoint cycles that cover every vertex in G. Describe and analyze an efficient algorithm to find a cycle cover for a given graph, or correctly report that no cycle cover exists. *[Hint: Use bipartite matching!]*

8. Suppose you are given an $n \times n$ checkerboard with some of the squares deleted. You have a large set of dominos, just the right size to cover two squares of the checkerboard. Describe and analyze an algorithm to determine whether one tile the board with dominos—each domino must cover exactly two undeleted squares, and each undeleted square must be covered by exactly one domino.

   Your input is a boolean array $Deleted[1..n, 1..n]$, where $Deleted[i, j] =$ TRUE if and only if the square in row i and column j has been deleted. Your output is a single boolean; you do ***not*** have to compute the actual placement of dominos. For example, for the board shown in Figure 11.12, your algorithm should return TRUE.

Figure 11.12. Covering a partial checkerboard with dominos.

9. Suppose we are given an $n \times n$ square grid, some of whose squares are colored black and the rest white. Describe and analyze an algorithm to determine whether tokens can be placed on the grid so that

   - every token is on a white square;
   - every row of the grid contains exactly one token; and
   - every column of the grid contains exactly one token.

Your input is a two dimensional array $IsWhite[1..n, 1..n]$ of booleans, indicating which squares are white. Your output is a single boolean. For example, given the grid in Figure 11.13 as input, your algorithm should return True.

Figure 11.13. Marking every row and column in a grid.

10. Suppose we are given a set of boxes, each specified by their height, width, and depth in centimeters. All three side lengths of every box lie strictly between 10cm and 20cm. As you should expect, one box can be placed inside another if the first box can be rotated so that its height, width, and depth are respectively smaller than the height, width, and depth of the second box. Boxes can be nested recursively. Call a box is *visible* if it is not inside another box.

    Describe and analyze an algorithm to nest the boxes so that the number of visible boxes is as small as possible.

11. Suppose we are given an $n \times n$ grid, some of whose cells are marked; the grid is represented by an array $M[1..n, 1..n]$ of booleans, where $M[i, j] = \text{True}$ if and only if cell (i, j) is marked. A *monotone* path through the grid starts at the top-left cell, moves only right or down at each step, and ends at the bottom-right cell. Our goal is to cover the marked cells with as few monotone paths as possible.

Figure 11.14. Greedily covering the marked cells in a grid with four monotone paths.

(a) Describe an algorithm to find a monotone path that covers the largest number of marked cells.

(b) There is a natural greedy heuristic to find a small cover by monotone paths: If there are any marked cells, find a monotone path π that covers the largest number of marked cells, unmark any cells covered by π those marked cells, and recurse. Show that this algorithm does *not* always compute an optimal solution.

(c) Describe and analyze an efficient algorithm to compute the smallest set of monotone paths that covers every marked cell.

12. The Faculty Senate at Sham-Poobanana University has decided to convene a committee to determine whether Uncle Gabby, Professor Bobo Cornelius, or Mofo the Psychic Gorilla should replace the recently disgraced Baron Factotum as the new official ~~mascot~~ *symbol* of SPU's athletic teams (The Fighting Pooh-bahs). Exactly one faculty member must be chosen from each academic department to serve on this committee. Some faculty members have appointments in multiple departments, but each committee member can represent only one department. For example, if Prof. Blagojevich is affiliated with both the Department of Corruption and the Department of Stupidity, and he is chosen as the Stupidity representative, then someone else must represent Corruption. Finally, University policy requires that every faculty committee must contain exactly the same number of assistant professors, associate professors, and full professors. Fortunately, the number of departments is a multiple of 3.

    Describe and analyze an algorithm to choose a subset of the SPU faculty to staff The Post-Factotum Simian ~~Mascot~~ *Symbol* Committee, or correctly report that no valid committee is possible. Your input is a bipartite graph indicating which professors belong to which departments; each professor vertex is labeled with that professor's rank (assistant, associate, or full).

13. The Department of Commuter Silence at Sham-Poobanana University has a flexible curriculum with a complex set of graduation requirements. The department offers n different courses, and there are m different requirements. Each requirement specifies a subset of the n courses and the number of courses that must be taken from that subset. The subsets for different requirements may overlap, but each course can be used to satisfy *at most one* requirement.

    For example, suppose there are $n = 5$ courses A, B, C, D, E and $m = 2$ graduation requirements:

    - You must take at least 2 courses from the subset $\{A, B, C\}$.
    - You must take at least 2 courses from the subset $\{C, D, E\}$.

    Then a student who has only taken courses B, C, D cannot graduate, but a student who has taken either A, B, C, D or B, C, D, E can graduate.

Describe and analyze an algorithm to determine whether a given student can graduate. The input to your algorithm is the list of m requirements (each specifying a subset of the n courses and the number of courses that must be taken from that subset) and the list of courses the student has taken.

14. You're organizing the First Annual SPU Commuter Silence 72-Hour Dance Exchange, to be held all day Friday, Saturday, and Sunday. Several 30-minute sets of music will be played during the event, and a large number of DJs have applied to perform. You need to hire DJs according to the following constraints.

    - Exactly k sets of music must be played each day, and thus $3k$ sets altogether.
    - Each set must be played by a single DJ in a consistent music genre (ambient, bubblegum, dubstep, horrorcore, K-pop, Kwaito, mariachi, straight-ahead jazz, trip-hop, Nashville country, parapara, ska, . . .).
    - Each genre must be played at most once per day.
    - Each candidate DJ has given you a list of genres they are willing to play.
    - Each DJ can play at most three sets during the entire event.

    Suppose there are n candidate DJs and g different musical genres available. Describe and analyze an efficient algorithm that either assigns a DJ and a genre to each of the $3k$ sets, or correctly reports that no such assignment is possible.

15. Suppose you are running a web site that is visited by the same set of people every day. Each visitor claims membership in one or more *demographic groups*; for example, a visitor might describe himself as male, 40–50 years old, a father, a resident of Illinois, an academic, a blogger, and a fan of Gilbert and Sullivan.[13] Your site is supported by advertisers. Each advertiser has told you which demographic groups should see its ads and how many of its ads you must show each day. Altogether, there are n visitors, k demographic groups, and m advertisers.

    Describe an efficient algorithm to determine, given all the data described in the previous paragraph, whether you can show each visitor exactly *one* ad per day, so that every advertiser has its desired number of ads displayed, and every ad is seen by someone in an appropriate demographic group.

16. Suppose we are given an array $A[1..m][1..n]$ of non-negative real numbers. We want to *round* A to an integer matrix, by replacing each entry x in A

[13] I am a very good theoretical computer scientist, specifically, a geometric algorithm specialist.

with either $\lfloor x \rfloor$ or $\lceil x \rceil$, without changing the sum of entries in any row or column of A. For example:

$$\begin{bmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{bmatrix} \longmapsto \begin{bmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{bmatrix}$$

(a) Describe and analyze an efficient algorithm that either rounds A in this fashion, or reports correctly that no such rounding is possible.

(b) Prove that a legal rounding is possible *if and only if* the sum of entries in each row is an integer, and the sum of entries in each column is an integer. In other words, prove that either your algorithm from part (a) returns a legal rounding, or a legal rounding is *obviously* impossible.

♥(c) Suppose we are guaranteed that none of the entries in the input matrix A is an integer. Describe and analyze an even faster algorithm that either rounds A or reports correctly that no such rounding is possible. For full credit, your algorithm must run in $O(mn)$ time. *[Hint: **Don't** use flows.]*

17. ***Ad-hoc networks*** are made up of low-powered wireless devices. In principle[14], these networks can be used on battlefields, in regions that have recently suffered from natural disasters, and in other hard-to-reach areas. The idea is that a large collection of cheap, simple devices could be distributed through the area of interest (for example, by dropping them from an airplane); the devices would then automatically configure themselves into a functioning wireless network.

These devices can communicate only within a limited range. We assume all the devices are identical; there is a distance D such that two devices can communicate if and only if the distance between them is at most D.

We would like our ad-hoc network to be reliable, but because the devices are cheap and low-powered, they frequently fail. If a device detects that it is likely to fail, it should transmit its information to some other *backup* device within its communication range. We require each device x to have k potential backup devices, all within distance D of x; we call these k devices the ***backup set*** of x. Also, we do not want any device to be in the backup set of too many other devices; otherwise, a single failure might affect a large fraction of the network.

So suppose we are given the communication radius D, parameters b and k, and an array $d[1..n, 1..n]$ of distances, where $d[i, j]$ is the distance between device i and device j. Describe an algorithm that either computes

[14] but not so much in practice

a backup set of size k for each of the n devices, such that no device appears in more than b backup sets, or reports (correctly) that no good collection of backup sets exists.

18. Faced with the threat of brutally severe budget cuts, Potemkin University has decided to hire actors to sit in classes as "students", to ensure that every class they offer is completely full. Because actors are expensive, the university wants to hire as few of them as possible.

    Building on their previous leadership experience at the now-defunct Sham-Poobanana University, the administrators at Potemkin have given you a directed acyclic graph $G = (V, E)$, whose vertices represent classes, and where each edge $i \to j$ indicates that the same "student" can attend class i and then later attend class j. In addition, you are also given an array $cap[1..V]$ listing the maximum number of "students" who can take each class. Describe an analyze an algorithm to compute the minimum number of "students" that would allow every class to be filled to capacity.

19. Quentin, Alice, and the other Brakebills Physical Kids are planning an excursion through the Neitherlands to Fillory. The Neitherlands is a vast, deserted city composed of several plazas, each containing a single fountain that can magically transport people to a different world. Adjacent plazas are connected by gates, which have been cursed by the Beast. The gates between plazas are open only for five minutes every hour, all simultaneously—from 12:00 to 12:05, then from 1:00 to 1:05, and so on—and are otherwise locked. During those five minutes, if more than one person passes through any single gate, the Beast will detect their presence.[15] Moreover, anyone attempting to open a locked gate, or attempting to pass through more than one gate within the same five-minute period will turn into a niffin.[16] However, any number of people can safely pass through *different* gates at the same time and/or pass through the same gate at *different* times.

    You are given a map of the Neitherlands, which is a graph G with a vertex for each fountain and an edge for each gate, with the fountains to Earth and Fillory clearly marked.

    (a) Suppose you are also given a positive integer h. Describe and analyze an algorithm to compute the maximum number of people that can walk from the Earth fountain to the Fillory fountain in at most h hours—that is, after the gates have opened at most h times—without anyone alerting the Beast or turning into a niffin. The running time of your algorithm should depend on h. *[Hint: Build a different graph.]*

[15] This is very bad.

[16] This is very very bad.

♣♥(b) Describe an analyze an algorithm for part (a) whose running time is polynomial in V and E, with *no* dependence on h.

(c) On the other hand, suppose you are also given an integer k. Describe and analyze an algorithm to compute the minimum number of hours that allow k people to walk from the Earth fountain to the Fillory fountain, without anyone alerting the Beast or turning into a niffin. *[Hint: Use part (a).]*

♥20. Let $G = (L \sqcup R, E)$ be a bipartite graph, whose left vertices L are indexed $\ell_1, \ell_2, \ldots, \ell_n$ and whose right vertices are indexed $r_1, r_2, \ldots, r_n$. A matching M in G is ***non-crossing*** if, for every pair of edges $\ell_i r_j$ and $\ell_{i'} r_{j'}$ in M, we have $i < i'$ if and only if $j < j'$.

(a) Describe and analyze an algorithm to find the largest non-crossing matching in G. *[Hint: This is not really a flow problem.]*

(b) Describe and analyze an algorithm to find the smallest number of non-crossing matchings $M_1, M_2, \ldots, M_k$ such that each edge in G is in exactly one matching M_i. *[Hint: This is really a flow problem.]*

♥21. Let $G = (L \sqcup R, E)$ be a bipartite graph, whose left vertices L are indexed $\ell_1, \ell_2, \ldots, \ell_n$ in some arbitrary order.

(a) A matching M in G is ***dense*** if there are no consecutive unmatched vertices in L; that is, for each index i, at least one of the vertices ℓ_i and ℓ_{i+1} is incident to an edge in M. Describe an algorithm to determine whether G has a dense matching.

(b) A matching M in G is ***sparse*** if there are no consecutive *matched* vertices in L; that is, for each index i, at least one of the vertices ℓ_i and ℓ_{i+1} is *not* incident to an edge in M. (In particular, the empty matching is sparse.) Describe an algorithm to find the largest sparse matching in G.

(c) A matching M in G is ***palindromic*** if, for every index i, either ℓ_i and ℓ_{n-i+1} are both incident to edges in M, or neither ℓ_i nor ℓ_{n-i+1} is incident to an edge in M. (In particular, the empty matching is palindromic.) Describe an algorithm to find the largest palindromic matching in G.

None of these problems restrict which vertices in R are matched or unmatched.

♥22. A *rooted tree* is a directed acyclic graph, in which every vertex has exactly one incoming edge, except for the *root*, which has no incoming edges. Equivalently, a rooted tree consists of a root vertex, which has edges pointing to the roots of zero or more smaller rooted trees. Describe an efficient algorithm to compute, given two rooted trees A and B, the largest rooted

tree that is isomorphic to both a subgraph of A and a subgraph of B. More briefly, describe an algorithm to find the largest common subtree of two rooted trees.

[Hint: This would be a relatively straightforward dynamic programming problem if either every node had $O(1)$ children or the children of each node were ordered from left to right. But for unordered trees with large degree, you need another technique to combine recursive subproblems efficiently.]

[I]n his short and broken treatise he provides an eternal example—not of laws, or even of method, for there is no method except to be very intelligent, but of intelligence itself swiftly operating the analysis of sensation to the point of principle and definition.

— T. S. Eliot on Aristotle, "The Perfect Critic", *The Sacred Wood* (1921)

The nice thing about standards is that you have so many to choose from; furthermore, if you do not like any of them, you can just wait for next year's model.

— Andrew S. Tannenbaum, *Computer Networks* (1981)

It is a rare mind indeed that can render the hitherto non-existent blindingly obvious. The cry "I could have thought of that" is a very popular and misleading one, for the fact is that they didn't, and a very significant and revealing fact it is too.

— Dirk Gently to Richard McDuff
in Douglas Adams' *Dirk Gently's Holistic Detective Agency* (1987)

If a problem has no solution, it may not be a problem, but a fact — not to be solved, but to be coped with over time.

— Shimon Peres, as quoted by David Rumsfeld, *Rumsfeld's Rules* (2001)

12

NP-Hardness

12.1 A Game You Can't Win

Imagine that a salesman in a red suit, who looks suspiciously like Tom Waits, presents you with a black steel box with n binary switches on the front and a light bulb on top. The salesman tells you that the state of the light bulb is controlled by a complex *boolean circuit*—a collection of AND, OR, and NOT gates connected by wires, with one input wire for each switch and a single output wire for the bulb. He then asks you a simple question: Is it possible to set the switches so that the light bulb turns on? If you can answer this question correctly, he will give you ~~one million~~ one hundred billion dollars; if you answer incorrectly, or if you die without answering, he will take your soul.

Figure 12.1. An AND gate, an OR gate, and a NOT gate.

Figure 12.2. A boolean circuit. Inputs enter from the left, and the output leaves to the right.

As far as you can tell, the Adversary hasn't connected the switches to the light bulb at all, so no matter how you set the switches, the light bulb will stay off. If you declare that it *is* possible to turn on the light, the Adversary will open the box and reveal that there is no circuit at all. But if you declare that it is *not* possible to turn on the light, before testing all 2^n settings, the Adversary will magically create a circuit inside the box that turns on the light *if and only if* the switches are in one of the settings you haven't tested, and then flip the switches to that setting, turning on the light. (You can't detect the Adversary's cheating, because you can't see inside the box until the end.) The only way to *provably* answer the Adversary's question correctly is to try all 2^n possible settings. You quickly realize that this will take *far* longer than you expect to live, so you gracefully decline the Adversary's offer.

The Adversary smiles and says, in a growl like Heath Ledger's Joker after smoking a carton of Marlboros, "Ah, yes, of course, you have no reason to trust me. But perhaps I can set your mind at ease." He hands you a large roll of parchment—which you hope was made from *sheep* skin—with a circuit diagram drawn (or perhaps tattooed) on it. "Here are the complete plans for the circuit inside the box. Feel free to poke around inside the box to make sure the plans are correct. Or build your own circuit from these plans. Or write a computer program to simulate the circuit. Whatever you like. If you discover that the plans don't match the actual circuit in the box, you win the hundred billion bucks." A few spot checks convince you that the plans have no obvious flaws; subtle cheating appears to be impossible.

But you should still decline the Adversary's "generous" offer. The problem that the Adversary is posing is called ***circuit satisfiability*** or ***CircuitSat***: Given a boolean circuit, is there a set of inputs that makes the circuit output True, or conversely, whether the circuit *always* outputs False. For any *particular* input setting, we can calculate the output of the circuit in polynomial (actually, *linear*) time using depth-first-search. But nobody knows how to solve CircuitSat faster than trying all 2^n possible inputs to the circuit by brute force, which requires exponential time. Admittedly, nobody has actually formally *proved* that we can't

beat brute force—maybe, just *maybe*, there's a clever algorithm that just hasn't been discovered yet—but nobody has actually formally proved that anti-gravity unicorns don't exist, either. For all practical purposes, it's safe to assume that there is no fast algorithm for CircuitSat.

You tell the salesman no. He smiles and says, "You're smarter than you look, kid," and then flies away on his anti-gravity unicorn.

12.2 P versus NP

A minimal requirement for an algorithm to be considered "efficient" is that its running time is bounded by a polynomial function of the input size: $O(n^c)$ for some constant c, where n is the size of the input.[1] Researchers recognized early on that not all problems can be solved this quickly, but had a hard time figuring out exactly which ones could and which ones couldn't. There are several so-called ***NP-hard*** problems, which most people believe *cannot* be solved in polynomial time, even though nobody can prove a super-polynomial lower bound.

A *decision problem* is a problem whose output is a single boolean value: Yes or No. Let me define three classes of decision problems:

- ***P*** is the set of decision problems that can be solved in polynomial time. Intuitively, P is the set of problems that can be solved quickly.
- ***NP*** is the set of decision problems with the following property: If the answer is Yes, then there is a *proof* of this fact that can be checked in polynomial time. Intuitively, NP is the set of decision problems where we can verify a Yes answer quickly if we have the solution in front of us.
- ***co-NP*** is essentially the opposite of NP. If the answer to a problem in co-NP is No, then there is a proof of this fact that can be checked in polynomial time.

For example, the circuit satisfiability problem is in NP. If a given boolean circuit is satisfiable, then any set of m input values that produces True output is a proof that the circuit is satisfiable; we can check the proof by evaluating the circuit in polynomial time. It is widely believed that circuit satisfiability is *not* in P or in co-NP, but nobody actually knows.

Every decision problem in P is also in NP. If a problem is in P, we can verify Yes answers in polynomial time recomputing the answer from scratch! Similarly, every problem in P is also in co-NP.

[1] This notion of efficiency was independently formalized by Alan Cobham in 1965, Jack Edmonds in 1965, and Michael Rabin in 1966, although similar notions were considered more than a decade earlier by Kurt Gödel, John Nash, and John von Neumann.

Perhaps the single most important unanswered question in theoretical computer science—if not all of computer science—if not all of ***science***—is whether the complexity classes P and NP are actually different. Intuitively, it seems obvious to most people that $P \neq NP$; the homeworks and exams in your algorithms and data structures classes have (I hope) convinced you that problems can be incredibly hard to solve, even when the solutions are simple in retrospect. It's completely obvious; *of course* solving problems from scratch is harder than verifying that a given solution is correct. We can reasonably accept—and most algorithm designers *do* accept—the statement "$P \neq NP$" as a law of nature, similar to other laws of nature like Maxwell's equations, general relativity, and the sun rising tomorrow morning that are strongly supported by evidence, but have no mathematical proof.

But if we're being mathematically rigorous, we have to admit that nobody knows how to *prove* that that $P \neq NP$. In fact, there has been little or no real progress toward a proof for decades.[2] The Clay Mathematics Institute lists P-versus-NP as the first of its seven Millennium Prize Problems, offering a $1,000,000 reward for its solution. And yes, in fact, several people *have* lost their souls, or at least their sanity, attempting to solve this problem.

A more subtle but still open question is whether the complexity classes NP and co-NP are different. Even if we can verify every Yes answer quickly, there's no reason to believe we can also verify No answers quickly. For example, as far as we know, there is no short proof that a boolean circuit is *not* satisfiable. It is generally believed that NP $\neq$ co-NP, but again, nobody knows how to prove it.

Figure 12.3. What we *think* the world looks like.

12.3 NP-hard, NP-easy, and NP-complete

A problem Π is ***NP-hard*** if a polynomial-time algorithm for Π would imply a polynomial-time algorithm for *every problem in NP*. In other words:

$$\boxed{\textbf{Π is NP-hard} \iff \textbf{If Π can be solved in polynomial time, then P=NP}}$$

[2]Perhaps the most significant progress has taken the form of *barrier* results, which imply that entire avenues of attack are doomed to fail. In a very real sense, not only do we have no idea how to prove $P \neq NP$, but we can actually *prove* that we have no idea how to prove $P \neq NP$!

Intuitively, if we could solve one particular NP-hard problem quickly, then we could quickly solve *any* problem whose solution is easy to understand, using the solution to that one special problem as a subroutine. NP-hard problems are at least as hard as every problem in NP.

Finally, a problem is ***NP-complete*** if it is both NP-hard and an element of NP (or "NP-easy"). Informally, NP-complete problems are the hardest problems in NP. A polynomial-time algorithm for even one NP-complete problem would immediately imply a polynomial-time algorithm for *every* NP-complete problem. Literally *thousands* of problems have been shown to be NP-complete, so a polynomial-time algorithm for one (and therefore all) of them seems incredibly unlikely.

Calling a problem NP-hard is like saying "If I own a dog, then it can speak fluent English." You probably don't know whether or not I own a dog, but I bet you're pretty sure that I don't own a *talking* dog. Nobody has a mathematical *proof* that dogs can't speak English—the fact that no one has ever heard a dog speak English is *evidence*, as are the hundreds of examinations of dogs that lacked the proper mouth shape and brainpower, but mere evidence is not a mathematical *proof*. Nevertheless, no sane person would believe me if I said I owned a dog that spoke fluent English.[3] So the statement "If I own a dog, then it can speak fluent English" has a natural corollary: No one in their right mind should believe that I own a dog! Similarly, if a problem is NP-hard, no one in their right mind should believe it can be solved in polynomial time.

Figure 12.4. More of what we *think* the world looks like.

It is not immediately obvious that *any* problems are NP-hard. The following remarkable theorem was first published by Stephen Cook in 1971 and independently by Leonid Levin in 1973.[4]

[3]. . . The department chair shakes his head sadly and says, "Oh, come on, that just sounds like barking. Let *me* ask a question. Who was the greatest complexity theorist of all time?" The dog cocks his head, pauses for a few seconds, and then says "Karp!" After the chair chases them out of his office, the dog turns to its owner and says, "Maybe I should have said Impagliazzo?"

[4]Levin first reported his results at seminars in Moscow in 1971, while still a PhD student. News of Cook's result did not reach the Soviet Union until at least 1973, after Levin's announcement of his results had been published; in accordance with Stigler's Law, this result is often called "Cook's Theorem". Levin was denied his PhD at Moscow University for political reasons; he emigrated to the US in 1978 and earned a PhD at MIT a year later. Cook was denied tenure by the Berkeley

The Cook-Levin Theorem. *Circuit satisfiability is NP-hard.*

I won't even sketch a proof here, because I've been (deliberately) vague about the definitions.[5]

♥12.4 Formal Definitions (*HC SVNT DRACONES*)

Formally, the complexity classes P, NP, and co-NP are defined in terms of *languages* and *Turing machines*. A language is a set of strings over some finite alphabet Σ; without loss of generality, we can assume that $\Sigma = \{0, 1\}$. A Turing machine is a very restrictive type of computer—crudely, a finite-state machine with an unbounded memory tape—whose precise definition is surprisingly unimportant. P is the set of languages that can be decided in ***P***olynomial time by a deterministic single-tape Turing machine. Similarly, NP is the set of all languages that can be decided in polynomial time by a nondeterministic Turing machine; NP is an abbreviation for ***N***ondeterministic ***P***olynomial-time.

The requirement of polynomial time is sufficiently crude that we do not have to specify the precise form of Turing machine (number of tapes, number of heads, number of tracks, size of the tape alphabet, and so on). In fact, any algorithm that runs on a random-access machine[6] in $T(n)$ time can be simulated by a single-tape, single-track, single-head Turing machine that runs in $O(T(n)^4)$ time. This simulation result allows us to argue formally about computational complexity in terms of standard high-level programming constructs like arrays and loops and recursion, instead of describing everything directly in terms of Turing machines.

Formally, a problem Π is NP-hard if and only if, for every language $\Pi' \in \text{NP}$, there is a polynomial-time ***Turing reduction*** from Π' to Π. A Turing reduction means a reduction that can be executed on a Turing machine; that is, a Turing machine M that can solve Π' using another Turing machine M' for Π as a black-box subroutine. Turing reductions are also called *oracle reductions*; polynomial-time Turing reductions are also called *Cook reductions*.

mathematics department in 1970, just one year before publishing his seminal paper; he (but not Levin) later won the Turing award for his proof.

[5]Interested readers find a proof in my lecture notes on nondeterministic Turing machines at http://algorithms.wtf, or in Boaz Barak's excellent *Introduction to Theoretical Computer Science*.

[6]Random-access machines are a model of computation that more faithfully models physical computers. A standard random-access machine has unbounded random-access memory, modeled as an unbounded array $M[0..\infty]$ where each address $M[i]$ holds a single w-bit integer, for some fixed integer w, and can read to or write from any memory addresses in constant time. RAM algorithms are formally written in assembly-like language, using instructions like **ADD** i, j, k (meaning "$M[i] \leftarrow M[j] + M[k]$"), **INDIR** i, j (meaning "$M[i] \leftarrow M[M[j]]$"), and **IF0GOTO** i, ℓ (meaning "if $M[i] = 0$, go to line ℓ"), but the precise instruction set is surprisingly irrelevant. By definition, each instruction executes in unit time. In practice, RAM algorithms can be faithfully described using higher-level pseudocode, as long as we're careful about arithmetic precision.

Researchers in complexity theory prefer to define NP-hardness in terms of polynomial-time ***many-one reductions***, which are also called *Karp reductions*. A *many-one* reduction from one language $L' \subseteq \Sigma^*$ to another language $L \subseteq \Sigma^*$ is a function $f : \Sigma^* \to \Sigma^*$ such that $x \in L'$ if and only if $f(x) \in L$. Then we can define a *language* L to be NP-hard if and only if, for any language $L' \in$ NP, there is a many-one reduction from L' to L that can be computed in polynomial time.

Every Karp reduction "is" a Cook reduction, but not vice versa. Specifically, any Karp reduction from one decision problem Π to another decision Π' is equivalent to transforming the input to Π into the input for Π', invoking an oracle (that is, a subroutine) for Π', and then returning the answer verbatim. However, as far as we know, not every Cook reduction can be simulated by a Karp reduction.

Complexity theorists prefer Karp reductions primarily because NP is closed under Karp reductions, but is *not* closed under Cook reductions (unless NP=co-NP, which is considered unlikely). There are natural problems that are (1) NP-hard with respect to Cook reductions, but (2) NP-hard with respect to Karp reductions only if P=NP. One trivial example of such a problem is UNSAT: Given a boolean formula, is it *always false*? On the other hand, many-one reductions apply *only* to decision problems (or more formally, to languages); formally, no optimization or construction problem is Karp-NP-hard.

To make things even more confusing, both Cook and Karp originally defined NP-hardness in terms of ***logarithmic-space*** reductions. Every logarithmic-space reduction is a polynomial-time reduction, but (as far as we know) not vice versa. It is an open question whether relaxing the set of allowed (Cook or Karp) reductions from logarithmic-space to polynomial-time changes the set of NP-hard problems.

Fortunately, none of these subtleties rear their ugly heads in practice—in particular, every reduction described in this chapter can be formalized as a logarithmic-space many-one reduction—so you can wake up now.

12.5 Reductions and SAT

To prove that any problem other than circuit satisfiability is NP-hard, we use a *reduction argument*. Reducing problem A to another problem B means describing an algorithm to solve problem A under the assumption that an algorithm for problem B already exists. You've already been doing reduction for years, even before starting this book, only you probably called them something else, like subroutines or utility functions or modular programming or using a calculator. To prove something is NP-hard, we describe a similar transformation between problems, but not in the direction that most people expect.

You should tattoo the following rule of onto the back of your hand, right next to your mom's birthday and the *actual* rules of Monopoly.[7]

To prove that problem A is NP-hard, reduce a known NP-hard problem to A.

In other words, to prove that your problem is hard, you need to describe an efficient algorithm to solve a *different* problem, which you already know is hard, using an hypothetical efficient algorithm for *your* problem as a black-box subroutine. The essential logic is a proof by contradiction. The reduction implies that if your problem were easy, then the other problem would be easy, which it ain't. Equivalently, since you know the other problem is hard, the reduction implies that your problem must also be hard; your hypothetical efficient algorithm does not actually exist.

As a canonical example, consider the *formula satisfiability* problem, usually just called ***SAT***. The input to SAT is a boolean *formula* like

$$(a \vee b \vee c \vee \bar{d}) \Leftrightarrow ((b \wedge \bar{c}) \vee \overline{(\bar{a} \Rightarrow d)} \vee (c \neq a \wedge b)),$$

and the question is whether it is possible to assign boolean values to the variables $a, b, c, \ldots$ so that the entire formula evaluates to TRUE.

To prove that SAT is NP-hard, we need to give a reduction from a known NP-hard problem. The only problem we know is NP-hard so far is CIRCUITSAT, so let's start there.

Let K be an arbitrary boolean circuit. We can transform (or more accurately, *transcribe*) K into a boolean formula Φ as follows. First, label each interior wire by a new variable y_j, and label the output wire with a new variable z. The formula Φ consists of a list of equations, one for each gate, separated by ANDS, followed by a final $\wedge z$. Figure 12.5 shows the resulting transcription for our example circuit.

Now we claim that the original circuit K is satisfiable ***if and only if*** the resulting formula Φ is satisfiable. Like every other "if and only if" statement, we prove this claim in two steps:

[7] If a player lands on an available property and declines (or is unable) to buy it, that property is immediately auctioned off to the highest bidder; the player who originally declined the property may bid, and bids may be arbitrarily higher or lower than the list price. Players in Jail can still buy and sell property, buy and sell houses and hotels, and collect rent. The game has 32 houses and 12 hotels; once they're gone, they're gone. In particular, if all houses are already on the board, you cannot downgrade a hotel to four houses; you must raze all the hotels in the group to the ground. Players can sell or exchange *undeveloped* properties with each other, but cannot sell property back to the bank; on the other hand, players can sell buildings to the bank (at half price), but cannot sell or exchange buildings with each other. All penalties are paid directly to the bank. A player landing on Free Parking does not win anything. A player landing on Go gets exactly $200, no more. Railroads are not magic transporters. Finally, Jeff *always* gets the boot. No, not the T-Rex or the penguin—the *boot*, dammit.

$$(y_1 = x_1 \wedge x_4) \wedge (y_2 = \overline{x_4}) \wedge (y_3 = x_3 \wedge y_2) \wedge (y_4 = y_1 \vee x_2) \wedge$$
$$(y_5 = \overline{x_2}) \wedge (y_6 = \overline{x_5}) \wedge (y_7 = y_3 \vee y_5) \wedge (z = y_4 \wedge y_7 \wedge y_6) \wedge z$$

Figure 12.5. Transcribing a boolean circuit as a boolean formula.

$\Rightarrow$ Given a set of inputs that satisfy the circuit K, we can derive a satisfying assignment for the formula Φ by computing the output of every gate in K.

$\Leftarrow$ Given a satisfying assignment for the formula Φ, we can obtain a satisfying input the circuit by simply ignoring the internal wire variables y_i and the output variable z.

The entire transformation from circuit to formula can be carried out in linear time. Moreover, the size of the resulting formula is at most a constant factor larger than any reasonable representation of the circuit.

Now suppose, for the sake of argument, there is an algorithm that can determine in polynomial time whether a given boolean formula is satisfiable. Then given any boolean circuit K, we can decide whether K is satisfiable by first transforming K into a boolean formula Φ as described above, and then asking our magical mystery SAT algorithm whether Φ is satisfiable, as suggested by the following cartoon. Each box represents an algorithm. The red box on the left is the transformation subroutine. The box on the right is the hypothetical magic SAT algorithm. It *must* be magic, because it has a *rainbow* on it.[8]

If you prefer magic pseudocode to magic boxes:

[8]Kay Erickson, personal communication, 2011. For those of you reading black-and-white printed copies: Yes, that round thing is a rainbow.

```
CircuitSat(K):
  transcribe K into a boolean formula Φ
  return Sat(Φ)        ⟨⟨ ★★★MAGIC★★★ ⟩⟩
```

Transcribing K into Φ requires only polynomial time (in fact, only *linear* time, but whatever), so the entire CircuitSat algorithm also runs in polynomial time.

$$T_{\text{CircuitSat}}(n) \leq O(n) + T_{\text{Sat}}(O(n))$$

We conclude that any polynomial-time algorithm for Sat would give us a polynomial-time algorithm for CircuitSat, which in turn would imply P=NP. So Sat is NP-hard!

12.6 3Sat (from CircuitSat)

A special case of Sat that is particularly useful in proving NP-hardness results is called ***3CNF-Sat*** or more often simply ***3Sat***.

A boolean formula is in *conjunctive normal form* (CNF) if it is a conjunction (and) of several *clauses*, each of which is the disjunction (or) of several *literals*, each of which is either a variable or its negation. For example:

$$\overbrace{(a \vee b \vee c \vee d)}^{\text{clause}} \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b})$$

A *3CNF* formula is a CNF formula with exactly three literals per clause; the previous example is not a 3CNF formula, since its first clause has four literals and its last clause has only two. 3Sat is the restriction of Sat to 3CNF formulas: Given a 3CNF formula, is there an assignment to the variables that makes the formula evaluate to True?

We could prove that 3Sat is NP-hard by a reduction from the more general Sat problem, but it's actually easier to start over from scratch, by reducing directly from CircuitSat.

Figure 12.6. A polynomial-time reduction from CircuitSat to 3Sat.

Given an arbitrary boolean circuit K, we transform K into an equivalent 3CNF formula in several stages. Except for the very last stage, this reduction

was actually described by Grigorii Tseitin in 1966, five years before Cook and Levin reported their proofs of the Cook-Levin Theorem. (In the same 1966 paper, Tseitin described the problem we now call CNF-SAT, possibly for the first time.) As we describe each stage, we will also prove that stage is correct.

- *Make sure every* AND *and* OR *gate in K has exactly two inputs.* If any gate has $k > 2$ inputs, replace it with a binary tree of $k-1$ binary gates. Call the resulting circuit K'. The circuits K and K' are logically equivalent circuits, so every satisfying input for K is a satisfying input for K' and vice versa.

- *Transcribe K' into a boolean formula Φ_1 with one clause per gate,* exactly as in our previous reduction to SAT. We already proved that every satisfying input for K' can be transformed into a satisfying assignment for Φ_1 and vice versa.

- *Replace each clause in Φ_1 with a CNF formula.* There are only three types of clauses in Φ_1, one for each type of gate in K':

$$\begin{aligned} a = b \wedge c &\longmapsto (a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee c) \\ a = b \vee c &\longmapsto (\bar{a} \vee b \vee c) \wedge (a \vee \bar{b}) \wedge (a \vee \bar{c}) \\ a = \bar{b} &\longmapsto (a \vee b) \wedge (\bar{a} \vee \bar{b}) \end{aligned}$$

  Call the resulting CNF formula Φ_2. Because Φ_1 and Φ_2 are logically equivalent formulas, every satisfying assignment for Φ_1 is also a satisfying assignment for Φ_2, and vice versa.

- *Replace each clause in Φ_2 with a 3CNF formula.* Every clause in Φ_2 has *at most* three literals, but we need clauses with *exactly* three literals. To obtain a 3CNF formula, we expand each two-literal clause in Φ_2 into two three-literal clauses by introducing one new variable, and we expand the final one-literal clause in Φ_2 into four three-literal clauses by introducing two new variables.

$$\begin{aligned} a \vee b &\longmapsto (a \vee b \vee x) \wedge (a \vee b \vee \bar{x}) \\ z &\longmapsto (z \vee x \vee y) \wedge (z \vee \bar{x} \vee y) \wedge (z \vee x \vee \bar{y}) \wedge (z \vee \bar{x} \vee \bar{y}) \end{aligned}$$

  Call the resulting 3CNF formula Φ_3. Every satisfying assignment for Φ_2 can be transformed into a satisfying assignment for Φ_3 by assigning *arbitrary* values to the new variables (x and y). Conversely, every satisfying assignment for Φ_3 can be transformed into a satisfying assignment for Φ_2 by ignoring the new variables.

For example, our example circuit is transformed into the following 3CNF formula; compare with Figure 12.5.

$$
\begin{gathered}
(y_1 \vee \overline{x_1} \vee \overline{x_4}) \wedge (\overline{y_1} \vee x_1 \vee z_1) \wedge (\overline{y_1} \vee x_1 \vee \overline{z_1}) \wedge (\overline{y_1} \vee x_4 \vee z_2) \wedge (\overline{y_1} \vee x_4 \vee \overline{z_2}) \\
\wedge (y_2 \vee x_4 \vee z_3) \wedge (y_2 \vee x_4 \vee \overline{z_3}) \wedge (\overline{y_2} \vee \overline{x_4} \vee z_4) \wedge (\overline{y_2} \vee \overline{x_4} \vee \overline{z_4}) \\
\wedge (y_3 \vee \overline{x_3} \vee \overline{y_2}) \wedge (\overline{y_3} \vee x_3 \vee z_5) \wedge (\overline{y_3} \vee x_3 \vee \overline{z_5}) \wedge (\overline{y_3} \vee y_2 \vee z_6) \wedge (\overline{y_3} \vee y_2 \vee \overline{z_6}) \\
\wedge (\overline{y_4} \vee y_1 \vee x_2) \wedge (y_4 \vee \overline{x_2} \vee z_7) \wedge (y_4 \vee \overline{x_2} \vee \overline{z_7}) \wedge (y_4 \vee \overline{y_1} \vee z_8) \wedge (y_4 \vee \overline{y_1} \vee \overline{z_8}) \\
\wedge (y_5 \vee x_2 \vee z_9) \wedge (y_5 \vee x_2 \vee \overline{z_9}) \wedge (\overline{y_5} \vee \overline{x_2} \vee z_{10}) \wedge (\overline{y_5} \vee \overline{x_2} \vee \overline{z_{10}}) \\
\wedge (y_6 \vee x_5 \vee z_{11}) \wedge (y_6 \vee x_5 \vee \overline{z_{11}}) \wedge (\overline{y_6} \vee \overline{x_5} \vee z_{12}) \wedge (\overline{y_6} \vee \overline{x_5} \vee \overline{z_{12}}) \\
\wedge (\overline{y_7} \vee y_3 \vee y_5) \wedge (y_7 \vee \overline{y_3} \vee z_{13}) \wedge (y_7 \vee \overline{y_3} \vee \overline{z_{13}}) \wedge (y_7 \vee \overline{y_5} \vee z_{14}) \wedge (y_7 \vee \overline{y_5} \vee \overline{z_{14}}) \\
\wedge (y_8 \vee \overline{y_4} \vee \overline{y_7}) \wedge (\overline{y_8} \vee y_4 \vee z_{15}) \wedge (\overline{y_8} \vee y_4 \vee \overline{z_{15}}) \wedge (\overline{y_8} \vee y_7 \vee z_{16}) \wedge (\overline{y_8} \vee y_7 \vee \overline{z_{16}}) \\
\wedge (y_9 \vee \overline{y_8} \vee \overline{y_6}) \wedge (\overline{y_9} \vee y_8 \vee z_{17}) \wedge (\overline{y_9} \vee y_6 \vee z_{18}) \wedge (\overline{y_9} \vee y_6 \vee \overline{z_{18}}) \wedge (\overline{y_9} \vee y_8 \vee \overline{z_{17}}) \\
\wedge (y_9 \vee z_{19} \vee z_{20}) \wedge (y_9 \vee \overline{z_{19}} \vee z_{20}) \wedge (y_9 \vee z_{19} \vee \overline{z_{20}}) \wedge (y_9 \vee \overline{z_{19}} \vee \overline{z_{20}})
\end{gathered}
$$

Yikes! At first glance, this formula might look a *lot* more complicated than the original circuit, but in fact, it's only larger by a constant factor. Specifically, the simplified circuit K' has at most twice as many wires as the original circuit K, each binary gate in K' is transformed into at most five clauses in Φ_3. Even if the formula size were a large *polynomial* function (like n^{374}) of the circuit size, we would still have a valid reduction.

Our reduction transforms an arbitrary boolean circuit K into a 3CNF formula Φ_3 in polynomial time (in fact, in linear time). Moreover, any satisfying input for the input circuit K can be transformed into a satisfying assignment for Φ_3, and any satisfying assignment for Φ_3 can be transformed into a satisfying input for K. In other words, K is satisfiable if and only if Φ_3 is satisfiable. Thus, if 3Sat can be solved in polynomial time, then CircuitSat can be solved in polynomial time, which implies that $\mathrm{P} = \mathrm{NP}$. We conclude that 3Sat is NP-hard.

12.7 Maximum Independent Set (from 3Sat)

For the next few problems we consider, the input is a simple, unweighted, undirected graph, and the problem asks for the size of the largest or smallest subgraph satisfying some structural property.

Let G be an arbitrary graph. An ***independent set*** in G is a subset of the vertices of G with no edges between them. The *maximum independent set* problem, which I'll abbreviate ***MaxIndSet***, asks for the size of the largest independent set in a given graph. I will prove that MaxIndSet is NP-hard using a reduction from 3Sat, as suggested by Figure 12.7.

Given an arbitrary 3CNF formula Φ, we construct a graph G as follows. Let k denote the number of clauses in Φ. The graph G contains exactly $3k$ vertices, one for each literal in Φ. Two vertices in G are connected by an edge if and only if either (1) they correspond to literals in the same clause, or (2) they correspond to a variable and its inverse. For example, the formula $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$ is transformed into the graph shown in Figure 12.8.

Figure 12.7. A polynomial-time reduction from 3SAT to MAXINDSET.

$$(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$$

Figure 12.8. A graph derived from the satisfiable 3CNF formula with 4 clauses, and an independent set of size 4.

Each independent set in G contains at most one vertex from each clause triangle, because any two vertices in each triangle are connected. Thus, the largest independent set in G has size *at most* k. I claim that G contains an independent set of size *exactly* k if and only if the original formula Φ is satisfiable. As usual for "if and only if" statements, the proof consists of two parts.

$\Rightarrow$ Suppose Φ is satisfiable. Fix an arbitrary satisfying assignment. By definition, each clause in Φ contains at least one TRUE literal. Thus, we can choose a subset S of k vertices in G that contains exactly one vertex per clause triangle, such that the corresponding k literals are all TRUE. Because each triangle contains at most one vertex in S, no two vertices in S are connected by a triangle edge. Because every literal corresponding to a vertex in S is TRUE, no two vertices in S are connected by a negation edge. We conclude that S is an independent set of size k in G.

$\Leftarrow$ On the other hand, suppose G contains an independent set S of size k. Each vertex in S must lie in a different clause triangle. Suppose we assign the value TRUE to each literal in S; because contradictory literals are connected by edges, this assignment is consistent. There may be variables x such that neither x nor $\bar{x}$ corresponds to a vertex in S; we can set these variables to

any value we like. Because S contains one vertex in each clause triangle, each clause in Φ contains (at least) one TRUE literal. We conclude that Φ is satisfiable.

Transforming the 3CNF formula Φ into the graph G takes polynomial time, even if we do everything by brute force. Thus, if we could solve MAXINDSET in polynomial time, then we could also solve 3SAT in polynomial time, by transforming the input formula Φ into a graph G and comparing the size of the largest independent set in G with the number of clauses in Φ. But that would imply P=NP, which is ridiculous! We conclude that MAXINDSET is NP-hard.

12.8 The General Pattern

All NP-hardness proofs—and more generally, all polynomial-time reductions—follow the same general outline. To reduce problem X to problem Y in polynomial time, we need to do three things:

1. Describe a polynomial-time algorithm to transform an ***arbitrary*** instance of x of X into a *special* instance y of Y.
2. Prove that if x is a "good" instance of X, then y is a "good" instance of Y.
3. Prove that if y is a "good" instance of Y, then x is a "good" instance of X. (This is usually the part that causes the most trouble.)

Of course, *developing* a correct reduction doesn't mean handling these three tasks one at a time. *First* writing down an algorithm that *seems* to work and *then* trying prove that it *actually* works is rarely successful, especially in time-limited settings like exams. We *must* develop the algorithm, the "if" proof, and the "only if" proof simultaneously.

To quote the late great Ricky Jay:[9] This is an *acquired* skill.

One point that confuses many students is that the reduction algorithm only "works one way"—from X to Y—but the correctness proof needs to "work both ways". But the correctness proofs are not actually symmetric. The "if" proof needs to handle *arbitrary* instances of X, but the "only if" only needs to handle the *special* instances of Y produced by the reduction algorithm. Exploiting this asymmetry is the key to successfully designing correct reductions.

I find it useful to think in terms of transforming *certificates*—proofs that a given instance is "good"—along with the instances themselves. For example, a certificate for CIRCUITSAT is a set of inputs that turns on the light bulb; a certificate for SAT or 3SAT is a satisfying assignment; a certificate for MAXINDSET is a large independent set. To reduce X to Y, we actually need to design *three* algorithms, one for each of the following tasks:

[9] from his 1996 off-Broadway show *Ricky Jay and his 52 Assistants*

- Transform an arbitrary instance x of X into a special instance y of Y in polynomial time.
- Transform an arbitrary certificate for x into a certificate for y, and
- Transform an arbitrary certificate for y into a certificate for x.

The second and third tasks refer to the input and output of the first algorithm. The *certificate* transformation needs to be reversible, not the *instance* transformation. We never have to transform instances of Y, and we don't need to think about *arbitrary* instances of Y at all. Only the first algorithm needs to run in polynomial time (although in practice, the second and third algorithms are almost always simpler than the first).

For example, our reduction from CircuitSat to 3Sat consists of three algorithms:

- The first transforms an *arbitrary* boolean circuit K into a special 3CNF boolean formula Φ_3, in polynomial time. (Encode each wire as a variable and each gate as a sub-formula, and then expand each sub-formula into 3CNF.)
- The second transforms an *arbitrary* satisfying input for K into a satisfying assignment for Φ_3. (Trace the input through the circuit, transfer values from each wire to the corresponding variable, and give any additional variables arbitrary values.)
- The third transforms an *arbitrary* satisfying assignment for Φ_3 into a satisfying input for K. (Transfer values from each wire variable in Φ_3 to the corresponding wire in K.)

The reduction works because the first algorithm *encodes* any boolean circuit K into a highly structured 3CNF formula Φ_3. The specific structure of Φ_3 restricts *how* it can be satisfied; every satisfying assignment for Φ_3 must "come from" some satisfying input for K. We don't have to think about arbitrary 3CNF formulas at all.

Similarly, our reduction from 3Sat to MaxIndSet consists of three algorithms:

- The first transforms an *arbitrary* 3CNF formula Φ into a special graph G and a specific integer k, in polynomial time.
- The second transforms an *arbitrary* satisfying assignment for Φ into an independent set in G of size k.
- The third transforms an *arbitrary* independent set in G of size k into a satisfying assignment for Φ.

Again, our first transformation *encodes* the input formula Φ into a highly structured graph G and a specific integer k. The structure of G ensures that every independent set of size k "comes from" a satisfying assignment for Φ. We don't consider arbitrary graphs or arbitrary independent set sizes at all.

12.9 Clique and Vertex Cover (from Independent Set)

A ***clique*** is another name for a complete graph, that is, a graph where every pair of vertices is connected by an edge. The MAXCLIQUE problem asks for the number of nodes in its largest complete subgraph in a given graph. A ***vertex cover*** of a graph is a set of vertices that touches every edge in the graph. The MINVERTEXCOVER problem asks for the size of the smallest vertex cover in a given graph.

Figure 12.9. A graph whose largest independent set, largest clique, and smallest vertex cover all have size 4.

We can prove that MAXCLIQUE is NP-hard using the following easy reduction from MAXINDSET. Any graph G has an *edge-complement* $\overline{G}$ with the same vertices, but with exactly the opposite set of edges—uv is an edge in $\overline{G}$ if and only if uv is *not* an edge in G. A set of vertices is independent in G if and only if the same vertices define a clique in $\overline{G}$. Thus, the largest independent in G has the same vertices (and thus the same size) as the largest clique in the complement of G.

Figure 12.10. An easy reduction from MAXINDSET to MAXCLIQUE.

The proof that MINVERTEXCOVER is NP-hard is even simpler, because it relies on the following easy observation: I is an independent set in a graph $G = (V, E)$ if and only if its complement $V \setminus I$ is a vertex cover of the same graph G. Thus, the *largest* independent set in any graph is the complement of the *smallest* vertex cover of the same graph! Thus, if the smallest vertex cover in an n-vertex graph has size k, then the largest independent set has size $n - k$.

Figure 12.11. An even easier reductions from MaxIndSet to MinVertexCover.

12.10 Graph Coloring (from 3SAT)

A ***proper k-coloring*** of a graph $G = (V, E)$ is a function $C : V \to \{1, 2, \ldots, k\}$ that assigns one of k "colors" to each vertex, so that every edge has two different colors at its endpoints. (The "colors" are really arbitrary labels, which for simplicity we represent by small positive integers, rather than electromagnetic frequencies, CMYK vectors, or Pantone numbers, for example.) The graph coloring problem asks for the smallest possible number of colors in a legal coloring of a given graph.

To prove that graph coloring is NP-hard, it suffices to consider the decision problem ***3Color***: Given a graph, does it have a proper 3-coloring? We prove 3Color is NP-hard using a reduction from 3Sat. (Why 3Sat? Because it has a 3 in it. You probably think I'm joking, but I'm not.) Given a 3CNF formula Φ, we construct a graph G that is 3-colorable if and only if Φ is satisfiable, as suggested by the usual diagram.

We describe the reduction using a standard strategy of decomposing the output graph G into ***gadgets***, subgraphs that enforce various semantics of the input formula Φ in the language of graph coloring. Decomposing reductions into separate gadgets is not only helpful for understanding existing reductions and proving them correct, but for designing new NP-hardness reductions.[10]

[10] Our reduction from CircuitSat to Sat encoding each gate in the input circuit as a clause in the output formula; these clauses are "gate gadgets". Similarly, our reduction from 3Sat to MaxIndSet used two type of gadgets: "clause gadgets" (triangles) and "variable gadgets" (edges between contradicting literals).

Our formula-to-graph reduction uses three types of gadgets:

- There is a single *truth gadget*: a triangle with three vertices T, F, and X, which intuitively stand for TRUE, FALSE, and OTHER. Since these vertices are all connected, they must have different colors in any 3-coloring. For the sake of convenience, we will *name* those colors TRUE, FALSE, and OTHER. Thus, when we say that a node is colored TRUE, we mean that it has the same color as vertex T.

- For each variable a, the graph contains a *variable gadget*, which is a triangle joining two new nodes labeled a and $\overline{a}$ to node X in the truth gadget. Node a must be colored either TRUE or FALSE, and therefore node $\overline{a}$ must be colored either FALSE or TRUE, respectively.

Figure 12.12. The truth gadget and a variable gadget for a.

- Finally, for each clause in Φ, the graph contains a *clause gadget*. Each clause gadget joins three literal nodes (from the corresponding variable gadgets) to node T (from the truth gadget) using five new unlabeled nodes and ten edges, as shown below.

Figure 12.13. A clause gadget for $(a \vee b \vee \bar{c})$.

In effect, each triangle in the clause gadget behaves like a "majority gate". In any valid 3-coloring, if the two vertices to the left of the triangle have the same color, the rightmost vertex of the triangle must have the same color; on the other hand, if the two left vertices have different colors, the color of the right vertex can be chosen arbitrarily. See Figure 12.14.

It follows that there is no valid 3-coloring of a clause gadget where all three literal nodes are colored FALSE. On the other hand, any coloring of the literal nodes with more than one color can be extended to a valid 3-coloring of the clause gadget. The variable gadgets force each literal node to be colored either TRUE or FALSE; thus, in any valid 3-coloring of the clause gadget, at least one literal node is colored TRUE.

Figure 12.14. All valid 3-colorings of a "half-gadget", up to permutations of the colors

The final graph G contains exactly *one* node T, exactly *one* node F, and exactly *two* nodes a and $\bar{a}$ for each variable. For example, Figure 12.15 shows the graph that results from the same 3CNF formula $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$ that we previously used to illustrate the MaxIndSet reduction in Figure 12.8. The 3-coloring is one of several that correspond to the satisfying assignment $a = c =$ True, $b = d =$ False.

$(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$

Figure 12.15. The 3-colorable graph derived from a satisfiable 3CNF formula.

We've already done most of the work for a proof of correctness. If the formula is satisfiable, then we can color the literal nodes according to any satisfying assignment, and then (because each clause is satisfied) extend the coloring across every clause gadget. On the other hand, if the graph is 3-colorable, then we can extract a satisfying assignment from any 3-coloring—at least one of the three literal nodes in every clause gadget is colored True.

Because 3Color is a special case of the more general graph coloring problem—

What is the minimum number of colors?—the more general optimization problem is also NP-hard.

12.11 Hamiltonian Cycle

A ***Hamiltonian cycle*** in a graph is a cycle that visits every vertex exactly once. (This is very different from an *Euler circuit*, which is a closed *walk* that traverses every *edge* exactly once; Euler circuits are easy to find and construct in linear time using depth-first search.) Here we consider two different proofs that the Hamiltonian cycle problem in *directed* graphs is NP-hard.

From Vertex Cover

Our first NP-hardness proof reduces from the decision version of the vertex cover problem. Given an *undirected* graph G and an integer k, we construct a *directed* graph H, such that H has a Hamiltonian cycle if and only if G has a vertex cover of size k. As in our previous reductions, the output graph H is composed of several gadgets, each corresponding to certain features of the inputs G and k.

- For each undirected edge uv in G, the directed graph H contains an *edge gadget* consisting of four vertices (u, v, in), (u, v, out), (v, u, in), (v, u, out) and six directed edges

$$(u,v,\text{in})\to(u,v,\text{out}) \qquad (u,v,\text{in})\to(v,u,\text{in}) \qquad (v,u,\text{in})\to(u,v,\text{in})$$
$$(v,u,\text{in})\to(v,u,\text{out}) \qquad (u,v,\text{out})\to(v,u,\text{out}) \qquad (v,u,\text{out})\to(u,v,\text{out})$$

  as shown in Figure 12.16. Each "in" vertex has an additional incoming edge, and each "out" vertex has an additional outgoing edge. Any Hamiltonian cycle in H must pass through an edge gadget in one of three ways—either straight through on both sides, or with a detour from one side to the other and back. Eventually, these options will correspond to both u and v, only u, or only v belonging to some vertex cover.
- For each vertex u in G, all the edge gadgets for incident edges uv are connected in H into a single directed path, which we call a *vertex chain*. Specifically, suppose vertex u has d neighbors $v_1, v_2, \ldots, v_d$. Then H has $d-1$ additional edges $(u, v_i, \text{out})\to(u, v_{i+1}, \text{in})$ for each i from 1 to $d-1$.

Figure 12.16. An edge gadget and its only possible intersections with a Hamiltonian cycle.

- Finally, H also contains k *cover vertices* $x_1, x_2, \ldots, x_k$. Each cover vertex has a directed edge to the first vertex in each vertex chain, and a directed edge from the last vertex in each vertex chain.

Figure 12.17 shows a complete example of our transformation; each double-arrowed blue segment represents a pair of directed edges.

Figure 12.17. Example of our reduction from VERTEXCOVER to DIRECTEDHAMCYCLE.

As usual, we prove our reduction is correct in two stages.

⇒ First, suppose $C = \{u_1, u_2, \ldots, u_k\}$ is a vertex cover of G of size k. We can construct a Hamiltonian cycle in H that "encodes" C as follows. For each index i from 1 to k, we traverse a path from cover vertex x_i, through the vertex chain for u_i, to cover vertex x_{i+1} (or cover vertex x_1 if $i = k$). As we traverse the chain for each vertex u_i, we determine how to proceed from each node (u_i, v, in) as follows:

  - If $v \in C$, follow the edge $(u_i, v, \text{in}) \to (u_i, v, \text{out})$.
  - If $v \notin C$, detour through $(u_i, v, \text{in}) \to (v, u_i, \text{in}) \to (v, u_i, \text{out}) \to (u_i, v, \text{out})$.

  Thus, for each edge uv of G, the Hamiltonian cycle visits (u, v, in) and (u, v, out) as part of u's vertex chain if $u \in C$ and as part of v's vertex chain otherwise. See Figure 12.18.

Figure 12.18. Every vertex cover of size k in G corresponds to a Hamiltonian cycle in H and vice versa.

⇐ On the other hand, suppose H contains a Hamiltonian cycle C. This cycle must contain an edge from each cover vertex to the start of some vertex chain. Our case analysis of edge gadgets inductively implies that after C enters the vertex chain for some vertex u, it must traverse the entire vertex chain. Specifically, at each vertex (u, v, in), the cycle must contain either the single edge $(u, v, \text{in}) \to (u, v, \text{out})$ or the detour path $(u, v, \text{in}) \to (v, u, \text{in}) \to (v, u, \text{out}) \to (u, v, \text{out})$, followed by an edge to the next edge gadget in u's vertex chain, or to a cover vertex if this is the last edge gadget in u's vertex chain. In particular, if C contains the detour edge $(u, v, \text{in}) \to (v, u, \text{in})$, it cannot contain edges between any cover vertex and v's vertex chain. It follows that C traverses exactly k vertex chains. Moreover, these vertex chains describe a vertex cover of the original graph G, because C visits the vertex (u, v, in) for every edge uv in G.

We conclude that G has a vertex cover of size k if and only if H contains a Hamiltonian cycle. The transformation from G to H takes at most $O(V^2)$ time; it follows that the directed Hamiltonian cycle problem is NP-hard.

From 3SAT

We can also prove that the directed Hamiltonian cycle problem is NP-hard by reducing directly from 3SAT. Given an arbitrary 3CNF formula Φ with n variables $x_1, x_2, \ldots, x_n$ and k clauses $c_1, c_2, \ldots, c_k$, we construct a directed graph H that contains a Hamiltonian cycle if and only if Φ is satisfiable, as follows.

For each variable x_i, we construct a *variable gadget*, which consists of a doubly-linked list of $2k$ vertices $(i, 0), (i, 1), \ldots, (i, 2k)$, connected by edges $(i, j-1) \to (i, j)$ and $(i, j) \to (i, j-1)$ for each index j. We connect the first and

last nodes in each adjacent pair of variable gadgets by adding edges

$$(i,0)\to(i+1,0) \quad (i,2k)\to(i+1,0) \quad (i,0)\to(i+1,2k) \quad (i,2k)\to(i+1,2k)$$

for each index i; we also connect the endpoints of the first and last variable gadgets with the edges

$$(n,0)\to(1,0) \qquad (n,2k)\to(1,0) \qquad (n,0)\to(1,2k) \qquad (n,2k)\to(1,2k).$$

The resulting graph G has exactly 2^n Hamiltonian cycles, one for each assignment of boolean values to the n variables of Φ. Specifically, for each i, we traverse the ith variable gadget from left to right if $x_i = \textsc{True}$ and right to left if $x_i = \textsc{False}$. See Figure 12.19.

Figure 12.19. Left: Variable gadgets and connectors in G, for any formula with 4 variables and 4 clauses. Right: The Hamiltonian cycle in G corresponding to the assignment $a = b = d = \textsc{True}$ and $c = \textsc{False}$

Now we extend G to a larger graph H by adding a *clause vertex* $[j]$ for each clause c_j, connected to the variable gadgets by six edges, as shown in Figure 12.20. For each positive literal x_i in c_j, we add the edges $(i,2j-1)\to[j]\to(i,2j)$, and for each negative literal $\bar{x}_i$ in c_j, we add the edges $(i,2j)\to[j]\to(i,2j-1)$. The connections to the clause vertices guarantee that a Hamiltonian cycle in G can be extended to a Hamiltonian cycle in H if and only if the corresponding variable assignment satisfies Φ. Exhaustive case analysis now implies that H has a Hamiltonian cycle if and only if Φ is satisfiable.

Transforming the formula Φ into the graph H takes $O(kn)$ time, which is at most quadratic in the total length of the formula; we conclude that the directed Hamiltonian cycle problem is NP-hard.

Variants and Extensions

Trivial modifications of the previous reductions imply that the Hamiltonian *path* problem in directed graphs is also NP-hard. A Hamiltonian path in a graph G is of course a simple path that visits every vertex of G exactly once. In fact, there

Figure 12.20. Left: Clause gadgets for the formula $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$. Right: A hamiltonian cycle in H corresponding to the satisfying assignment $a = b = d =$ TRUE and $c =$ FALSE.

are simple polynomial-time reductions from the Hamiltonian cycle problem to the Hamiltonian path problem and vice versa. I'll leave the details of these reductions as exercises.

Both of the previous reductions deal with directed graphs, but the corresponding question in undirected graph is also NP-hard. In fact, there is a relatively simple reduction from the directed Hamiltonian cycle/path problem to the undirected Hamiltonian cycle/path problem; again, I'll leave the details of this reduction as an exercise.

Finally, the infamous *traveling salesman problem* asks to find the shortest Hamiltonian cycle (or path) in a graph with weighted edges. Since finding the shortest cycle/path is obviously harder than determining if a cycle/path exists at all—Consider a graph where every edge has weight 1!—TRAVELINGSALESMAN is also NP-hard.

12.12 Subset Sum (from Vertex Cover)

The next problem that we prove NP-hard is the SUBSETSUM problem considered in Chapter 2: Given a set X of positive integers and an integer T, determine whether X has a subset whose elements sum to T.

We once again reduce from VERTEXCOVER. Given a graph G and an integer k, we need to compute a set X of positive integers and an integer T, such that X has a subset that sums to T if and only if G has an vertex cover of size k. Our transformation uses only two types of "gadgets", which are *integers* representing the vertices and edges in G.

Number the *edges* of G arbitrarily from 0 to $E - 1$. Our set X contains the

integer $b_i := 4^i$ for each edge i, and the integer

$$a_v := 4^E + \sum_{i \in \Delta(v)} 4^i$$

for each vertex v, where $\Delta(v)$ is the set of edges that have v has an endpoint. Alternately, we can think of each integer in X as an $(E+1)$-digit number written in base 4. The Eth digit is 1 if the integer represents a vertex, and 0 otherwise; and for each $i < E$, the ith digit is 1 if the integer represents edge i or one of its endpoints, and 0 otherwise. Finally, we set the target sum

$$T := k \cdot 4^E + \sum_{i=0}^{E-1} 2 \cdot 4^i.$$

Now let's prove that the reduction is correct.

$\Rightarrow$ First, suppose G has a vertex cover C of size k. Consider the subset

$$X' := \{a_v \mid v \in C\} \cup \{b_i \mid \text{edge } i \text{ has exactly one endpoint in } C\}$$

The sum of the elements of X', written in base 4, has most significant digit k and all other digits equal to 2. Thus, the elements of X' sum to exactly T.

$\Leftarrow$ On the other hand, suppose there is a subset $X' \subseteq X$ that sums to T. Specifically, we must have

$$\sum_{v \in V'} a_v + \sum_{i \in E'} b_i = t$$

for some subsets $V' \subseteq V$ and $E' \subseteq E$. Again, if we sum these base-4 numbers, there are no carries in the first E digits, because for each i there are only three numbers in X whose ith digit is 1. Each edge number b_i contributes only one 1 to the ith digit of the sum, but the ith digit of t is 2. Thus, for each edge in G, at least one of its endpoints must be in V'. In other words, V' is a vertex cover. On the other hand, only vertex numbers are larger than 4^E, and $\lfloor T/4^E \rfloor = k$, so V' has at most k elements. (In fact, it's not hard to see that V' has *exactly* k elements.)

For example, given the four-vertex graph $G = (V, E)$ where $V = \{u, v, w, x\}$ and $E = \{uv, uw, vw, vx, wx\}$, our set X might contain the following base-4 integers:

$$\begin{aligned} a_u &:= 111000_4 = 1344 & b_{uv} &:= 010000_4 = 256 \\ a_v &:= 110110_4 = 1300 & b_{uw} &:= 001000_4 = 64 \\ a_w &:= 101101_4 = 1105 & b_{vw} &:= 000100_4 = 16 \\ a_x &:= 100011_4 = 1029 & b_{vx} &:= 000010_4 = 4 \\ & & b_{wx} &:= 000001_4 = 1 \end{aligned}$$

If we are looking for a vertex cover of size $k = 2$, our target sum would be $T := 222222_4 = 2730$. Indeed, the vertex cover $\{v, w\}$ corresponds to the subset $\{a_v, a_w, b_{uv}, b_{uw}, b_{vx}, b_{wx}\}$, whose sum is $1300+1105+256+64+4+1 = 2730$.

The reduction can clearly be performed in polynomial time. We've already proved that VertexCover is NP-hard, so it follows that SubsetSum is NP-hard.

Caveat Reductor!

One subtle point must be emphasized here. 300-something pages ago, back in Chapter 3, we developed a dynamic programming algorithm to solve SubsetSum in $O(nT)$ time. Isn't this a polynomial-time algorithm? Didn't we just prove that P=NP? Hey, where's my million dollars?!

Alas, life is not so simple. True, the running time is a polynomial function of the variables n and T, but to qualify as a true polynomial-time algorithm, the running time must be a polynomial function of the *input size*—the number of *bits* required to represent the input. The *values* of the elements of X and the target sum T could be exponentially larger than the number of input bits. Indeed, the reduction we just described produces a value of T that is exponentially larger than the size of our original input graph, which forces our dynamic programming algorithm to run in exponential time.

Algorithms like this are said to run in ***pseudo-polynomial time***, and any NP-hard problem with such an algorithm is called ***weakly NP-hard***. Equivalently, a weakly NP-hard problem is a problem that can be solved in polynomial time when all input numbers are represented in *unary* (as a sum of 1s), but becomes NP-hard when all input numbers are represented in *binary*. If a problem is NP-hard even when all the input numbers are represented in unary, we say that the problem is ***strongly NP-hard***. A good example of a *strongly* NP-hard problem is TravelingSalesman, which remains NP-hard even if the input graph is complete and all edge weights are equal to 1 or 2.

12.13 Other Useful NP-hard Problems

Literally thousands of problems have been proved to be NP-hard. Here I will list a few NP-hard problems that are useful in deriving reductions.[11] I won't describe the NP-hardness proofs for these problems in detail, but you can find most of them in Garey and Johnson's classic *Scary Black Book of NP-Completeness*.[12] All

[11] As someday it may happen a reduction must be found, I've got a little list. I've got a little list. Of some Herculean problems we can use to lower bound. Fast solutions don't exist. Our proofs they can assist.

[12] Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., 1979. And yes, it really is black.

of the problems I've discussed so far, and most of the problems in the following list, were first proved NP-hard in a single landmark 1972 paper by Richard Karp.[13]

- **PlanarCircuitSat:** Given a boolean circuit that can be embedded in the plane so that no two wires cross, is there an input that makes the circuit output True? This problem can be proved NP-hard by reduction from the general circuit satisfiability problem, by replacing each crossing with a small assemblage of gates.
- **1-in-3Sat:** Given a 3CNF formula, is there an assignment of values to the variables so that each clause contains *exactly* one True literal? This problem can be proved NP-hard by reduction from the usual 3Sat.
- **NotAllEqual3Sat:** Given a 3CNF formula, is there an assignment of values to the variables so that every clause contains at least one True literal *and* at least one False literal? This problem can be proved NP-hard by reduction from the usual 3Sat.
- **Planar3Sat:** Given a 3CNF boolean formula, consider a bipartite graph whose vertices are the clauses and variables, where an edge indicates that a variable (or its negation) appears in a clause. If this graph is planar, the 3CNF formula is also called planar. The Planar3Sat problem asks, given a planar 3CNF formula, whether it has a satisfying assignment. This problem can be proved NP-hard by reduction from PlanarCircuitSat.[14]
- **Exact3DimensionalMatching** or **X3M:** Given a set S and a collection of three-element subsets of S, called *triples*, is there a sub-collection of disjoint triples that exactly cover S? This problem can be proved NP-hard by a reduction from 3Sat, because it has a 3 in it.
- **Partition:** Given a set S of n integers, are there subsets A and B such that $A \cup B = S$, $A \cap B = \varnothing$, and
$$\sum_{a \in A} a = \sum_{b \in B} b?$$
This problem can be proved NP-hard by a simple reduction from SubsetSum. Like SubsetSum, the Partition problem is only weakly NP-hard.
- **3Partition:** Given a set S of $3n$ integers, can it be partitioned into n disjoint three-element subsets, such that every subset has exactly the same sum? Despite the similar names, this problem is *very* different from Partition; sorry, I didn't make up the names. This problem can be proved NP-hard by reduction from X3M, because it has a 3 in it. Unlike Partition, the 3Partition problem is *strongly* NP-hard; it remains NP-hard even if every input number is at most n^3.

[13] Later performed off-Broadway as *Richard Karp and his 21 Assistants*, for which Karp won a well-deserved ~~Tony~~ Turing award.

[14] Surprisingly, PlanarNotAllEqual3Sat is solvable in polynomial time!

- **SetCover:** Given a collection of sets $\mathbf{S} = \{S_1, S_2, \ldots, S_m\}$, find the smallest sub-collection of S_i's that contains all the elements of $\bigcup_i S_i$. This problem is a generalization of both VertexCover and X3M.
- **HittingSet:** Given a collection of sets $\mathbf{S} = \{S_1, S_2, \ldots, S_m\}$, find the minimum number of elements of $\bigcup_i S_i$ that hit every set in $\mathbf{S}$. This problem is also a generalization of VertexCover.
- **LongestPath:** Given a non-negatively weighted graph G (either directed or undirected) and two vertices u and v, what is the longest simple path from u to v in the graph? A path is *simple* if it visits each vertex at most once. This problem is a generalization of the corresponding Hamiltonian path problem. Of course, the corresponding *shortest* path problem can be solved in polynomial time.
- **SteinerTree:** Given a weighted, undirected graph G with some of the vertices marked, what is the minimum-weight subtree of G that contains every marked vertex? If *every* vertex is marked, the minimum Steiner tree is the minimum spanning tree; if exactly two vertices are marked, the minimum Steiner tree is the shortest path between them. This problem can be proved NP-hard by reduction from VertexCover.
- **Max2Sat:** Given a Boolean formula in conjunctive normal form, with exactly *two* literals per clause, find a variable assignment that maximizes the number of clauses with at least one True literal. This problem can be proved NP-hard by reduction from 3Sat (yes, even though it doesn't have a 3 in it). The simpler decision problem 2Sat, which asks if there is an assignment that satisfies *every* clause, can actually be solved in polynomial time.
- **MaxCut:** Given an undirected graph $G = (V, E)$, find a subset $S \subset V$ that maximizes the number of edges with exactly one endpoint in S. Equivalently, find the largest bipartite subgraph of G. This problem can be proved NP-hard by reduction from Max2Sat.

In addition to these dry but useful problems, most interesting puzzles and solitaire games have been shown to be NP-hard, or to have NP-hard generalizations. (Arguably, if a game or puzzle isn't at least NP-hard, it isn't interesting!) Here are some examples you may find familiar:

- Minesweeper (from CircuitSat)[15]
- Sudoku (utlimately from 3Sat)[16]

[15]Richard Kaye. Minesweeper is NP-complete. *Mathematical Intelligencer* 22(2):9–15, 2000. But see also: Allan Scott, Ulrike Stege, and Iris van Rooij. Minesweeper may not be NP-complete but is hard nonetheless. *Mathematical Intelligencer* 33(4):5–17, 2011.

[16]Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E86-A(5):1052–1060, 2003. http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/MasterThesis.pdf.

- Tetris (from 3Partition)[17]
- Klondike, aka "Solitaire" (from 3Sat)[18]
- Pac-Man (from HamiltonianCycle)[19]
- Super Mario Brothers (from 3Sat)[20]
- Candy Crush Saga (from a variant of 3Sat)[21]
- Threes/2048 (from 3Sat, of course)[22]
- Trainyard (from DominatingSet; see Exercise 26)[23]
- Shortest $n \times n \times n$ Rubik's cube solution (from 3Sat via a special case of PlanarUndirectedHamCycle).[24]
- Cookie Clicker (from Partition or 3Partition)[25]

This list is necessarily incomplete, thanks to a limited footnote budget.[26] As of June 2019, nobody has published a proof that a generalization of Ultimate Paperclips, Line Rider, Twister, or Cards Against Humanity is NP-hard, but I'm sure it's only a matter of time.

12.14 Choosing the Right Problem

One of the most difficult steps in proving that a problem is NP-hard is choosing the best problem to reduce from. The Cook-Levin Theorem implies that if there is a reduction from *any* NP-hard problem to problem X, then there is a reduction

[17] Ron Breukelaar, Erik D. Demaine, Susan Hohenberger, Hendrik J. Hoogeboom, Walter A. Kosters, and David Liben-Nowell. Tetris is hard, even to approximate. *International Journal of Computational Geometry and Applications* 14:41–68, 2004.

[18] Luc Longpré and Pierre McKenzie. The complexity of Solitaire. *Proceedings of the 32nd International Mathematical Foundations of Computer Science*, 182–193, 2007.

[19] Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems*, 54(4):595–621, 2014. http://giovanniviglietta.com/papers/gaming2.pdf.

[20] Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo games Are (computationally) hard. *Theoretical Computer Science* 586:135–160, 2015. http://arxiv.org/abs/1203.1895.

[21] Luciano Gualà, Stefano Leucci, Emanuele Natale. Bejeweled, Candy Crush and other match-three games are (NP-)hard. *Proc. 2014 IEEE Conference on Computational Intelligence and Games*, 2014. http://arxiv.org/abs/1403.5830.

[22] Stefan Langerman and Yushi Uno. Threes!, Fives, 1024!, and 2048 are Hard. *Proc. 8th International Conference on Fun with Algorithms*, 2016. https://arxiv.org/abs/1505.04274.

[23] Matteo Almanza, Stefano Leucci, and Alessandro Panconesi. Trainyard is NP-Hard. *Proc. 8th International Conference on Fun with Algorithms*, 2016. https://arxiv.org/abs/1603.00928.

[24] Erik D. Demaine, Sarah Eisenstat, and Mikhail Rudoy. Solving the Rubik's Cube optimally is NP-complete. *Proc. 35th Symposium on Theoretical Aspects of Computer Science*, 2018. https://arxiv.org/abs/1706.06708.

[25] Erik D. Demaine, Hiro Ito, Stefan Langerman, Jayson Lynch, Mikhail Rudoy, and Kai Xiao. Cookie Clicker. Preprint, August 2018. https://arxiv.org/abs/1808.07540.

[26] See https://xkcd.com/1208/

from *every* NP-complete problem to problem X, but some problems are easier to work with than others. There's no systematic method for choosing the right problem, but here are a few useful rules of thumb.

- If the problem asks how to assign bits to objects, or to choose a subset of objects, or to partition objects into two different subsets, try reducing from some version of Sat or Partition.
- If the problem asks how to assign labels to objects from a small fixed set, or to partition objects into a small number of subsets, try reducing from *k*Color or even 3Color.
- If the problem asks to arrange a set of objects in a particular order, try reducing from DirectedHamCycle or DirectedHamPath or Traveling-Salesman.
- If the problem asks to find a *small* subset satisfying some constraints, try reducing from MinVertexCover.
- If the problem asks to find a *large* subset satisfying some constraints, try reducing from MaxIndSet or MaxClique or Max2Sat.
- If the problem asks to partition objects into a large number of small subsets, try reducing from 3Partition.
- If the number 3 appears naturally in the problem, try 3Sat or 3Color or X3M or 3Partition. (No, this is not a joke.)
- If all else fails, try 3Sat or even CircuitSat!

I do not recommend trying to reduce from Tetris, SuperMarioBros, or Trainyard. You really want to choose a starting problem that is as simple as possible, while still capturing *some* feature of your problem that makes it difficult to solve.

12.15 A Frivolous Real-World Example

Draughts is a family of board games that have been played for thousands of years. Most Americans are familiar with the version called *checkers* or *English draughts*, but the most common variant worldwide, known as ***international draughts*** or ***Polish draughts***, originated in the Netherlands in the 16th century. For a complete set of rules, the reader should consult Wikipedia; here a few important differences from the Anglo-American game:

- **Flying kings:** As in checkers, a piece that ends a move in the row closest to the opponent becomes a *king* and gains the ability to move backward. Unlike in checkers, however, a king in international draughts can move any distance along a diagonal line in a single turn, as long as the intermediate squares are empty or contain exactly one opposing piece (which is captured).

- **Forced maximum capture:** In each turn, the moving player must capture as many opposing pieces as possible. This is distinct from the forced-capture rule in checkers, which requires only that each player must capture if possible, and that a capturing move ends only when the moving piece cannot capture further. In other words, checkers requires capturing a *locally maximal* set of opposing pieces on each turn; whereas, international draughts requires a *globally maximum* capture.
- **Capture subtleties:** As in checkers, captured pieces are removed from the board only at the end of the turn. Any piece can be captured at most once. Thus, when an opposing piece is jumped, that piece remains on the board *but cannot be jumped again* until the end of the turn.

For example, in the first position shown below, each circle represents a piece, and doubled circles represent kings. Black *must* make the first indicated move, capturing five white pieces, because it is not possible to capture more than five pieces, and there is no other move that captures five. Black cannot extend his capture further, either northeast or southeast, because the captured White pieces remain on the board until his turn is over. Then White *must* make the second indicated move, thereby winning the game.

Figure 12.21. Two forced(!) moves in international draughts; doubled circles are kings.

The actual game, which is played on a 10×10 board with 20 pieces of each color, is computationally trivial; we can precompute the optimal move for both players in every possible board configuration and hard-code the results into a lookup table of constant size. Sure, it's a *big* constant, but it's still a constant!

But consider the natural generalization of international draughts to an $n \times n$ board. In this setting, ***finding a legal move is actually NP-hard!*** The following reduction from the Hamiltonian cycle problem in directed graphs was discovered by Bob Hearn in 2010.[27] In most two-player games, finding the *best* move is NP-hard (or worse). This is the only example I know of a game—and moreover a real game played by millions of people for centuries—where *merely following the rules* is NP-hard!

Given an undirected graph G with n vertices, we construct a board configuration for international draughts, such that White can capture a certain

[27] See Theoretical Computer Science Stack Exchange: http://cstheory.stackexchange.com/a/1999/111.

number of black pieces in a single move if and only if G has a Hamiltonian cycle. We treat G as a directed graph, with two arcs $u \to v$ and $v \to u$ in place of each undirected edge uv. Number the vertices arbitrarily from 1 to n. The final draughts configuration has several gadgets:

- The vertices of G are represented by rabbit-shaped *vertex gadgets*, which are evenly spaced along a horizontal line. Each arc $i \to j$ is represented by a path of two diagonal line segments from the "left ear" of vertex gadget i to the "right ear" of vertex gadget j. The path for arc $i \to j$ is located above the vertex gadgets if $i < j$, and below the vertex gadgets if $i > j$.

Figure 12.22. A high-level overview of the reduction from Hamiltonian cycle to international draughts.

- The bulk of each vertex gadget is a diamond-shaped region called a *vault*. The walls of the vault are composed of two solid layers of black pieces, which cannot be captured; these pieces are drawn as gray circles in the figures. There are N capturable black pieces inside each vault, for some large integer N to be determined later. A white king can enter the vault through the "right ear", capture every internal piece, and then exit through the "left ear". Both ears are hallways, again with walls two pieces thick, with gaps where the arc paths end to allow the white king to enter and leave. The lengths of the "ears" can easily be adjusted to align with the other gadgets.
- For each arc $i \to j$, we have a *corner gadget*, which allows a white king leaving vertex gadget i to be redirected to vertex gadget j.
- Finally, wherever two arc paths cross, we have a *crossing gadget*; these gadgets allow the white king to traverse either arc path, but forbid switching from one arc path to the other.

A single white king starts at the bottom corner of one of the vaults. In any legal move, this king must alternate between traversing entire arc paths and clearing vaults. The king can traverse the various gadgets backward, entering

Figure 12.23. Left: A vertex gadget with three entrances and three exits. Right: A white king emptying the vault. Gray circles are black pieces that cannot be captured.

Figure 12.24. Left: One of two paths through a corner gadget. Right: One of two paths through a crossing gadget.

each vault through the exit and vice versa. But the reversal of a Hamiltonian cycle in G is another Hamiltonian cycle in G, so walking backward is fine.

If there is a Hamiltonian cycle in G, the white king can capture at least nN black pieces by visiting each of the other vaults and returning to the starting vault. On the other hand, if there is no Hamiltonian cycle in G, the white king can capture at most half of the pieces in the starting vault, and thus can capture at most $(n-1/2)N + O(n^3)$ enemy pieces altogether. The $O(n^3)$ term accounts for the corner and crossing gadgets; each edge passes through one corner gadget and at most $n^2/2$ crossing gadgets.

To complete the reduction, we set $N = n^4$. Summing up, we obtain an $O(n^5) \times O(n^5)$ board configuration, with $O(n^5)$ black pieces and one white king. We can clearly construct this board configuration by brute force in polynomial time. Figure 12.25 shows a complete example of the construction.

It is still open whether the following related question is NP-hard: Given an $n \times n$ board configuration for international draughts, can (and therefore *must*) White capture *all* the black pieces (thereby winning the game) in a single turn?

Figure 12.25. The final draughts configuration for the 4-vertex graph in Figure 12.22. (The green arrows are not actually part of the configuration.)

♥12.16 On Beyond Zebra

P and NP are only the first two steps in an enormous hierarchy of complexity classes. To close this chapter (and the book), let me describe a few more classes of interest.

Polynomial Space

PSPACE is the set of decision problems that can be solved using polynomial *space*. Every problem in NP (and therefore in P) is also in PSPACE. It is generally believed that NP $\neq$ PSPACE, but nobody can even prove that P $\neq$ PSPACE. A problem Π is ***PSPACE-hard*** if, for any problem Π' that can be solved using polynomial *space*, there is a polynomial-*time* many-one reduction from Π' to Π. If any PSPACE-hard problem is in NP, then PSPACE=NP; similarly, if any PSPACE-hard problem is in P, then PSPACE=P.

The canonical PSPACE-hard problem is the *quantified boolean formula* problem, or ***QBF***: Given a boolean formula Φ that may include any number of universal or existential quantifiers, but no free variables, is Φ equivalent to TRUE? For example, the following expression is a valid input to QBF:

$$\exists a\colon \forall b\colon \exists c\colon (\forall d\colon a \vee b \vee c \vee \bar{d}) \Leftrightarrow ((b \wedge \bar{c}) \vee (\exists e\colon \overline{(\bar{a} \Rightarrow e)} \vee (c \neq a \wedge e))).$$

SAT is equivalent to the special case of QBF where the input formula contains only existential quantifiers ($\exists$). QBF remains PSPACE-hard even when the input formula must have all its quantifiers at the beginning, the quantifiers strictly alternate between $\exists$ and $\forall$, and the quantified proposition is in conjunctive normal form, with exactly three literals in each clause, for example:

$$\exists a\colon \forall b\colon \exists c\colon \forall d\colon \big((a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})\big)$$

This restricted version of QBF can also be phrased as a two-player strategy question. Suppose two players, Alice and Bob, are given a 3CNF predicate with free variables $x_1, x_2, \ldots, x_n$. The players alternately assign values to the variables in order by index—Alice assigns a value to x_1, Bob assigns a value to x_2, and so on. Alice eventually assigns values to every variable with an odd index, and Bob eventually assigns values to every variable with an even index. Alice wants to make the expression TRUE, and Bob wants to make it FALSE. Assuming Alice and Bob play perfectly, who wins this game? Not surprisingly, most two-player games[28] like tic-tac-toe, reversi, checkers, go, chess, and mancala—or more accurately, appropriate generalizations of these constant-size games to arbitrary board sizes—are PSPACE-hard.

Another canonical PSPACE-hard problem is *NFA totality*: Given a non-deterministic finite-state automaton M over some alphabet Σ, does M accept every string in Σ^*? The closely related problems *NFA equivalence* (Do two given NFAs accept the same language?) and *NFA minimization* (Find the smallest NFA that accepts the same language as a given NFA) are also PSPACE-hard, as are the corresponding questions about regular expressions. (The corresponding questions about *deterministic* finite-state automata are actually solvable in polynomial time.)

Exponential Time

The next significantly larger complexity class, ***EXP*** (also called EXPTIME), is the set of decision problems that can be solved in exponential time, that is, using at most 2^{n^c} steps for some constant $c > 0$. Every problem in PSPACE (and

[28]For a good (but inevitably aging) overview of known results on the computational complexity of games and puzzles, see Erik Demaine and Bob Hearn's monograph *Games, Puzzles, and Computation* (CRC Press, 2009).

therefore in NP (and therefore in P)) is also in EXP. It is generally believed that PSPACE $\subsetneq$ EXP, but nobody can even prove that NP $\neq$ EXP. A problem Π is ***EXP-hard*** if, for any problem Π' that can be solved in *exponential* time, there is a *polynomial*-time many-one reduction from Π' to Π. If any EXP-hard problem is in PSPACE, then EXP=PSPACE; similarly, if any EXP-hard problem is in NP, then EXP=NP. We *do* know that P $\neq$ EXP; in particular, no EXP-hard problem is in P.

Natural generalizations of many interesting 2-player games—like checkers, chess, mancala, and go—are actually EXP-hard. The boundary between PSPACE-complete games and EXP-hard games is rather subtle. For example, there are three ways to draw in chess (the standard 8×8 game): stalemate (the player to move is not in check but has no legal moves), repeating the same board position three times, or moving fifty times without capturing or moving a pawn. The $n \times n$ generalization of chess is either in PSPACE or EXP-hard depending on how we generalize these rules. If we declare a draw after (say) n^3 capture-free moves, then every game must end after a polynomial number of moves, so we can simulate all possible games from any given position using only polynomial space. On the other hand, if we ignore the capture-free move rule entirely, the resulting game can last an exponential number of moves, so there no obvious way to detect a repeating position using only polynomial space; indeed, this version of $n \times n$ chess is EXP-hard.

Excelsior!

Naturally, even exponential time is not the end of the story. ***NEXP*** is the class of decision problems that can be solve in *nondeterministic* exponential time; equivalently, a decision problem is in NEXP if and only if, for every YES instance, there is a *proof* of this fact that can be checked in exponential time. ***EXPSPACE*** is the set of decision problems that can be solved using exponential *space*. Even these larger complexity classes have hard problems; for example, if we add the intersection operator $\cap$ to the syntax of regular expressions, deciding whether two such expressions describe the same language is EXPSPACE-hard. Beyond EXPSPACE are complexity classes with *doubly*-exponential resource bounds (EEXP, NEEXP, and EEXPSPACE), then *triply* exponential resource bounds (EEEXP, NEEEXP, and EEEXPSPACE), and so on ad infinitum.

All these complexity classes can be ordered by inclusion:

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXP} \subseteq \mathrm{NEXP} \subseteq \mathrm{EXPSPACE} \subseteq \mathrm{EEXP} \subseteq \mathrm{NEEXP} \subseteq \cdots$$

Most complexity theorists strongly believe that every inclusion in this sequence is strict; that is, no two of these complexity classes are equal. However, the strongest result that has been proved is that every class in this sequence is strictly contained in the class *three* steps later in the sequence. For example, we have

proofs that P $\neq$ EXP and PSPACE $\neq$ EXPSPACE, but not whether P $\neq$ PSPACE or NP $\neq$ EXP.

The limit of this series of increasingly exponential complexity classes is the class ***ELEMENTARY*** of decision problems that can be solved using time or space bounded by a function the form $2 \uparrow^k n$ for some constant integer k, where

$$2 \uparrow^k n := \begin{cases} n & \text{if } k = 0, \\ 2^{2\uparrow^{k-1} n} & \text{otherwise.} \end{cases}$$

For example, $2 \uparrow^1 n = 2^n$ and $2 \uparrow^2 n = 2^{2^n}$.

It may be tempting to conjecture that *every* natural decidable problem can be solved in elementary time, but in fact this conjecture is incorrect. Consider the ***generalized regular expressions*** defined by recursively combining (possibly empty) strings over some finite alphabet by concatenation (xy), union ($x + y$), Kleene closure (x^*), and negation ($\overline{x}$). For example, the generalized regular expression $\overline{(0+1)^*00(0+1)^*}$ represents the set of strings in $\{0, 1\}^*$ that do *not* contain two 0s in a row. It is possible to determine algorithmically whether two generalized regular expressions describe identical languages, by recursively converting each expression into an equivalent NFA, converting each NFA into a DFA, and then minimizing the DFA. However, the running time of this algorithm has the non-elementary bound $2 \uparrow^{\Theta(n)} 2$, intuitively because each layer of recursive negation can exponentially increase the number of states. In fact, Larry Stockmeyer proved in 1974 that this problem *cannot* be solved in merely elementary time, even if we forbid Kleene closure.

Exercises

1. (a) Describe and analyze and algorithm to solve Partition in time $O(nM)$, where n is the size of the input set and M is the sum of the absolute values of its elements.

   (b) Why doesn't this algorithm imply that P=NP?

2. Consider the following problem, called BoxDepth: Given a set of n axis-aligned rectangles in the plane, how big is the largest subset of these rectangles that contain a common point?

   (a) Describe a polynomial-time reduction from BoxDepth to MaxClique.

   (b) Describe and analyze a polynomial-time algorithm for BoxDepth. *[Hint: $O(n^3)$ time should be easy, but $O(n \log n)$ time is possible.]*

   (c) Why don't these two results imply that P=NP?

3. A boolean formula is in *disjunctive normal form* (or *DNF*) if it consists of a *disjunction* (OR) or several *terms*, each of which is the conjunction (AND) of one or more literals. For example, the formula

$$(\overline{x} \wedge y \wedge \overline{z}) \vee (y \wedge z) \vee (x \wedge \overline{y} \wedge \overline{z})$$

is in disjunctive normal form. DNF-SAT asks, given a boolean formula in disjunctive normal form, whether that formula is satisfiable.

(a) Describe a polynomial-time algorithm to solve DNF-SAT.

(b) What is the error in the following argument that P=NP?

> Suppose we are given a boolean formula in conjunctive normal form with at most three literals per clause, and we want to know if it is satisfiable. We can use the distributive law to construct an equivalent formula in disjunctive normal form. For example,
>
> $$(x \vee y \vee \overline{z}) \wedge (\overline{x} \vee \overline{y}) \iff (x \wedge \overline{y}) \vee (y \wedge \overline{x}) \vee (\overline{z} \wedge \overline{x}) \vee (\overline{z} \wedge \overline{y})$$
>
> Now we can use the algorithm from part (a) to determine, in polynomial time, whether the resulting DNF formula is satisfiable. We have just solved 3SAT in polynomial time. Since 3SAT is NP-hard, we must conclude that P=NP!

4. The problem ALLORNOTHING3SAT asks, given a 3CNF boolean formula, whether there is an assignment to the variables such that each clause either has three TRUE literals or has three FALSE literals.

(a) Describe a polynomial-time algorithm to solve ALLORNOTHING3SAT.

(b) But 3SAT is NP-hard! Why doesn't the existence of this algorithm prove that P=NP?

5. (a) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary weighted graph G, the length of the shortest Hamiltonian cycle in G. Describe and analyze a **polynomial-time** algorithm that computes, given an arbitrary weighted graph G, the shortest Hamiltonian cycle in G, using this magic black box as a subroutine.

(b) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary graph G, the number of vertices in the largest complete subgraph of G. Describe and analyze a **polynomial-time** algorithm that computes, given an arbitrary graph G, a complete subgraph of G of maximum size, using this magic black box as a subroutine.

(c) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary graph G, whether G is 3-colorable. Describe and analyze a **polynomial-time** algorithm that either computes a proper 3-coloring of a given graph or correctly reports that no such coloring exists, using the magic black box as a subroutine. *[Hint: The input to the magic black box is a graph. Only a graph. Vertices and edges. Nothing else.]*

(d) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary boolean formula Φ, whether Φ is satisfiable. Describe and analyze a **polynomial-time** algorithm that either computes a satisfying assignment for a given boolean formula or correctly reports that no such assignment exists, using the magic black box as a subroutine.

(e) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary set X of positive integers, whether X can be partitioned into two sets A and B such that $\sum A = \sum B$. Describe and analyze a **polynomial-time** algorithm that either computes an equal partition of a given set of positive integers or correctly reports that no such partition exists, using the magic black box as a subroutine.

♣♥(f) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary generalized regular expression R (as defined just before the Exercises), whether R matches any string. Describe and analyze a **polynomial-time** algorithm that either finds a single string that matches a given generalized regular expression or correctly reports that no such string exists, using the magic black box as a subroutine.

6. **There's something special about the number 3.**

(a) Describe and analyze a polynomial-time algorithm for 2PARTITION. Given a set S of $2n$ positive integers, your algorithm will determine in polynomial time whether the elements of S can be split into n disjoint pairs whose sums are all equal.

(b) Describe and analyze a polynomial-time algorithm for 2COLOR. Given an undirected graph G, your algorithm will determine in polynomial time whether G has a proper coloring that uses only two colors.

(c) Describe and analyze a polynomial-time algorithm for 2SAT. Given a boolean formula Φ in conjunctive normal form, with exactly *two* literals per clause, your algorithm will determine in polynomial time whether Φ has a satisfying assignment. *[Hint: This problem is strongly connected to topics described in an earlier chapter.]*

7. **There's nothing special about the number 3.**
   (a) The problem 12Partition is defined as follows: Given a set S of $12n$ positive integers, determine whether the elements of S can be split into n subsets, each with 12 elements, whose sums are all equal. Prove that 12Partition is NP-hard. *[Hint: Reduce from 3Partition. It may be easier to consider multisets first.]*
   (b) The problem 12Color is defined as follows: Given an undirected graph G, determine whether we can color each vertex with one of twelve colors, so that every edge touches two different colors. Prove that 12Color is NP-hard. *[Hint: Reduce from 3Color.]*
   (c) The problem 12SAT is defined as follows: Given a boolean formula Φ in conjunctive normal form, with exactly twelve literals per clause, determine whether Φ has a satisfying assignment. Prove that 12Sat is NP-hard. *[Hint: Reduce from 3Sat.]*

8. There are two different versions of the Hamiltonian cycle problem, one for directed graphs and one for undirected graphs. Earlier in this chapter you can find two proofs that the *directed* Hamiltonian cycle problem is NP-hard.
   (a) Describe a polynomial-time reduction from the *undirected* Hamiltonian cycle problem to the *directed* Hamiltonian cycle problem. Prove your reduction is correct.
   (b) Describe a polynomial-time reduction from the *directed* Hamiltonian cycle problem to the *undirected* Hamiltonian cycle problem. Prove your reduction is correct.
   (c) Which of these two reductions implies that the *undirected* Hamiltonian cycle problem is NP-hard?

9. (a) Describe a polynomial-time reduction from UndirectedHamiltonianCycle to DirectedHamiltonianCycle.
   (b) Describe a polynomial-time reduction from DirectedHamiltonianCycle to UndirectedHamiltonianCycle.

10. (a) Describe a polynomial-time reduction from the HamiltonianPath problem to HamiltonianCycle.
    (b) Describe a polynomial-time reduction from the HamiltonianCycle problem to HamiltonianPath. *[Hint: A polynomial-time reduction can call the black-box subroutine more than once, but it doesn't have to.]*

11. Consider the following subtle variants of CNFSat. For each problem, the input is a boolean formula Φ in conjunctive normal form, and the goal is to determine whether Φ has a satisfying assignment.

(a) Suppose every clause of Φ contains ***at most*** three literals and each variable appears in at most ***three*** clauses. Prove that this variant of CNFSat is NP-hard.

(b) Suppose every clause of Φ contains ***exactly*** three literals and each variable appears in at most ***four*** clauses. Prove that this variant of 3Sat is NP-hard. *[Hint: Solve part (a) first.]*

♥(c) Suppose every clause of Φ can contain *any number of* literals, but each variable appears in at most ***two*** clauses. Describe a polynomial-time algorithm for this variant of CNFSat.

♥(d) Suppose every clause of Φ contains ***exactly*** three literals and each variable appears in at most ***three*** clauses. Prove that Φ must be satisfiable. (So this variant of 3Sat is completely trivial!)

12. (a) Prove that NotAllEqual3Sat is NP-hard.

(b) Prove that 1-in-3Sat is NP-hard.

13. A boolean formula in *exclusive-or conjunctive normal form* (XCNF) is a conjunction (And) of several *clauses*, each of which is the *exclusive*-or of several literals; that is, a clause is true if and only if it contains an odd number of true literals. The XCNF-Sat problem asks whether a given XCNF formula is satisfiable. Either describe a polynomial-time algorithm for XCNF-Sat or prove that XCNF-Sat is NP-hard. *[Hint: Do not try to do both.]*

♥14. Consider the following variant of 3Sat, called Majority3Sat. Just like 3Sat, the input to Majority3Sat is a boolean formula Φ in conjunctive normal form, with exactly three literals er clause. Majority3Sat asks whether there is an assignment to the variables of Φ, such that every clause contains *at least two* True literals.

Either describe an algorithm that solves Majority3Sat in polynomial time or prove that Majority3Sat is NP-hard. *[Hint: Do not try to do both.]*

♠♥15. For any subset $X \subseteq \{0, 1, 2, 3\}$, consider the following problem, which I'll call ***X-3Sat***. The input is a boolean formula Φ in conjunctive normal form, with exactly three literals in each clause. The problem is to decide whether there is an assignment to the variables of Φ such that in each clause of Φ, the number of True literals is in the set X. For example:

- $\{1, 2, 3\}$-3Sat is the standard 3Sat problem.
- $\{0, 3\}$-3Sat is the same as AllOrNothing3Sat. (See Exercise 4.)
- $\{1, 2\}$-3Sat is usually called NotAllEqual3Sat. (See Exercise 12(a).)
- $\{1\}$-3Sat is usually called 1-in-3Sat. (See Exercise 12(b).)

- $\{1,3\}$-3Sat is usually called XCNF-3Sat. (See Exercise 13.)
- $\{2,3\}$-3Sat is usually called Majority3Sat. (See Exercise 14.)

Give a complete list of all subsets $X \subseteq \{0,1,2,3\}$ such that X-3Sat is solvable in polynomial time, assuming P $\neq$ NP. *[Hint: Don't give* 16 *different arguments.]*

16. Prove that the following problems are NP-hard.
    (a) Given an undirected graph G, does G contain a simple path that visits all but 17 vertices?
    (b) Given an undirected graph G, does G have a spanning tree in which every node has degree at most 23?
    (c) Given an undirected graph G, does G have a spanning tree with at most 42 leaves?
    (d) Given an undirected graph $G = (V, E)$, what is the size of the largest subset of vertices $S \subseteq V$ such that at most 374 edges in E have both endpoints in S?
    (e) Given an undirected graph $G = (V, E)$, what is the size of the largest subset of vertices $S \subseteq V$ such that each vertex in S has at most 473 neighbors in S?
    (f) Given an undirected graph G, is it possible to color the vertices of G with three different colors, so that at most 31337 edges have both endpoints the same color?

17. Prove that the following variants of the minimum spanning tree problem are NP-hard.
    (a) Given a graph G, compute the *maximum-diameter* spanning tree of G. (The diameter of a tree T is the length of the longest path in T.)
    (b) Given a graph G with weighted edges, compute the minimum-weight *depth-first* spanning tree of G.
    (c) Given a graph G with weighted edges and a subset S of vertices of G, compute the minimum-weight spanning tree all of whose leaves are in S.
    (d) Given a graph G with weighted edges and an integer ℓ, compute the minimum-weight spanning tree with at most ℓ leaves.
    (e) Given a graph G with weighted edges and an integer Δ, compute the minimum-weight spanning tree where every node has degree at most Δ.

18. (a) Using the gadget in Figure 12.26(a), prove that deciding whether a given *planar* graph is 3-colorable is NP-hard. *[Hint: Show that the gadget can be 3-colored, and then replace any crossings in a planar embedding with the gadget appropriately.]*

(b) Using part (a) and the gadget in Figure 12.26(b), prove that deciding whether a planar graph *with maximum degree 4* is 3-colorable is NP-hard. *[Hint: Replace any vertex with degree greater than 4 with a collection of gadgets connected so that no degree is greater than four.]*

(a) (b)

Figure 12.26. (a) Gadget for planar 3-colorability. (b) Gadget for degree-4 planar 3-colorability.

19. Prove that PlanarCircuitSat is NP-hard. *[Hint: Construct a gadget for crossing wires.]*

20. (a) Describe a polynomial-time reduction from 3Sat to 4Sat.
    (b) Describe a polynomial-time reduction from 4Sat to 3Sat.

♥21. Describe a direct polynomial-time reduction from 4Color to 3Color. (This is a lot harder than the opposite direction.)

22. A *domino* is a 1×2 rectangle divided into two squares, each of which is labeled with an integer.[29] In a *legal arrangement* of dominos, the dominos are lined up end-to-end so that the numbers on adjacent ends match.

Figure 12.27. A legal arrangement of dominos in which every integer between 0 and 6 appears twice

For each of the following problems, either describe a polynomial-time algorithm or prove that the problem is NP-hard:

(a) Given an arbitrary bag D of dominos, is there a legal arrangement of *all* the dominos in D?

(b) Given an arbitrary bag D of dominos, is there a legal arrangement of a dominos from D in which every integer between 1 and n appears exactly twice?

[29] These integers are usually represented by pips, exactly like dice. On a standard domino, the number of pips on each side is between 0 and 6, although one can buy sets with up to 9 or even 12 pips on each side; we will allow arbitrary integer labels. A standard set of dominos contains exactly one domino for each possible unordered pair of labels; we do *not* assume that the inputs to our problems have this property.

♥(c) Given an arbitrary bag D of dominos, what is the largest number of dominos we can take from D to make a legal arrangement?

23. *Pebbling* is a solitaire game played on an undirected graph G, where each vertex has zero or more *pebbles*. A single *pebbling move* consists of removing two pebbles from a vertex v and adding one pebble to an arbitrary neighbor of v. (Obviously, the vertex v must have at least two pebbles before the move.) The PEBBLEDESTRUCTION problem asks, given a graph $G = (V, E)$ and a pebble count $p(v)$ for each vertex v, whether is there a sequence of pebbling moves that removes all but one pebble. Prove that PEBBLEDESTRUCTION is NP-hard.

24. Recall that a 5-coloring of a graph G is a function that assigns each vertex of G a "color" from the set $\{0, 1, 2, 3, 4\}$, such that for any edge uv, vertices u and v are assigned different "colors". A 5-coloring is *careful* if the colors assigned to adjacent vertices are not only distinct, but differ by more than 1 (mod 5). Prove that deciding whether a given graph has a careful 5-coloring is NP-hard. *[Hint: Reduce from the standard 5COLOR problem.]*

Figure 12.28. A careful 5-coloring.

25. (a) A subset S of vertices in an undirected graph G is *half-independent* if each vertex in S is adjacent to *at most one* other vertex in S. Prove that finding the size of the largest half-independent set of vertices in a given undirected graph is NP-hard.

(b) A subset S of vertices in an undirected graph G is *sort-of-independent* if if each vertex in S is adjacent to *at most 374* other vertices in S. Prove that finding the size of the largest sort-of-independent set of vertices in a given undirected graph is NP-hard.

(c) A subset S of vertices in an undirected graph G is *almost independent* if at most 374 edges in G have both endpoints in S. Prove that finding the size of the largest almost-independent set of vertices in a given undirected graph is NP-hard.

26. Let $G = (V, E)$ be a graph. A *dominating set* in G is a subset S of the vertices such that every vertex in G is either in S or adjacent to a vertex in S. The

DominatingSet problem asks, given a graph G and an integer k as input, whether G contains a dominating set of size k. Prove that this problem is NP-hard.

Figure 12.29. A dominating set of size 3 in the Petersen graph.

27. A subset S of vertices in an undirected graph G is *triangle-free* if, for every triple of vertices $u, v, w \in S$, at least one of the three edges uv, uw, vw is *absent* from G. Prove that finding the size of the largest triangle-free subset of vertices in a given undirected graph is NP-hard.

Figure 12.30. A triangle-free subset of 7 vertices. This is not the largest such subset in this graph.

28. The RectangleTiling problem is defined as follows: Given one large rectangle and several smaller rectangles, determine whether the smaller rectangles can be placed inside the large rectangle with no gaps or overlaps.
    (a) Prove that RectangleTiling is NP-hard.
    (b) Prove that RectangleTiling is *strongly* NP-hard.

Figure 12.31. A positive instance of the RectangleTiling problem.

29. (a) A subset B of vertices in a graph G is a ***Burr set*** if removing every vertex in B from G leaves a subgraph that does not contain a Hamiltonian path. Prove that finding the smallest Burr set in a given graph is NP-hard.
    (b) A subset S of vertices in a graph G is a ***Schuyler set*** if removing every vertex in S from G leaves a subgraph that *does* contain a Hamiltonian

path. Prove that finding the smallest Schuyler set in a given graph is NP-hard.

30. (a) A *tonian path* in a graph G is a path that goes through at least half of the vertices of G. Show that determining whether a graph has a tonian path is NP-hard.

    (b) A *tonian cycle* in a graph G is a cycle that goes through at least half of the vertices of G. Show that determining whether a graph has a tonian cycle is NP-hard. *[Hint: Use part (a). Or not.]*

31. Let G be an undirected graph with weighted edges. A *heavy Hamiltonian cycle* is a cycle C that passes through each vertex of G exactly once, such that the total weight of the edges in C is more than half of the total weight of all edges in G. Prove that deciding whether a graph has a heavy Hamiltonian cycle is NP-hard.

Figure 12.32. A heavy Hamiltonian cycle. The cycle has total weight 34; the graph has total weight 67.

32. For each of the following problems, either describe a polynomial-time algorithm or prove that the problem is NP-hard.

    (a) A *double-Eulerian tour* in an undirected graph G is a closed walk that traverses every edge in G exactly twice. Given a graph G, does G have a double-Eulerian tour?

    (b) A *double-Hamiltonian tour* in an undirected graph G is a closed walk that visits every vertex in G exactly twice. Given a graph G, does G have a double-Hamiltonian tour?

    (c) A *double-Hamiltonian* ***circuit*** in an undirected graph G is a closed walk that visits every vertex in G exactly twice *and traverses each edge in G at most once*. Given a graph G, does G have a double-Hamiltonian circuit?

    (d) A *triple-Eulerian tour* in an undirected graph G is a closed walk that traverses every edge in G exactly three times. Given a graph G, does G have a triple-Eulerian tour?

(e) A *triple-Hamiltonian tour* in an undirected graph G is a closed walk that visits every vertex in G exactly three times. Given a graph G, does G have a triple-Hamiltonian tour?

33. This exercise asks you to prove that a certain reduction from VertexCover to SteinerTree is correct. Suppose we want to find the smallest vertex cover in a given undirected graph $G = (V, E)$. We construct a new graph $H = (V', E')$ as follows:

- $V' = V \cup E \cup \{z\}$
- $E' = \{ve \mid v \in V \text{ is an endpoint of } e \in E\} \cup \{vz \mid v \in V\}$.

Equivalently, we construct H by subdividing each edge in G with a new vertex, and then connecting all the original vertices of G to a new *apex* vertex z.

Prove that G has a vertex cover of size k if and only if there is a subtree of H with $k + |E| + 1$ vertices that contains every vertex in $E \cup \{z\}$.

34. Consider the following solitaire game. The puzzle consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions: (1) every row contains at least one stone, and (2) no column contains stones of both colors. For some initial configurations of stones, reaching this goal is impossible.

A solvable puzzle and one of its many solutions. An unsolvable puzzle.

Prove that it is NP-hard to determine, given an initial configuration of red and blue stones, whether the puzzle can be solved.

35. Each of the following games involves an $n \times m$ grid of squares, where each square is either empty or occupied by a stone. In a single move, you can remove *all* the stones in an arbitrary column.

(a) Prove that it is NP-hard to find the *smallest* subset of columns that can be cleared so that *at most* one stone remains in each row of the grid.

(b) Prove that it is NP-hard to find the *largest* subset of columns that can be cleared so that *at least* one stone remains in each row of the grid.

♥(c) Prove that it is NP-hard to determine whether *any* subset of columns can be cleared so that *exactly* one stone remains in each row of the grid.

36. Jeff tries to make his students happy. At the beginning of class, he passes out a questionnaire that lists a number of possible course policies in areas where he is flexible. Every student is asked to respond to each possible course policy with one of "strongly favor", "mostly neutral", or "strongly oppose". Each student may respond with "strongly favor" or "strongly oppose" to at most five questions. Because Jeff's students are very understanding, each student is happy if (but only if) he or she prevails in at least one of their strong policy preferences. Either describe a polynomial-time algorithm for setting course policy to maximize the number of happy students, or show that the problem is NP-hard.

37. You're in charge of choreographing a musical for your local community theater, and it's time to figure out the final pose of the big show-stopping number at the end. ("Streetcar!") You've decided that each of the n cast members in the show will be positioned in a big line when the song finishes, all with their arms extended and showing off their best spirit fingers.

    The director has declared that during the final flourish, each cast member must either point both their arms up or point both their arms down; it's your job to figure out who points up and who points down. Moreover, the director has also given you a list of arrangements that will upset his delicate artistic temperament. Each forbidden arrangement is a subset of the cast members paired with arm positions; for example: "Marge may not point her arms up while Ned, Apu, and Smithers point their arms down."

    Prove that finding an acceptable arrangement of arm positions is NP-hard.

38. The next time you are at a party, one of the guests will suggest everyone play a round of Three-Way Mumbletypeg, a game of skill and dexterity that requires three teams and a knife. The official Rules of Three-Way Mumbletypeg (fixed during the Holy Roman Three-Way Mumbletypeg Council in 1625) require that (1) each team *must* have at least one person, (2) any two people on the same team *must* know each other, and (3) everyone watching the game *must* be on one of the three teams. Of course, it will be a really *fun* party; nobody will want to leave. There will be several pairs of people at the party who don't know each other. The host of the party, having heard thrilling tales of your prowess in all things algorithmic, will hand you a list of which pairs of party-goers know each other and ask you to choose the teams, while he sharpens the knife.

Either describe and analyze a polynomial time algorithm to determine whether the party-goers can be split into three legal Three-Way Mumbletypeg teams, or prove that the problem is NP-hard.

39. The party you are attending is going great, but now it's time to line up for *The Algorithm March (アルゴリズムこうしん)*! This dance was originally developed by the Japanese comedy duo Itsumo Kokokara (いつもここから) for the children's television show PythagoraSwitch (ピタゴラスイッチ). The Algorithm March is performed by a line of people; each person in line starts a specific sequence of movements one measure later than the person directly in front of them. Thus, the march is the dance equivalent of a musical round or canon, like "Row Row Row Your Boat" or "Frère Jacques".

    Proper etiquette dictates that each marcher must know the person directly in front of them in line, lest a minor mistake lead to horrible embarrassment between strangers. Suppose you are given a complete list of which people at your party know each other. Prove that it is NP-hard to determine the largest number of party-goers that can participate in the Algorithm March. You may assume without loss of generality that there are no ninjas at your party.

♣40. Prove that the following problems about nondeterministic finite-state automata and regular expressions are NP-hard:

(a) Given an NFA M over the alphabet $\Sigma = \{0, 1\}$, is there a string in Σ^* that M does *not* accept?

(b) Given an *acyclic* NFA M over the alphabet $\Sigma = \{0, 1\}$, what is the length of the *shortest* string in Σ^* that M does *not* accept?

(c) Given a regular expression R over the alphabet $\Sigma = \{0, 1\}$, is there a string in Σ^* that R does *not* match?

(d) Given a *star-free* regular expression R over the alphabet $\Sigma = \{0, 1\}$, what is the length of the *shortest* string in Σ^* that R does *not* match?

(In fact, problems (a) and (c) are PSPACE-complete; even proving that these problems are in PSPACE is nontrivial.)

♣41. (a) Describe a polynomial-time algorithm for the following problem: Given an NFA M over the alphabet $\Sigma = \{0, 1\}$, is there a string in Σ^* that M *does* accept?

(b) Describe a polynomial-time algorithm for the following problem: Given a regular expression R over the alphabet $\Sigma = \{0, 1\}$, is there a string in Σ^* that R *does* match?

(c) The complement of any regular language is another regular language. So why don't these two algorithms, together with the NP-hardness results in Problem 40, prove that P=NP?

42. Charon needs to ferry n recently deceased people across the river Acheron into Hades. Certain pairs of these people are sworn enemies, who cannot be together on either side of the river unless Charon is also present. (If two enemies are left alone, one will steal the obol from the other's mouth, leaving them to wander the banks of the Acheron as a ghost for all eternity. Let's just say this is a Very Bad Thing.) The ferry can hold at most k passengers at a time, including Charon, and only Charon can pilot the ferry.[30]

    Prove that it is NP-hard to decide whether Charon can ferry all n people across the Acheron unharmed (aside from being, you know, dead). The input for Charon's problem consists of the integers k and n and an n-vertex graph G describing the pairs of enemies. The output is either True or False.

    Please do not write your solution in classical Latin.

---

[30]This problem is a generalization of the well-known wolf-goat-and-cabbage puzzle, whose first known appearance is in the remarkable medieval manuscript *Propositiones ad Acuendos Juvenes* [*Problems to Sharpen the Young*].

> XVIII. Propositio De Homine et Capra et Lvpo.
>
> Homo quidam debebat ultra fluuium transferre lupum, capram, et fasciculum cauli. Et non potuit aliam nauem inuenire, nisi quae duos tantum ex ipsis ferre ualebat. Praeceptum itaque ei fuerat, ut omnia haec ultra illaesa omnino transferret. Dicat, qui potest, quomodo eis illaesis transire potuit?
>
> **Solutio.** Simili namque tenore ducerem prius capram et dimitterem foris lupum et caulum. Tum deinde uenirem, lupumque transferrem: lupoque foris misso capram naui receptam ultra reducerem; capramque foris missam caulum transueherem ultra; atque iterum remigassem, capramque assumptam ultra duxissem. Sicque faciendo facta erit remigatio salubris, absque uoragine lacerationis.

For those few readers whose classical Latin is a little rusty, here is an English translation:

> XVIII. The Problem of the Man, the Goat, and the Wolf.
>
> A man needed to transfer a wolf, a goat, and a bundle of cabbage across a river. However, he found that his boat could only bear the weight of two [objects at a time, including the man]. And he had to get everything across unharmed. Tell me if you can: How they were able to cross unharmed?
>
> **Solution.** In a similar fashion [as an earlier problem], I would first take the goat across and leave the wolf and cabbage on the opposite bank. Then I would take the wolf across; leaving the wolf on shore, I would retrieve the goat and bring it back again. Then I would leave the goat and take the cabbage across. And then I would row across again and get the goat. In this way the crossing would go well, without any threat of slaughter.

The most likely author of the *Propositiones* is the prolific 8th-century English scholar Alcuin of York. The evidence for Alcuin's authorship of this treatise is somewhat circumstantial; however, we do know from his correspondence with Charlemagne that he sent the emperor some "simple arithmetical problems for fun". Most modern scholars believe that even if Alcuin did write the *Propositiones*, he did not invent all of the problems himself, but rather collected them from even earlier sources.

Some things never change.

If we had an index file, we could look it up in the index file under "index file".

— Tegan Jovanka [Janet Fielding], "Castrovalva (Part 1)", *Doctor Who*, Season 19 (January 4, 1982)

I started with the phone book. Looking up "mensa" was not going to be easy, what with having to follow the strict alphabetizing rules that are so common nowadays. I prefer a softer, more fuzzy alphabetizing scheme, one that allows the mind to float free and "happen" upon the word. There is pride in that. The dictionary is a perfect example of over-alphabetization, with its harsh rules and every little word neatly in place. It almost makes me never want to eat again.

— Steve Martin, "How I Joined Mensa", *The New Yorker*, July 21, 1997.

Index

For some topics with multiple references, bold page numbers indicate the primary reference. Humans and pseudocode are indexed separately.

1-IN-3SAT, 405
2048 (game), 407
2COLOR, 417
2PARTITION, 417
2SAT, 406, 417
3CNF formula, 388
3COLOR, 395
    reduction from 3SAT, 395
3PARTITION, 405
3SAT, 388
    reduction from CIRCUITSAT, 388, 393, 395
    reduction to 3COLOR, 395
    reduction to DIRECTEDHAMCYCLE, 400
    reduction to MAXINDSET, 390, 393, 395
    rule of three, 395, 405, 408
4:20, 161

academic job market, 170
active vertex (depth-first search), 228
acyclic graph (= forest), 191
acyclic maximum flow, 339, 345

ad-hoc networks, 375
addition chains, 94
increment and double only, 181
additional recurrence parameter, 295, 313, 318
adjacency matrix, 361
adjacent vertices, 191
airline scheduling, 362
alternating path, 356
amortized analysis, 264, 267
The Announcer's Test, 17
antanairesis, *see* Euclid's algorithm
Antarctica, 123, 143, 181
APSP, *see* shortest paths, all-pairs
arbitrage, 321
arithmetic takes time, 104, 325
arpedonaptai, 7, 317
arrow notation ($a \uparrow^b c$), 415
articulation point, *see* cut vertex
artificial source vertex, 227, 312, 355, 367
Āryabhaṭa's pulverizer, *see* Euclid's algorithm
assignment, *see* matching, tuple selection
augmenting path, 332
"average case" analysis, 31

B-tree, 147
back edge (depth-first search), 229
backtracking, **71**
recursive brute force, 80
sequence of decisions, 79
summary of past decisions, 79
backward induction, *see* dynamic programming
Baguenaudier, 45
balanced brackets, 148, 184
"The Barley Mow", 18
base case, 23
baseball elimination, 363
BeAMillionaireAndNeverPayTaxes
reduction from BeAMillionaire, 10
Bellman-Ford, 289
as dynamic programming, 294
Moore's variant, 292
Bellman-Kalaba, *see* Bellman-Ford
Bellman-Shimbel, *see* Bellman-Ford
Bellman's equation, *see* recurrence
"best case" analysis, 31
best-first search, 202
Dijkstra's algorithm, 288
Jarník's algorithm, 264
widest-path algorithm (Edmonds-Karp), 340, 348
BFS, *see* breadth-first search
binary search trees, 67
AA trees, 96, 145
AVL trees, 96, 144, 341
optimal, 91
reconfiguration, 68
red-black trees, 96, 144
left-leaning, *see* AA trees
binary to decimal conversion, 56
bipartite graph, 207
bipartite maximum matching, 355
bitonic, 60
black box, 12, 21, 22, 416
see also none of your business
Bob's mama sees a ukulele, 129
bond (minimal edge cut), 347
boolean circuits, 190, 379
boolean formula, 386
boolean matrix multiplication, 323
Borůvka's algorithm, 261
advantages, 262
Boston Pool algorithm, 172
bottleneck, *see also* minimum cut
bottleneck distance, 270, 347
bottleneck spanning tree, 348
breadth-first search, 202, **278**, 341
Bridges of Königsburg, 190
see also Euler Tour

BST, *see* binary search trees
Bubba sees a banana, 128
bus scheduling, 362

c_f (residual capacity), 331
Camelot, 185
Candy Crush Saga, 407
capacity scaling, 351
careful graph coloring, 422
central vertex of a tree, 64
checkerboard, 59, 222, 371
checkers, *see* draughts
children's songs, 16
choosing the right problem to reduce from, 407
circuit satisfiability, *see* CircuitSat
CircuitSat, 380
  reduction to 3Sat, 388, 393, 395
  reduction to Sat, 386
circulation, 337
clause, 388
clause gadget, 390, 395, 396, 401
clique, 394
closed walk in a graph, 191
CNF, *see* conjunctive normal form
CNF-Sat, *see* Sat
co-NP, 381
compass and straightedge, 7
component, 191
computationes canonica et legalis, 188
condensation, *see* strong component graph
configuration graph, 194, 220, 221
conjunctive normal form, 388
connected component, *see* component
connected graph, 191
conservation constraint, 328
convenience, 32, 79, 227, 237, 279, 396
Cook reduction, 384
Cookie Clicker, 407
counting graph components, 204
cover gadget, 399
cross edge (depth-first search), 229
crossing gadget, 410
cursus publicus, 188
cut (vertex partition), 329
cut capacity $\|S, T\|$, 329
cut vertex, 245
cycle cover, 371
cycle flow, 337
cycle in a graph, 191

dag, *see* directed acyclic graph
Dance Dance Revolution, 133
data structures for graphs, *see* graphs
decision problem, 381
decision tree, 53
decision versus optimization, 85
degree of a vertex, 191
DeNile, 146
dependency graph, 107, 109, 114, 119, 193, 234, 366
depth-first order, *see* preorder, postorder
depth-first search, 76, 201, **225**, 282, 380
DFA, *see* finite-state automaton
DFS, *see* depth-first search
Dijkstra's algorithm, 203, 284
  with negative edges, 288
    exponential running time, 289, 300
  with no negative edges, 286
Dinic's [Dinitz's] algorithm, 341
directed acyclic graph, 192, 231
directed cycle, 192
directed graph, 190
DirectedHamCycle, 398
  reduction from 3Sat, 400

reduction from VERTEXCOVER, 398
DIRECTEDHAMPATH, 401
in a directed acyclic graph, 250
in a tournament, 208
reduction to shortest simple path, 275
directed path, 192
directed walk, 192
disconnected graph traversal, 203
disjoint paths
edge-disjoint, 353
vertex-disjoint, 354
disjoint-path cover, 253
in directed acyclic graphs, 360
NP-hard in general graphs, 360
disjoint-set data structure, 267, 271
disjunctive normal form, 416
$dist_{\leq i}(v)$ (length of shortest walk to v with at most i edges), 291
$dist(u, v)$, 309
$dist(v)$ (tentative distance), 276
distance multiplication, *see* min-plus matrix multiplication
distance tables, 310
divide and conquer, 26, 29, **31**, 37, 91, 315, 323
domain transformation, 34, 56
DOMINATINGSET, 407, 422
in interval graphs, 178
dominos, 371, 421
Don't try to be clever, 80, 82, 87, 89
DP, *see* Deadpool, dynamic programming
Dr. Seuss [Theodor Suess Giesel]
On Beyond Zebra, 167
Dr. Seuss [Theodore Suess Giesel]
The Cat in the Hat Comes Back, 21
draughts, 100, 408
English ("checkers"), 222
international, 408
reduction from UNDIRECTEDHAMCYCLE, 409
drinking songs, 1, 16, 18
duplation and mediation, 5, 21, 23, 43
dynamic programming, **97**, **101**, 178, 183, 185, 222, 282
as postorder traversal, 234, 255
before Bellman, 101
boilerplate, 106
in directed acyclic graphs, 235, 250
in trees, 120
not *always* better than memoization, 117
sequential, 105
space optimization, 102
tree-shaped, 117

edge (pair of vertices), 187
edge capacity, 329
edge contraction, 269
edge demands, 347
edge gadget, 398, 402, 410
edge reweighting, 311
edge-complement $\overline{G}$, 394
edge-disjoint paths, 353
edit distance, 111, 139, 149, 193, 235, 237
Edmonds-Karp algorithms
fattest augmenting paths, 340
shortest augmenting paths, 341
EDVAC, 27
Egyptian multiplication, *see* duplation and mediation
ELEMENTARY, 415
elves, *see* Recursion Fairy
empty edge (flows), 329

endpoints of an edge, 191
epiphany, 166, 242
errors, vii
escape problem, 369
Ethiopian peasant multiplication, *see* duplation and mediation
Euclid's algorithm, 57
Euler tour, 190, 208, 223, 398
evaluation order, 107, 109, 114, 119
  as postorder, 235
  single and double arrows, 110, 120
EXACT3DIMENSIONALMATCHING, 405
EXP (exponential time), 413
EXP-hard, 414
exponential decay, 336, 340, 349, 351
exponentiation, 42

$|f|$ (flow value), 328
factorial, 57
fake-sugar-packet game, 74
Fantastic Mr. Fox, 133
fast Fourier transform, 42
feasible flow, 329
FFT, *see* fast Fourier transform
Fibonacci heaps, 264, 288
Fibonacci numbers, 98, 193, 249
Fight Club, 180
finished vertex (depth-first search), 228
finite-state automaton, 194, 253, 255
  non-deterministic, 195
    PSPACE-hard problems, 413
First make it work, then make it fast, 22, 26, 40, 80, 84, 95, 96, 99, 106, 108, 109, 123, 289
First what, then how, 106
Fizzbuzz (standard interview question), 17
flood fill, 205
flow, 328
flow decomposition, 336, 358
  algorithm, 339
flow value $|f|$, 328
flow vector space, 346
flying kings, 223, 408
Ford-Fulkerson, 334
  can run forever, 335, 340, 349
  exponential running time, 335
  fattest augmenting paths, 340
  shortest augmenting paths, 341
Ford's relaxation algorithm, 276
  exponential running time, 300
forest (= acyclic graph), 191
formula satisfiability, *see* SAT
forward edge (depth-first search), 229
French flag walk, 303
French invasion of Indochina, 24
funny matrix multiplication, *see* min-plus matrix multiplication

G_f (residual graph), 332
gadgets, 395
Gale-Shapley algorithm, 173
game state, 75, 194
game trees, 74, **75**, 79, 80, 100
garbage collection, 210
gate gadgets, 395
general patterns
  backtracking, 79
  divide and conquer, 31
  dynamic programming, 105
  graph traversal, 199
  greedy exchange arguments, 164
  minimum-spanning-tree algorithms, 259

NP-hardness proofs, 392
shortest-path algorithms, 276
generic graph traversal, *see* whatever-first search
George of the Jungle, 65
Giggle, 151, 301
Gilbert and Sullivan
HMS Pinafore, 108
The Mikado, 404
The Pirates of Penzance, 374
golden ratio, 99, 335, 349
good pivot, 33, 36
median of medians, 37
graph coloring, 395
in interval graphs, 178
graph embedding, 192
graph reduction, 205, 212
graph traversal, 199, 209, 247
disconnected graphs, 203, 205, 227
graphical statics, 189
graphs
data structures
adjacency list, 195, 198
adjacency matrix, 196, 198
comparison, 198
implicit, 198, 207, 235
historical examples, 187
modern examples, 192
terminology, 190
greatest common divisor, 57
greedy algorithms, **159**
are usually wrong, 107
that don't work, 172, 177, 183, 185, 223, 349, 373
try dynamic programming first, 108
greedy exchange arguments, 160, 161, 163, **164**, 164, 168, 258, 260
guillotine subdivision, 145
see also kd-tree
Gulliver's Travels, 2, 3, 96, 98, 145

Hamiltonian cycle, *see* DIRECTEDHAMCYCLE, UNDIRECTEDHAMCYCLE
definition, 208, 398
Hamiltonian path, *see* DIRECTEDHAMPATH, UNDIRECTEDHAMPATH
definition, 401
Handshake Lemma, 218
hashtags, 80
head of an edge, 191
Hellenistic snobbery, 57
helpful drawings
evaluation order arrows, 110, 120
NP-hardness reduction, 387
recursion trees, 32
heuristic, 30, 76
see also algorithm that doesn't work
HITTINGSET, 406
How do I. . .
choose the right problem to reduce from?, 407
derive a dynamic programming algorithm?, 106
design a backtracking algorithm?, 79
prove that a greedy algorithm is correct?, 164
prove that a problem is NP-hard?, 386, 392
Huffman codes, 165, 179
Huntington-Hill algorithm, 22
Hyperbole and a Half, 261, 291
hypercube, 208

IBM, 26
implicit graph representation, 195
in-degree of a vertex, 191

incorrect proofs that P=NP, 404, 415, 416
independent set, *see* MaxIndSet, 390
index formulation, 77, 83, 88
indice, *see* index (dammit)
induction, ii, 23, 28, 29, 40, 75, 77, 100, 103, 163–165, 175, 200, 201, 230, 242, 260, 277, 282, 284, 287, 291, 318, 334, 336–338, 341
  backward, *see* dynamic programming
  see also recursion
induction hypothesis, *see* Recursion Fairy
infinite loop, 16, 172, 277, 294, 313, 335, 349
input size, 404
integer maximum flow, 334, 348, 358
integer multiplication
  divide-and-conquer, 40
  duplation and mediation, 5
  Karatsuba's algorithm, 41
  Toom-Cook algorithm, 42
  via fast Fourier transform, 42
integer multiplication lattice algorithm, 3
Integrality Theorem (maximum flows), 334
international draughts, *see* draughts
interpuncts (word·spacing), 80
intersection graph, 192
interval graph, 192
interview questions, 124, 143, 156
inverse Ackerman function $\alpha(n)$, 267
inversion counting, 51

Jarník's algorithm, 203, 263
Jeff actually did this, 249, 349
Johnson's algorithm, 312
jump in the middle, 80, 81, 86, 89

Kaniel the Dane, 218
Karp reduction, 385
kd-tree, 65
  see also guillotine subdivision
Klondike, 407
knights and knaves, 59
Kosaraju-Sharir algorithm, 240
Kruskal's algorithm, 265
Kubla Khan, 134
kuṭṭaka, *see* Euclid's algorithm

label of a path, 251–253
labeling graph components, 204
language (set of strings), 384
largest common subtree, 377
Latin, 2, 80, 428
lattice multiplication, 3
laws of physics, irrelevance of, 336
Let that which does not matter truly slide, 12, 14, 80, 82
level of a vertex, 341
Levenshtein distance, *see* edit distance
line breaking, 130
linear-time selection, 37, 53, 55
list of NP-hard problems, 404
literal, 388
local maximum, 63
local minimum, 61
logarithmic-space reduction, 385
logic gates, 379
longest common increasing subsequence, 126
longest common subsequence, 94, 115, 125
longest increasing digital subsequence, 140
longest increasing subsequence, 86, 109, 237

LongestPath, 406
in directed acyclic graphs, 86, 88, 90, **236**
reduction from TravelingSalesman, 236
loop invariant, *see* induction hypothesis
low(v), 242
lower bound via adversary argument, 380

magnetic tape, 159
Majority3Sat, 419
majority gate, 396
many-one reduction, 385
marketing buzzwords, 102
Master Theorem, *see* recursion trees
matching, 355
non-crossing, 377
other special cases, 377
mātrāvṛtta, 97
matrice, *see* matrix (dammit)
matrix multiplication
boolean, 323
in sub-cubic time, 324
min-plus, **316**, 323, 325
standard, 323, 325
matrix rounding, 182, 374
Max2Sat, 406
MaxClique, 394
reduction from MaxIndSet, 394
MaxCut, 406
MaxIndSet, 390
in circular arc graphs, 185
in interval graphs, 162
in trees, 120
reduction from 3Sat, 390, 393, 395
reduction to MaxClique, 394
reduction to MinVertexCover, 394
maximum flows, **329**
acyclic, 339, 345
integer, 334
multiple sources and targets, 368
with vertex capacities, 354
maximum independent set, *see* MaxIndSet
maximum matching in bipartite graphs, 355
maximum subarray problem, 124
two-dimensional, 138
Maxwell-Cremona diagrams, 189
mazes, 190, 247
acute-angle, 216, 304
number, 212
median, *see* selection
median-of-medians selection, 37, 53
median-of-medians-of-medians selection, 55
median-of-three heuristic, 30, 50
memoization, 99, 194, 234, 271, 278
see also dynamic programming
memoized recursion is depth-first search, 234
mergesort, 26
mergesort recurrence, 28, 30, 33
metagraph, *see* strong component graph
methodisches Tatonnieren, 72
MinVertexCover, 394
reduction from MaxIndSet, 394
reduction to DirectedHamCycle, 398
reduction to SubsetSum, 402
min-plus matrix multiplication, **316**, 323, 325
Minesweeper, 406
minimum clique cover
in circular arc graphs, 186
in interval graphs, 178

minimum cuts, 330
minimum spanning trees, 203, **257**
    uniqueness, 258, 268
Minty's algorithm, *see* Dijkstra's algorithm
mom, *see* median of medians
mondegreen, 18
Monopoly, actual rules of, 386
Moore's algorithm, 292
Morse code, 97, 165
mountain climbing problem, 217
MST, *see* minimum spanning trees
multigraph, 191

n queens, 71, 79, 80
Nadirian Dream-Dollars, 123, 181
Napoleon Dynamite, 131
National Resident Matching Program, 171
"Needleman-Wunch" algorithm, 115
negative cycle detection, 292, 294, 299
negative cycles, 275, 277
negative edges, 274
neighbor, 191
Neitherlands (*The Magicians*), 376
nesting boxes, 372
new vertex (depth-first search), 228
NFA, *see* finite-state automaton
Nobel Prize in ~~Algorithms~~ Economics, 173
node, *see* vertex
none of your business, 21, 23, 25, 80, 206
    see also black box
NotAllEqual3Sat, 405
NP (nondeterministic polynomial time), 381
NP versus co-NP, 382
NP versus EXP, 414
NP versus PSPACE, 412
NP-complete, 383
NP-hard, 121, 138, 140, 235, 275, 358, 360, 381, **382**
    formal definition, 384
    weakly, 117, 404

obvious, 14, 76, 80, 317, 382, 383
Oh yeah, we already did this, 30, 78
one-armed quicksort, *see* quickselect
open problems
    all-pairs shortest paths, 317
    matrix multiplcation, 317
    optimal addition chains, 44
    optimal pancake flipping, 49
    P versus NP, 381
    sorting binary trees by swaps and rotations, 69
    winning international draughts in one turn, 411
open-pit mining
    see project selection, 366
optimal binary trees
    binary search trees, 91, 117
      variants, 96, 144
    expression trees, 141, 142
    prefix-free binary codes, 165
optimal substructure, *see also* correct recurrence, 112
ordered subtree, 156
Orlin's algorithm, 344
out-degree of a vertex, 191

P (polynomial time), 381
P versus NP, 382
P versus PSPACE, 412
P≠NP as a law of nature, 382
Pac-Man, 407
palindrome, 81, 128, 218, 222, 252, 377
pancake sorting, 49, 151
parallel assignment, 249
parent, 200, 276

see also mom
Partition
NP-hard problem, 405
subroutine in quicksort and quickselect, 29
party planning, 182
path compression, 271
path flow, 337
path in a graph, 191
peasant multiplication, *see* duplation and mediation
pebbling, 422
pecking order, 208
pixels, 206
Planar3Sat, 405
PlanarCircuitSat, 405
planar graph, 192
PlanarNotAllEqual3Sat, 405
Plankalkül, 205, 278
plumbus, 322
politics
academic, 59, 373, 376, 384
Illinois, 373
Renaissance Italian, 47, 59
Soviet, 383
postorder, 227
tree traversal, 65, 66, 122
power, *see* exponentiation
$pred(u, v)$, 309
$pred(v)$ (tentative predecessor), 276
predecessor of a vertex, 191
prefix, 77
prefix-free binary code, 165
preorder, 227
tree traversal, 65, 66
prerequisites, i
references, ii
Prim's algorithm, *see* Jarník's algorithm
project selection, 366
proper k-coloring, 395
proper subgraph, 191
Propositiones ad Acuendos Juvenes, 428
prosody, 97
see also Fibonacci nunbers
see also Morse code
pseudo-polynomial time, 117, 404
PSPACE (polynomial space), 412
PSPACE versus EXP, 414
PSPACE-hard, 412
punched cards, 26

QBF (quantified boolean formula), 413
quickselect, 35
quicksort, 29
quicksort recurrence, 33, 34

Racetrack, 220
rainbow, 387
RAND Corporation, 101
random-access machine, 384
$reach(v)$, 226
$reach^{-1}(v)$, 238
reachability, 191, 199
directed, 226
reciprocal diagrams, 189
recommended course policies, 108, 123
reconfiguration problems, 24, 45, 49, 68, 213, 215, 216, 219, 221, 428
recurrences
full history, 85, 92
removing floors and ceilings, 34
scary, 84, 92
solving with recursion trees, 31
recursion, **22**, 261, 291, 353
backtracking, 71
depth-first search, 225
divide and conquer, 26
smart, *see* dynamic programming

see also induction
Recursion Fairy, 22, 25, 26, 28, 37, 337
recursion trees, **31**, 49, 99
all levels equal, 28, 30, **33**, 33, 39, 49
backtracking, 72, 76
exponential decay, **33**, 37–39, 43, 49
exponential growth, **33**, 40, 49, 78, 85
path, 36, 37
weird, 30, 49
recursive brute force, *see* backtracking
reduced flow network, 331, 345
reduction, 21, 56, 205, 212, 385
regular expressions, 151
generalized, 414, 415, 417
PSPACE-hard problems, 413
relaxing a tense edge, 276
repeated squaring, 43, 44, 103, 317
replacement paths, 298
repricing, *see* vertex reweighting
residual capacity c_f, 331
residual graph G_f, 332
results by RAND researchers, 101, 276, 284, 331
results by students, 35, 40, 42, 166, 278, 324, 341, 383, 407
Revelation 13:15–18, 161
reversal $rev(G)$ of directed graph G, 234
reverse topological order, *see* postorder
Rick and Morty, 322
road maps, 187, 274, 310
rock climbing, 153, 184, 253
rooted subtree, 121
Rubik's Cube, 407
rule of three, 392, 393, 395, 405, 407, 408, 417
ruler function, 45, 46
Russian peasant multiplication, *see* duplation and mediation

$\|S, T\|$ (cut capacity), 329
(s, t)-cut, 329
(s, t)-flow, 328
safe edge, 259
SAT, 386
reduction from CIRCUITSAT, 386
satisfiability
circuit, *see* CIRCUITSAT
formula, *see* SAT
saturated edge (flows), 329
$scc(G)$ (strong component graph), 237
scheduling, 161
greedy algorithms that don't work, 177
via dynamic programming, 162
via greedy algorithm, 162, 363
via maximum flows, 358, 362
Scrabble, 134
scriptio continua, 80, 166
Seidel's algorithm, 317, 324
selection, 35, 52
median-of-medians, 37, 53
median-of-medians-of-medians, 55
quickselect, 35
self-descriptive sentence, 166
self-reduction, 416
semi-connected graph, 244
sequence alignment, *see* edit distance
sequence of decisions, 79, 112
series-parallel graph, 350
SETCOVER, 406
Sham-Poobanana University, 142, 161, 359, 362, 373

Shimbel's algorithm, *see* Bellman-Ford
shortest common supersequence, 95, 125
shortest path tree, 274
shortest paths, 203
  all-pairs, **309**
  analog algorithms, 278
  in directed acyclic graphs, 282
  in unweighted graphs, 278
  single-source, **273**
  versus shortest walks, 275
  with negative edges, 274, 288
    in undirected graphs, 275
  with non-negative edges, 284
shortest simple path
  reduction from DirectedHamPath, 275
shuffle, 126
simple graph, 191
sink (vertex with out-degree 0), 231
sink component, 239, 242
snails, 143
Snakes and Ladders, 212
soapbox, vi, 12, 106
Sollin's algorithm, *see* Borůvka's algorithm
solving a more general problem, 36, 80
solving the right problem, 83, 87, 88, 90, 91
sorting algorithms
  mergesort, 26
  quicksort, 29
source (in a flow network), 328
source (vertex with in-degree 0), 231
source component, 240
spanning forest, 192
spanning tree, 192
squaring and mediation, 44
SSSP, *see* shortest paths, single-source
stable matching, 170, 179
starting time of a vertex (depth-first search), 228, 242
SteinerTree, 406
Stigler's Law of Eponymy, 42, **98**, 114, 189, 261, 263, 284, 289, 383
Strassen's algorithm, 317
strong component graph, 237
strong components, 237
  connected in depth-first forest, 239
  in linear time, 238
    Kosaraju-Sharir, 240
    Tarjan's algorithm, 242
strong connectivity, 237
strongly connected components, *see* strong components
strongly connected graph, 192
subgraph, 191
subsequence, 86
subset construction, 195
SubsetSum, 76, 79, 80, 93, 116, 237, 402
  dynamic programming algorithm, 404
  in pseudo-polynomial time, 117
  reduction from VertexCover, 402
successor of a vertex, 191
Sudoku, 406
suffix, 83
Sumerian clay tablets, 56
summary of past decisions, 79
Super Mario Brothers, 407

Tabula Peutingeriana, 187
tail of an edge, 191
talking dog joke, 318, 383
tape sorting, 159

target (in a flow network), 328
Tarjan's algorithm, 242
tâtonner, 72
tense edge, 276, 312
Tetris, 407
text segmentation, **80**, 94, 105, 124, 130, 237
Theseus (maze-solving robot), 100, 278
Threes (game), 407
Tibetan Memory Trick, *see* The Announcer's Test
token (breadth-first search), 279
token (Moore's algorithm), 293
topological order, *see* reverse postorder
topological sort, **232**
  implicit, 232
Tower of Hanoi, 24, 44
  configuration graph, 194
  non-recursive solutions, 44
  recurrence, 26, 78, 85, 89
  variants, 46–48, 140
Trainyard, 407
transforming certificates, 392
transitive closure, 246, 323
transitive reduction, 246
TravelingSalesman, 402
  dynamic programming, 139
  Euclidean, convex position, 139
  reduction from DirectedHamCycle, 402
  reduction to LongestPath, 236
tree (connected acyclic graph), 191
  equivalent definitions, 207
tree edge (depth-first search), 229
tree traversal, 66, 227
  postorder, 122
trivial but useless $O(1)$-time algorithms, 17, 409
truth gadget, 396
TSP, *see* TravelingSalesman
tuple selection, 357
Turing machines, 384
Turing reduction, 384
Twitbook, 152, 301
typography, 80, 130

Ulam distance, *see* edit distance
undecided edge, 260
undirected graph, 190
UndirectedHamCycle, 402
  in a hypercube, 208
  reduction to international draughts, 409
UndirectedHamPath, 402
union-find, *see* disjoint-set data structure
unordered subtree, 156
UnSat, 385
useful deliberate ignorance, 22, 26, 35
useless edge, 259

vacuous base case, 25, 76, 87, 92
value of a node in a recursion tree, 31
Vankin's Kilometer, 138
Vankin's Mile, 137
variable gadget, 390, 395, 396, 400
vertex, 187
vertex cover, 394
vertex gadget, 398, 403, 410
vertex-disjoint paths, 354
vertice, *see* vertex (dammit)
Vidrach Itky Leda, 213

walk in a graph, 191
wavefront, 281, 284, 286
weakly NP-hard, 117, 404
weighted median, 53
WFS, *see* whatever-first search
Whackbat, 133
whatever-first search, 199

best-first (priority queue), *see also* best-first search, 202
breadth-first (queue), 202
depth-first (stack), 201
widest paths, 203, 270
word RAM model, 384

X3M, 405
XCNF-Sat, 419
xkcd, 407

zero cycles, 321

Dicebat Bernardus Carnotensis nos esse quasi nanos gigantium humeris insidentes, ut possimus plura eis et remotiora videre, non utique proprii visus acumine, aut eminentia corporis, sed quia in altum subvehimur et extollimur magnitudine gigantea.

[Bernard of Chartres used to say that we were like dwarfs seated on the shoulders of giants. He pointed out that we see more and farther than our predecessors, not because we have keener vision or greater height, but because we are lifted up and borne aloft on their gigantic stature.]

— John of Salisbury, *Metalogicon* (1159), translated by Daniel D. McGarry (1955)

The secret to productivity is getting dead people to do your work for you.

— Robert J. Lang (2009)

Index of People

Adelson-Velsky, Georgy, 96, 144, 341
Adler, Ilan, 364
al-Adli ar-Rumi, 190
Adversary, All-Powerful Malicious, 31, 161, 376, 379
Alcuin of York, 428
Alice, 217
Alighieri, Dante, 2
Alon, Noga, 317
Andersson, Arne, 96, 145
Apollonius of Perga, 3
Approximate Median Fairy, 33, 37
Archimedes, 3
Atlas, Charles, 102
St. Augustine of Hippo, 81

Bayer, Rudolf, 96, 144
Bellman, Richard, 101, 289
Berge, Claude, 356
Blagojevich, Rod, 373
Blum, Manuel, 35
Bob, 217
Borůvka, Otakar, 261
Brahmagupta, 3
Brosh, Allie, 261, 291

Cayley, Arthur, 190
Cegłowski, Maciej, 182

Chaucer, Geoffrey, 2
Chazelle, Bernard, 217
Chazelle, Damien, 217
Choquet, Gustav, 261
Chowdhury, Rezaul, 324
Cicero, Marcus Tullius, 80
Claus, N. (de Siam), *see* Lucas, Édouard
Clifford, William, 190
Cobham, Alan, 381
Cook, Stephen, 42, 383
Couper, Archibald, 190
Cremona, Luigi, 189
Culmann, Carl, 189

Dantzig, George, 276, 284, 289, 328
Demaine, Erik, 413
Dijkstra, Edsger, 210, 263, 269, 284
DiMaggio, Joe, 318
Dinitz, Yefim, 341
Durden, Tyler, 180
Dweighter, Harry (pseudonym of Jacob Goodman), 49

Edmonds, Jack, 311, 334, 340, 381
Elias, Peter, 331
"Engine Charlie", *see* Wilson, Charles Erwin
Eppstein, David, 209
Erera, Alan, 364
Erickson, Hannah, 216, 252
Erickson, Kay, 387
Euclid, 7, 57
Euler, Leonhard, 190, 247
Eutocius of Ascalon, 3

Fürer, Martin, 42
Fahlberg, Constantin, 74
Fano, Robert, 166
Feinstein, Amiel, 331
Fernández-Baca, David, 364
Fibonacci, *see* Leonardo of Pisa
Fischer, Michael, 114, 315
Floyd, Robert, 35, 318
Fontana, Giovanni, 190
Ford, Lester, 276, 331
Frederick II, Holy Roman Emperor, 46
Fredman, Micheal, 264
Frisius, Gemma, 190
Fulkerson, Delbert, 311, 331

Gödel, Kurt, 381
Gale, David, 172
Galil, Zvi, 317
Garey, Michael, 404
Gates, Bill, 49
Gauß, Karl Friedrich, 42, 72
Goldstine, Herman, 26
Goodrich, Michael, 209
Gregory IX, Pope, 47
Grimm, Jacob and Wilhelm, 23
Guibas, Leonidas, 96, 144
Gusfield, Dan, 364

Harris, Theodore, 327
Harvey, David, 42
Hearn, Robert, 409, 413
Herotodus, 2
Hierholzer, Carl, 190, 247
Hillier, John, 284
Hoare, Tony, 29, 35
Hochbaum, Dorit, 364
van der Hoeven, Joris, 42
Hopcroft, John, 356
Huffman, David, 166

Ingerman, Peter, 318

Jacobi, Carl, 356
Jarník, Vojtěch, 263
Jay, Ricky, 392
Johnson, David, 404
Johnson, Donald, 288, 311

Kőnig, Dénes, 356

Kalaba, Robert, 291
Kane, Daniel, 218
Karatsuba, Anatoliĭ, 40
Karp, Richard, 311, 334, 340, 356, 405
Karzanov, Alexander, 344
Kekulé, August, 190
al-Khwārizmī, Muḥammad ibn Mūsā, 2
Kirchhoff, Gustav, 190
Kleene, Stephen, 318
Kolmogorov, Andrei, 40
Kosaraju, Rao, 240
Kruskal, Joseph, 263
Kuhn, Harald, 356

Lamport, Leslie, 210
Landis Evgenii, 96, 144
Laquière, Emmanuel, 72
Ledger, Heath, 380
Lee, Chin Yang, 278
Leonardo of Pisa, 2, 4, 46, 97, 98, 101
Levin, Leonid, 383
Leyzorek, Michael, 284, 316
Loberman, Harry, 263, 266
Lucas, Édouard, 24, 72, 103
Łukaszewicz, Józef, 261

Mądry, Aleksander, 344
Margalit, Oded, 317
Marston, John, 18
Martel, Charles, 364
Martin, Alain J., 210
Martin, Steve, 10
Massé, Pierre, 101
Maxwell, James Clerk, 189
McKenna, Terence, 23
Meyer, Albert, 315
Michie, Donald, 100, 117
Miller, Gary, 50
Minty, George, 276, 284, 289
Mom, 38
Moore, Edward, 205, 273, 278, 289
Moreno, Jacob, 190
Morgenstern, Oskar, 101
Murena, Lucius Licinius, 80
Musk, Elon, 182

Nash, John, 381
Nauck, Franz, 72

Okasaki, Chris, 13
Olinick, Eli, 364
Orlin, James, 343

Pacioli, Luca, 45
Papadimitriou, Christos, 49
Pappus of Alexandria, 3
Park, Joon-Sang, 324
Peirce, Charles Sanders, 190
Penner, Michael, 324
Peranson, Elliott, 172
Piṅgala, 43, 97, 103
Pinker, Steven, 13
Pitt, Lenny, 74
Prasanna, Viktor, 324
Pratt, Vaughan, 35
Prim, Robert, 260, 263

Queyranne, Maurice, 349

Rabin, Michael, 381
Ramachandran, Vijaya, 324
Rebaudi, Ovidio, 74
Recursion Fairy, ii, 22, 77, 80, 82, 89, 164, 169
Remsen, Ira, 74
Rivest, Ronald, 35
Ross, Frank, 327
Roy, Bernard, 318
Rudraṭa, 190

Sainte-Laguë, André, 190
Sallows, Lee, 166

Samuel, Arthur, 100
Saxel, Jindřich, 261
Schönhage, Arnold, 42
Scholten, Carel S., 210
Schrijver, Lex, 327
Schumacher, Heinrich, 72
Schwartz, Benjamin, 364
Sedgewick, Robert, 96, 144, 145
Shannon, Claude, 100, 166, 190, 278, 331
Shapley, Lloyd, 172
Sharir, Micha, 240
Shier, Douglas, 289
Shimbel, Alfonso, 289, 314
Siedel, Raimund, 317
Skiena, Steve, vii
Smullyan, Raymond, 59
Snell, Willebrod, 190
Sollin, George, 261
Steele, Guy, 16
Steffens, Elisabeth, 210
Stevin, Simon, 189
Stigler, Stephen, 98
Stockmeyer, Larry, 415
Strassen, Volker, 42, 317
al-Suli, Abu Bakr Muhammad bin Yahya, 190
Sulpicius Rufus, Servius, 81
Sylvester, James, 190

Tarjan, Robert, 35, 242, 264
Tarry, Gaston, 247
Tomizawa, Nobuaki, 311
Toom, Andrei, 42
Trémaux, Charles, 247
Tseitin, Grigorii, 389
Turing, Alan, 101

Varignon, Pierre, 189
Virahāṇka, 98, 101
von Neumann, John, 26, 101, 381
von Staudt, Karl, 190

Wagner, Robert, 114
Waits, Tom, 379
Warshall, Stephen, 318
Wayne, Kevin, 366
Weinberger, Arnold, 263, 266
Weiss, Mark Allen, 96, 145
Whiting, Peter, 284
Whittlesey, Kim, 273
Wiener, Christian, 247
Wilson, Charles Erwin, 102
Witzgall, Christoph, 289
Woodbury, Max, 289

Yuval, Gideon, 325

Zermelo, Ernst, 75
Zuse, Konrad, 205, 278
Zwick, Uri, 335

We should explain, before proceeding, that it is not our object to consider this program with reference to the actual arrangement of the data on the Variables of the engine, but simply as an abstract question of the nature *and* number *of the* operations *required to be performed during its complete solution.*

— Ada Augusta Byron King, Countess of Lovelace,
translator's notes for Luigi F. Menabrea,
"Sketch of the Analytical Engine invented by Charles Babbage, Esq." (1843)

How to play the flute. [picks up a flute] Well, here we are.
You blow there and you move your fingers up and down here.

— Alan [John Cleese], "How to Do It",
Monty Python's Flying Circus, episode 28 (aired October 26, 1972)

Index of Pseudocode

This index includes only algorithms with explicit pseudocode; see the main index for other named algorithms.

AddAllSafeEdges, 262
AddSafeEdges, 272
AllPairsBellmanFord, 314
Alouette, 16
ApportionCongress, 9

BeAMillionaireAndNeverPayTaxes, 10
BellmanFord, 291, 292
BellmanFordDP, 295
BellmanFordDP2, 296
BellmanFordDP3, 296
BellmanFordFinal, 296
BFS, 279
BFSWithToken, 279
BinaryGCD, 58
Borůvka, 262, 272
BottlesOfBeer, 1

CircuitSat, 388
CollectStep, 211
ComputeOptCost, 119
ConstructSubset, 79
CountAndLabel, 204
CountComponents, 204
Cruel, 51

DagSSSP, 283
DFS, 225, 226, 228, 235
DFSAll, 227, 228
Dijkstra, 285
DynamicProgramming, 235

EagerWFS, 209
EuclidGCD, 58

Factorial, 57
Falling, 57
FastEuclidGCD, 58
FastLIS, 110
FastLIS2, 111
FastMultiply, 41
FastRecFibo, 104
FastSplittable, 106
FastSubsetSum, 117
FellmanBored, 299
FetchBit, 63
FibonacciMultiply, 4
FindLow, 243
FindLowDFS, 243
FindSafeEdges, 272
FischerMeyerAPSP, 315
FloydWarshall, 319
FordSSSP, 277

GarbageCollect, 211
GreedyFlow, 349
GreedySchedule, 163

Hanoi, 26
HHGuess, 19

InitF, 118
InitSSSP, 276
IsAcyclic, 231
IsAcyclicDFS, 231
IterativeDFS, 199
IterFibo, 101
IterFibo2, 103

Jarník, 265
JarníkInit, 265
JarníkLoop, 265
JohnsonAPSP, 313

KleeneAPSP, 319
KosarajuSharir, 241
Kruskal, 267

LabelOne, 204
LeyzorekAPSP, 316
LIS, 89, 90
LISbigger, 88
LISfirst, 90
LongestPath, 236, 237

MarkEveryVertexDuh, 203
MemFibo, 100
Memoize, 235
Merge, 27
MergeSort, 27
MomomSelect, 55
MomSelect, 37
$\text{Mom}_b\text{Select}$, 53
Moore, 293
MultiplyOrDivide, 8
Mutate, 212

NonnegativeDijkstra, 288

ObviousAPSP, 310
OptimalBST, 119
OptimalBST2, 120
OptimalBST3, 120

Partition, 29
PeasantMultiply, 6, 23
PeasantPower, 44
PiṅgalaPower, 43
PlaceQueens, 73
PlayAnyGame, 76
PostProcess, 233
PostProcessDag, 233
PostProcessDagDFS, 233
PostProcessDFS, 233
PostVisit, 227

Preprocess, 227
PreVisit, 227
PushDagSSSP, 284

QueyranneFatPaths, 350
QuickSelect, 36
QuickSort, 29

RecFibo, 99
RecTarry, 247
RecTarry2, 248
RecursiveDFS, 199
Relax, 277
RightAngle, 8
RulerHanoi, 45

ShimbelAPSP, 314
ShorterEdge, 259
SlowPower, 43
SplitMultiply, 40
Splittable, 83
SqrtSort, 52
StoogeSort, 50
StrongComponents, 239
SubsetSum, 77, 78

Tarjan, 244
TarjanDFS, 244
Tarry, 247
Tarry2, 248
ThreeColorQueueSearch, 211
ThreeColorQueueStep, 211
ThreeColorSearch, 210
ThreeColorStackSearch, 210
ThreeColorStackStep, 210
ThreeColorStep, 210
TopologicalSort, 233, 234
TopSortDFS, 233
TreeMIS, 122

Unusual, 51

WFSAll, 203
WhateverFirstSearch, 200, 205
WhoTargetsWhom, 62

A wisely chosen illustration is almost essential to fasten the truth upon the ordinary mind, and no teacher can afford to neglect this part of his preparation.

— Howard Crosby (c.1880)

One showing is worth a hundred sayings.

— Alan Watts (misquoting a Chinese proverb), *The Way of Zen* (1957)

Please do not think that this is a neutral matter and that the only advantage of doing without pictures is that of saving space. Pictures in textbooks actually interfere with the learning process.

— Neville Martin Gwynne, *Gwynne's Grammar* (2013)

Image Credits

All figures in this book, including the front cover, are original works of the author, except those listed below. All listed works are in the public domain unless otherwise indicated.

- Figure 0.1 (page 5) — Biblioteca nazionale Braidense (Milano)
  http://atena.beic.it/webclient/DeliveryManager?pid=2953344
- Figure 0.2 (page 5) — Internet Archive
  https://archive.org/details/archimedisopera05eutogoog/page/n377
- Figure 1.16 (page 45) — Internet Archive
  https://archive.org/details/p1rcrationsmoolucauoft/page/162
- Figure 1.25 (page 61) — Derived from a crayon portrait of the author by Tina Erickson (2000); included with permission of the artist.
- Figure 5.1 (page 188) — Wikimedia Commons
  https://commons.wikimedia.org/wiki/File:Tabula_Peutingeriana_-_Miller.jpg

- Figure 5.2 (page 189) — Gallery of "Legal Trees" published by the Yale Law Library under a Creative Commons Licence
  https://www.flickr.com/photos/yalelawlibrary/albums/72157621954683764
- Figure 5.3 (page 189) — Internet Archive
  https://archive.org/details/A077240124/page/n261
- Exercises 5.20 (page 216) and 8.22 (page 304) — Original puzzles by the author, inspired by Jason Batterson and Shannon Rogers, *Beast Academy Math: Practice 3A*, 2012.
  https://beastacademy.com/pdf/3A/printables/AngleMazes.pdf
  https://www.beastacademy.com/resources/printables.php
- Figure 10.1 (page 328) — T[homas] E. Harris and F[rank] S. Ross. Fundamentals of a method for evaluating rail net capacities. The RAND Corporation, Research Memorandum RM-1517, October 24, 1955. United States Government work in the public domain.
  http://www.dtic.mil/dtic/tr/fulltext/u2/093458.pdf

1. Have something to say.
2. Say it.
3. Stop when you have said it.
4. Give the paper a proper title.

— John Shaw Billings, "An Address on Our Medical Literature", International Medical Congress, London (1881)

You know, I could write a book.
And this book would be thick enough to stun an ox.

— Laurie Anderson, "Let X=X", *Big Science* (1982)

Colophon

This book was edited in TeXShop (version 4.27) and typeset with pdfLaTeX (MacTeX-2018) using the `memoir` document class (with `madsen` chapter style, `komalike` head style, and `Ruled` page style); several standard packages including `amsmath`, `babel`, `enumitem`, `imakeidx`, `mathdesign`, `microtype`, and `standalone`; and an embarrassing amount of customization and TeX-hAXing. The text is typeset in Bitstream Charter, Ἀρτεμισία, Roboto, and `Inconsolata`. Except as indicated in the Image Credits, all figures were drawn by the author using OmniGraffle Pro, exported at PDF files, and included using the `graphicx` LaTeX package.

Portions of our programming have been mechanically reproduced, and we now conclude our broadcast day.

Ceterum in problematis natura fundatum est, ut methodi quaecunque *continuo prolixiores evadant, quo maiores sunt numeri, ad quos applicantur.*
[It is in the nature of the problem that any *method will become more prolix as the numbers to which it is applied grow larger.]*

— Carl Friedrich Gauß, *Disquisitiones Arithmeticae* (1801)
English translation by A.A. Clarke (1965)

Illam vero methodum calculi mechanici taedium magis minuere, praxis tentantem docebit.
[Truly, that method greatly reduces the tedium of mechanical calculations; practice will teach whoever tries it.]

— Carl Friedrich Gauß, "Theoria interpolationis methodo nova tractata" (c. 1805)

After much deliberation, the distinguished members of the international committee decided unanimously (when the Russian members went out for a caviar break) that since the Chinese emperor invented the method before anybody else had even been born, the method should be named after him. The Chinese emperor's name was Fast, so the method was called the Fast Fourier Transform.

— Thomas S. Huang, "How the fast Fourier transform got its name" (1971)

A

Fast Fourier Transforms

[Read Chapters 0 and 1 first.]
Status: Beta

A.1 Polynomials

Polynomials are functions of one variable built from additions, subtractions, and multiplications (but no divisions). The most common representation for a polynomial $p(x)$ is as a sum of weighted powers of the variable x:

$$p(x) = \sum_{j=0}^{n} a_j x^j.$$

The numbers a_j are called the *coefficients* of the polynomial. The *degree* of the polynomial is the largest power of x whose coefficient is not equal to zero; in the example above, the degree is *at most* n. Any polynomial of degree n can be represented by an array $P[0..n]$ of $n+1$ coefficients, where $P[j]$ is the coefficient of the x^j term, and where $P[n] \neq 0$.

Here are three of the most common operations performed on polynomials:

© 2018 Jeff Erickson

- **Evaluate:** Give a polynomial p and a number x, compute the number $p(x)$.
- **Add:** Give two polynomials p and q, compute a polynomial $r = p + q$, so that $r(x) = p(x) + q(x)$ for all x. If p and q both have degree n, then their sum $p + q$ also has degree n.
- **Multiply:** Give two polynomials p and q, compute a polynomial $r = p \cdot q$, so that $r(x) = p(x) \cdot q(x)$ for all x. If p and q both have degree n, then their product $p \cdot q$ has degree $2n$.

We learned simple algorithms for all three of these operations in high-school algebra. The addition and multiplication algorithms are straightforward generalizations of the standard algorithms for integer arithmetic.

```
EVALUATE(P[0..n], x):
  X ← 1      ⟨⟨X = x^j⟩⟩
  y ← 0
  for j ← 0 to n
      y ← y + P[j] · X
      X ← X · x
  return y
```

```
ADD(P[0..n], Q[0..n]):
  for j ← 0 to n
      R[j] ← P[j] + Q[j]
  return R[0..n]
```

```
MULTIPLY(P[0..n], Q[0..m]):
  for j ← 0 to n + m
      R[j] ← 0
  for j ← 0 to n
      for k ← 0 to m
          R[j + k] ← R[j + k] + P[j] · Q[k]
  return R[0..n + m]
```

EVALUATE uses $O(n)$ arithmetic operations.[1] This is the best we can hope for, but we can cut the number of multiplications in half using *Horner's rule*:

$$p(x) = a_0 + x(a_1 + x(a_2 + \ldots + xa_n)).$$

```
HORNER(P[0..n], x):
  y ← P[n]
  for i ← n − 1 down to 0
      y ← x · y + P[i]
  return y
```

The addition algorithm also runs in $O(n)$ time, and this is clearly the best we can do.

The multiplication algorithm, however, runs in $O(n^2)$ time. In Chapter 1, we saw a divide-and-conquer algorithm (due to Karatsuba) for multiplying two n-bit integers

[1] All time analysis in this lecture assumes that each arithmetic operation takes $O(1)$ time. This may not be true in practice; in fact, one of the most powerful applications of fast Fourier transforms is fast *integer* multiplication. The fastest algorithm currently known for multiplying two n-bit integers, published by Martin Fürer in 2007, uses $O(n \log n \, 2^{O(\log^* n)})$ bit operations and is based on fast Fourier transforms.

in only $O(n^{\lg 3})$ steps; precisely the same approach can be applied here. Even cleverer divide-and-conquer strategies lead to multiplication algorithms whose running times are arbitrarily close to linear—$O(n^{1+\varepsilon})$ for your favorite value $e > 0$—but except for a few simple cases, these algorithms not worth the trouble in practice, thanks to large hidden constants.

A.2 Alternate Representations

Part of what makes multiplication so much harder than the other two operations is our input representation. Coefficient vectors are the most common representation for polynomials, but there are at least two other useful representations.

Roots

The Fundamental Theorem of Algebra states that every polynomial p of degree n has exactly n *roots* $r_1, r_2, \ldots r_n$ such that $p(r_j) = 0$ for all j. Some of these roots may be irrational; some of these roots may by complex; and some of these roots may be repeated. Despite these complications, this theorem implies a unique representation of any polynomial of the form

$$p(x) = s \prod_{j=1}^{n} (x - r_j)$$

where the r_j's are the roots and s is a scale factor. Once again, to represent a polynomial of degree n, we need a list of $n+1$ numbers: one scale factor and n roots.

Given a polynomial in this root representation, we can clearly evaluate it in $O(n)$ time. Given two polynomials in root representation, we can easily multiply them in $O(n)$ time by multiplying their scale factors and just concatenating the two root sequences.

Unfortunately, if we want to add two polynomials in root representation, we're out of luck. There's essentially *no* correlation between the roots of p, the roots of q, and the roots of $p + q$. We could convert the polynomials to the more familiar coefficient representation first—this takes $O(n^2)$ time using the high-school algorithms—but there's no easy way to convert the answer back. In fact, for most polynomials of degree 5 or more in coefficient form, it's *impossible* to compute roots exactly.[2]

Samples

Our third representation for polynomials comes from a different consequence of the Fundamental Theorem of Algebra. Given a list of $n+1$ pairs $\{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\}$, there is *exactly one* polynomial p of degree n such that $p(x_j) = y_j$ for all j. This is just a generalization of the fact that any two points determine a unique line, because a

[2] This is where numerical analysis comes from.

line is the graph of a polynomial of degree 1. We say that the polynomial p ***interpolates*** the points (x_j, y_j). As long as we agree on the sample locations x_j in advance, we once again need exactly $n+1$ numbers to represent a polynomial of degree n.

Adding or multiplying two polynomials in this sample representation is easy, as long as they use the same sample locations x_j. To add the polynomials, just add their sample values. To multiply two polynomials, just multiply their sample values; however, if we're multiplying two polynomials of degree n, we must *start* with $2n+1$ sample values for each polynomial, because that's how many we need to uniquely represent their product. Both algorithms run in $O(n)$ time.

Unfortunately, evaluating a polynomial in this representation is no longer straightforward. The following formula, due to Lagrange, allows us to compute the value of any polynomial of degree n at any point, given a set of $n+1$ samples.

$$p(x) = \sum_{j=0}^{n-1} \left(\frac{y_j}{\prod_{k \neq j}(x_j - x_k)} \prod_{k \neq j} (x - x_k) \right)$$

Hopefully it's clear that formula actually describes a polynomial function of x, since each term in the sum is a scaled product of monomials. It's also not hard to verify that $p(x_j) = y_j$ for every index j; most of the terms of the sum vanish. As I mentioned earlier, the Fundamental Theorem of Algebra implies that p is *the only* polynomial that interpolates the points $\{(x_j, y_j)\}$. Lagrange's formula can be translated mechanically into an $O(n^2)$-time algorithm.

Summary

We find ourselves in the following frustrating situation. We have three representations for polynomials and three basic operations. Each representation allows us to almost trivially perform a different pair of operations in linear time, but the third takes at least quadratic time, if it can be done at all!

representation	evaluate	add	multiply
coefficients	$O(n)$	$O(n)$	$\boldsymbol{O(n^2)}$
roots + scale	$O(n)$	$\boldsymbol{\infty}$	$O(n)$
samples	$\boldsymbol{O(n^2)}$	$O(n)$	$O(n)$

A.3 Converting Between Representations

What we need are fast algorithms to convert quickly from one representation to another. Then if we need to perform an operation that's hard for our default representation, we can switch to a different representation that makes the operation easy, perform the desired operation, and then switch back. This strategy immediately rules out the root

representation, since (as I mentioned earlier) finding roots of polynomials is impossible in general, at least if we're interested in exact results.

So how do we convert from coefficients to samples and back? Clearly, once we choose our sample positions x_j, we can compute each sample value $y_j = p(x_j)$ in $O(n)$ time from the coefficients using Horner's rule. So we can convert a polynomial of degree n from coefficients to samples in $O(n^2)$ time. Lagrange's formula can be used to convert the sample representation back to the more familiar coefficient form. If we use the naïve algorithms for adding and multiplying polynomials (in coefficient form), this conversion takes $O(n^3)$ time.

We can improve the cubic running time by observing that *both* conversion problems boil down to computing the product of a matrix and a vector. The explanation will be slightly simpler if we assume the polynomial has degree $n-1$, so that n is the number of coefficients or samples. Fix a sequence $x_0, x_1, \ldots, x_{n-1}$ of sample *positions*, and let V be the $n \times n$ matrix where $v_{ij} = x_i^j$ (indexing rows and columns from 0 to $n-1$):

$$V = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix}.$$

The matrix V is called a ***Vandermonde matrix***. The vector of coefficients $\vec{a} = (a_0, a_1, \ldots, a_{n-1})$ and the vector of sample *values* $\vec{y} = (y_0, y_1, \ldots, y_{n-1})$ are related by the matrix equation

$$V\vec{a} = \vec{y},$$

or in more detail,

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix}.$$

Given this formulation, we can clearly transform any coefficient vector $\vec{a}$ into the corresponding sample vector $\vec{y}$ in $O(n^2)$ time.

Conversely, if we know the sample values $\vec{y}$, we can recover the coefficients by solving a system of n linear equations in n unknowns, which can be done in $O(n^3)$ time using Gaussian elimination.[3] But we can speed this up by implicitly hard-coding the sample

[3] In fact, Lagrange's formula is just a special case of Cramer's rule for solving linear systems.

positions into the algorithm, To convert from samples to coefficients, we can simply multiply the sample vector by the inverse of V, again in $O(n^2)$ time:

$$\vec{a} = V^{-1}\vec{y}.$$

Computing V^{-1} would take $O(n^3)$ time if we had to do it from scratch using Gaussian elimination, but because we fixed the set of sample positions in advance, the matrix V^{-1} can be hard-coded directly into the algorithm.[4]

So we can convert from coefficients to samples and back in $O(n^2)$ time. At first lance, this result seems pointless; we can already add, multiply, or evaluate directly in either representation in $O(n^2)$ time, so why bother? But there's a degree of freedom we haven't exploited yet: ***We get to choose the sample positions!*** Our conversion algorithm is slow only because we're trying to be too general. If we choose a set of sample positions with the right recursive structure, we can perform this conversion more quickly.

A.4 Divide and Conquer

Any polynomial of degree at most $n-1$ can be expressed as a combination of two polynomials of degree at most $(n/2)-1$ as follows:

$$p(x) = p_{\text{even}}(x^2) + x \cdot p_{\text{odd}}(x^2).$$

The coefficients of p_{even} are just the even-degree coefficients of p, and the coefficients of p_{odd} are just the odd-degree coefficients of p. Thus, we can evaluate $p(x)$ by recursively evaluating $p_{\text{even}}(x^2)$ and $p_{\text{odd}}(x^2)$ and performing $O(1)$ additional arithmetic operations.

Now call a set X of n values ***collapsing*** if either of the following conditions holds:

- X has one element.
- The set $X^2 = \left\{x^2 \;\middle|\; x \in X\right\}$ has exactly $n/2$ elements and is (recursively) collapsing.

Clearly the size of any collapsing set is a power of 2. Given a polynomial p of degree $n-1$, and a collapsing set X of size n, we can compute the set $\{p(x) \mid x \in X\}$ of sample values as follows:

1. Recursively compute $\left\{p_{\text{even}}(x^2) \;\middle|\; x \in X\right\} = \left\{p_{\text{even}}(y) \;\middle|\; y \in X^2\right\}$.
2. Recursively compute $\left\{p_{\text{odd}}(x^2) \;\middle|\; x \in X\right\} = \left\{p_{\text{odd}}(y) \;\middle|\; y \in X^2\right\}$.
3. For each $x \in X$, compute $p(x) = p_{\text{even}}(x^2) + x \cdot p_{\text{odd}}(x^2)$.

The running time of this algorithm satisfies the familiar recurrence $T(n) = 2T(n/2) + \Theta(n)$, which as we all know solves to $T(n) = \Theta(n \log n)$.

[4]Actually, it is possible to invert an $n \times n$ matrix in $o(n^3)$ time, using fast matrix multiplication algorithms that closely resemble Karatsuba's sub-quadratic divide-and-conquer algorithm for integer/polynomial multiplication. On the other hand, my numerical-analysis colleagues have reasonable cause to shoot me in the face for daring to suggest, even in passing, that anyone actually invert a matrix at all, ever.

Great! Now all we need is a sequence of arbitrarily large collapsing sets. The simplest method to construct such sets is just to invert the recursive definition: If X is a collapsible set of size n *that does not contain the number* 0, then $\sqrt{X} = \{\pm\sqrt{x} \mid x \in X\}$ is a collapsible set of size $2n$. This observation gives us an infinite sequence of collapsible sets, starting as follows:[5]

$$
\begin{aligned}
X_1 &:= \{1\} \\
X_2 &:= \{1, -1\} \\
X_4 &:= \{1, -1, i, -i\} \\
X_8 &:= \left\{1, -1, i, -i, \frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}i, -\frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}i, \frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}i, -\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}i\right\}
\end{aligned}
$$

A.5 The Discrete Fourier Transform

For any n, the elements of X_n are called the ***complex nth roots of unity***; these are the roots of the polynomial $x^n - 1 = 0$. These n complex values are spaced exactly evenly around the unit circle in the complex plane. Every nth root of unity is a power of the *primitive* nth root

$$\boxed{\omega_n = e^{2\pi i/n} = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}.}$$

A typical nth root of unity has the form

$$\omega_n^k = e^{(2\pi i/n)k} = \cos\left(\frac{2\pi}{n}k\right) + i\sin\left(\frac{2\pi}{n}k\right).$$

These complex numbers have several useful properties for any integers n and k:

- There are exactly n different nth roots of unity: $\omega_n^k = \omega_n^{k \bmod n}$.
- If n is even, then $\omega_n^{k+n/2} = -\omega_n^k$; in particular, $\omega_n^{n/2} = -\omega_n^0 = -1$.
- $1/\omega_n^k = \omega_n^{-k} = \overline{\omega_n^k} = (\overline{\omega_n})^k$, where the bar represents complex conjugation: $\overline{a + bi} = a - bi$
- $\omega_n = \omega_{kn}^k$. Thus, every nth root of unity is also a (kn)th root of unity.

These properties imply immediately that if n is a power of 2, then the set of all nth roots of unity is collapsible!

If we sample a polynomial of degree $n-1$ at the nth roots of unity, the resulting list of sample values is called the ***discrete Fourier transform*** of the polynomial (or more

[5]In this chapter, lower case italic i always represents the square root of -1. Computer scientists are used to thinking of i as an integer index into a sequence, an array, or a for-loop, but we obviously can't do that here. The engineers' habit of using $j = \sqrt{-1}$ just delays the problem—How do engineers write quaternions?—and typographical hacks like I or $\mathbf{i}$ or ι or Mathematica's $\mathring{\imath}$ are just stupid.

formally, of its coefficient vector). Thus, given an array $P[0..n-1]$ of coefficients, its discrete Fourier transform is the vector $P^*[0..n-1]$ defined as follows:

$$\boxed{P^*[j] \;:=\; p(\omega_n^j) \;=\; \sum_{k=0}^{n-1} P[k]\cdot\omega_n^{jk}}$$

As we already observed, the fact that sets of roots of unity are collapsible implies that we can compute the discrete Fourier transform in $O(n\log n)$ time. The resulting algorithm, called the ***fast Fourier transform***, was popularized by Cooley and Tukey in 1965.[6] The algorithm assumes that n is a power of two; if necessary, we can just pad the coefficient vector with zeros.

```
RADIX2FFT(P[0..n−1]):
  if n = 1
      return P

  for j ← 0 to n/2 − 1
      U[j] ← P[2j]
      V[j] ← P[2j + 1]

  U* ← RADIX2FFT(U[0..n/2 − 1])
  V* ← RADIX2FFT(V[0..n/2 − 1])

  ω_n ← cos(2π/n) + i sin(2π/n)
  ω ← 1

  for j ← 0 to n/2 − 1
      P*[j]         ← U*[j] + ω · V*[j]
      P*[j + n/2] ← U*[j] − ω · V*[j]
      ω ← ω · ω_n

  return P*[0..n − 1]
```

Figure A.1. The Cooley-Tukey radix-2 fast Fourier transform algorithm.

Variants of this divide-and-conquer algorithm were previously described by Good in 1958, by Thomas in 1948, by Danielson and Lánczos in 1942, by Stumpf in 1937, by Yates in 1932, and by Runge in 1903; some special cases were published even earlier by Everett in 1860, by Smith in 1846, and by Carlini in 1828. But the algorithm, in its full modern recursive generality, was first described *and used* by Gauss around 1805 for calculating the periodic orbits of asteroids from a finite number of observations. In fact, Gauss's recursive algorithm predates even Fourier's introduction of harmonic analysis by two years. So, of course, the algorithm is universally called the *Cooley-Tukey*

[6]Tukey apparently developed this algorithm to help detect Soviet nuclear tests without actually visiting Soviet nuclear facilities, by interpolating off-shore seismic readings. Without his rediscovery, the nuclear test ban treaty might never have been ratified, and we might all be speaking Russian, or more likely, whatever language radioactive glass speaks.

algorithm. Gauss's work built on earlier research on trigonometric interpolation by Bernoulli, Lagrange, Clairaut, and Euler; in particular, the first explicit description of the discrete "Fourier" transform was published by Clairaut in 1754, more than half a century before Fourier's work. Alas, nobody will understand you if you talk about Gauss's fast Clairaut transform algorithms. Hooray for Stigler's Law![7]

A.6 More General Factoring

The algorithm in Figure A.1 is often called the ***radix-2 fast Fourier transform*** to distinguish it from other FFT algorithms. In fact, both Gauss and (much later) Cooley and Tukey described a more general divide-and-conquer strategy that does not assume n is a power of two, but can be applied to any composite order n. Specifically, if $n = pq$, we can decompose the discrete Fourier transform of order n into simpler discrete Fourier transforms as as follows. For all indices $0 \le a < p$ and $0 \le b < q$, we have

$$\begin{aligned}
x^*_{aq+b} &= \sum_{\ell=0}^{pq-1} x_{jp+k}\,(\omega_{pq}^{jp+k})^{\ell} \\
&= \sum_{k=0}^{p-1}\sum_{j=0}^{q-1} x_{jp+k}\,\omega_{pq}^{(aq+b)(jp+k)} \\
&= \sum_{k=0}^{p-1}\sum_{j=0}^{q-1} x_{jp+k}\,\omega_{q}^{bj}\,\omega_{pq}^{bk}\,\omega_{p}^{ak} \\
&= \sum_{k=0}^{p-1}\left(\left(\sum_{j=0}^{q-1} x_{jp+k}\,(\omega_q^b)^j\right)\omega_{pq}^{bk}\right)(\omega_p^a)^k,
\end{aligned}$$

The innermost sum in this expression is one coefficient of a discrete Fourier transform of order q, and the outermost sum is one coefficient of a discrete Fourier transform of order q. The intermediate factors ω_{pq}^{bk} are now formally known as "twiddle factors".

[7] Lest anyone believe that Stigler's Law has treated Gauss unfairly, remember that "Gaussian elimination" was not discovered by Gauss; the algorithm was not even given that name until the mid-20th century! Elimination became the standard method for solving systems of linear equations in Europe in the early 1700s, when it appeared in Isaac Newton's influential textbook *Arithmetica universalis*.[8] Although Newton apparently (and perhaps even correctly) believed he had invented the elimination algorithm, it actually appears in several earlier works, including the eighth chapter of the Chinese manuscript *The Nine Chapters of the Mathematical Art*. The authors and precise age of the *Nine Chapters* are unknown, but commentary written by Liu Hui in 263CE claims that the text was already several centuries old. It was almost certainly *not* invented by a Chinese emperor named Fast.

[8] *Arithmetica universalis* was compiled from Newton's lecture notes and published over Newton's strenuous objections. He refused to have his name associated with the book, and he even considered buying up every copy of the first printing to destroy them. Apparently he didn't want anyone to think it was his latest research. The first edition crediting Newton as the author did not appear until 25 years after his death.

No, seriously, that's actually what they're called. This wall of symbols implies that the discrete Fourier transform of order n can be evaluated as follows:

1. Write the input vector into a $p \times q$ array in row-major order.
2. Apply a discrete Fourier transform of order p to each column of the 2d array.
3. Multiply each entry of the 2d array by the appropriate twiddle factor.
4. Apply a discrete Fourier transform of order q to each row of the 2d array.
5. Extract the output vector from the $p \times q$ array in column-major order.

The algorithm is described in more detail in Figure A.2.

```
FactorFFT(P[0 .. pq − 1]):
  ⟨⟨Copy/typecast to 2d array in row-major order⟩⟩
  for j ← 0 to p − 1
      for k ← 0 to q − 1
          A[j, k] ← P[jp + k]

  ⟨⟨Recursively apply order-p FFTs to columns⟩⟩
  for k ← 0 to q − 1
      B[·, k] ← FFT(A[·, k])

  ⟨⟨Multiply by twiddle factors⟩⟩
  for j ← 0 to p − 1
      for k ← 0 to q − 1
          B[·, k] ← B[·, k] · ω_pq^jk

  ⟨⟨Recursively apply order-q FFTs to rows⟩⟩
  for j ← 0 to p − 1
      C[j, ·] ← FFT(C[j, ·])

  ⟨⟨Copy/typecast to 1d array in column-major order⟩⟩
  for j ← 0 to p − 1
      for k ← 0 to q − 1
          P*[j + kq] ← C[j, k]

  return P*[0 .. pq − 1]
```

Figure A.2. The Gauss-Cooley-Tukey FFT algorithm

We can recover the original radix-2 FFT algorithm in Figure A.1 by setting $p = n/2$ and $q = 2$. The lines $P^*[j] \leftarrow U^*[j] + \omega \cdot V^*[j]$ and $P^*[j + n/2] \leftarrow U^*[j] - \omega \cdot V^*[j]$ are applying an order-2 discrete Fourier transform; in particular, the multipliers ω and $-\omega$ are the "twiddle factors".

Both Gauss[9] and Cooley and Tukey recommended applying this factorization approach recursively. Cooley and Tukey observed further that if all prime factors of n are

[9]Gauss wrote: "*Nulla iam amplius explicatione opus erit, quomodo illa partitio adhuc ulterius extendi et ad eum casum applicari possit, ubi multitudo omnium valorum propositorum numerus e tribus pluribusve factoribus compositus est, e.g. si numerus μ rursus esset compositus, in quo casu manifesto quaevis periodus μ terminorum in plures periodos minores subdividi potest.*" ["There is no need to explain how that division can be extended further and applied to the case where most of the proposed values are composed of three or more factors, for example, if the number μ is again composite, in which case each period of μ terms can

smaller than some constant, so that the subproblems at the leaves of the recursion tree can be solved in $O(1)$ time, the entire algorithm runs in only $O(n \log n)$ time.

Using a completely different approach, Charles Rader and Leo Bluestein described FFT algorithms that run in $O(n \log n)$ time when n is an arbitrary prime number, by reducing to two FFTs whose orders have only small prime factors. Combining these two approaches yields an $O(n \log n)$-time algorithm to compute discrete Fourier transforms of any order n.

A.7 Inverting the FFT

We also need to recover the coefficients of the product from the new sample values. Recall that the transformation from coefficients to sample values is *linear*; the sample vector is the product of a Vandermonde matrix V and the coefficient vector. For the discrete Fourier transform, each entry in V is an nth root of unity; specifically,

$$\boxed{v_{jk} = \omega_n^{jk}}$$

for all integers j and k. More explicitly:

$$V = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix}$$

To invert the discrete Fourier transform, converting sample values back to coefficients, it suffices to multiply the vector P^* of sample values by the inverse matrix V^{-1}. The following amazing fact implies that this is almost the same as multiplying by V itself:

Lemma A.1. $V^{-1} = \overline{V}/n$

Proof: We just have to show that $M = V\overline{V}$ is the identity matrix scaled by a factor of n. We can compute a single entry in M as follows:

$$m_{jk} = \sum_{l=0}^{n-1} \omega_n^{jl} \cdot \overline{\omega_n}^{lk} = \sum_{l=0}^{n-1} \omega_n^{jl-lk} = \sum_{l=0}^{n-1} (\omega_n^{j-k})^l$$

obviously be subdivided into several smaller periods."]

If $j = k$, then $\omega_n^{j-k} = \omega_n^0 = 1$, so

$$m_{jk} = \sum_{l=0}^{n-1} 1 = n,$$

and if $j \neq k$, we have a geometric series

$$m_{jk} = \sum_{l=0}^{n-1} (\omega_n^{j-k})^l = \frac{(\omega_n^{j-k})^n - 1}{\omega_n^{j-k} - 1} = \frac{(\omega_n^n)^{j-k} - 1}{\omega_n^{j-k} - 1} = \frac{1^{j-k} - 1}{\omega_n^{j-k} - 1} = 0. \qquad \square$$

In other words, if $W = V^{-1}$ then $w_{jk} = \overline{v_{jk}}/n = \overline{\omega_n^{jk}}/n = \omega_n^{-jk}/n$. What this observation implies for us computer scientists is that any algorithm for computing the discrete Fourier transform can be trivially adapted or modified to compute the inverse transform as well; see Figure A.3.

```
InverseFFT(P*[0..n−1]):
  P[0..n−1] ← FFT(P*)
  for j ← 0 to n−1
      P*[j] ← conj(P*[j])/n
  return P[0..n−1]
```

```
InverseRadix2FFT(P*[0..n−1]):
  if n = 1
      return P

  for j ← 0 to n/2−1
      U*[j] ← P*[2j]
      V*[j] ← P*[2j+1]

  U ← InverseRadix2FFT(U*[0..n/2−1])
  V ← InverseRadix2FFT(V*[0..n/2−1])

  conj(ω_n) ← cos(2π/n) − i sin(2π/n)
  conj(ω) ← 1

  for j ← 0 to n/2−1
      P[j]         ← (U[j] + conj(ω)·V[j])/2
      P[j+n/2] ← (U[j] − conj(ω)·V[j])/2
      conj(ω) ← conj(ω)·conj(ω_n)

  return P[0..n−1]
```

Figure A.3. Generic and radix-2 inverse FFT algorithms.

A.8 Fast Polynomial Multiplication

Finally, given two polynomials p and q, represented by an arrays of length m and n, respectively, we can multiply them in $\Theta((m+n)\log(m+n))$ arithmetic operations as follows. First, pad the coefficient vectors with zeros to length $m+n$ (or to the next larger power of 2 if we plan to use the radix-2 FFT algorithm). Then compute the discrete Fourier transforms of each coefficient vector, and multiply the resulting sample values one by one. Finally, compute the inverse discrete Fourier transform of the resulting sample vector.

```
FFTMultiply(P[0..m−1], Q[0..n−1]):
  for j ← m to m+n−1
      P[j] ← 0
  for j ← n to m+n−1
      Q[j] ← 0
  P* ← FFT(P)
  Q* ← FFT(Q)
  for j ← 0 to 2^ℓ − 1
      R*[j] ← P*[j] · Q*[j]
  return InverseFFT(R*)
```

Figure A.4. Multiplying polynomials in $O((m+n)\log(m+n))$ time.

A.9 Inside the Radix-2 FFT

Fast Fourier transforms are often implemented in hardware as circuits; Cooley and Tukey's radix-2 algorithm unfolds into a particularly nice recursive structure, as shown in Figure A.5 for $n = 16$. On the left, the n top-level inputs and outputs are connected to the inputs and outputs of the recursive calls, represented here as gray boxes. On the left we split the input P into two recursive inputs U and V. On the right, we combine the outputs U^* and V^* to obtain the final output P^*.

Figure A.5. The recursive structure of the radix-2 FFT algorithm.

If we expand this recursive structure completely, we see that the circuit splits naturally into two parts.

- The left half computes the *bit-reversal permutation* of the input. To find the position of $P[k]$ in this permutation, write k in binary and then read the bits backward. For example, in an 8-element bit-reversal permutation, $P[3] = P[011_2]$ ends up in position $6 = 110_2$.

- The right half of the FFT circuit is called a *butterfly network*. Butterfly networks are often used to route between processors in massively-parallel computers, because they allow any two processors to communicate in only $O(\log n)$ steps.

When n is a power of 2, recursively applying the more general FactorFFT gives us exactly the same recursive structure, just clustered differently. For many applications of FFTs, including polynomial multiplication, the bit-reversal permutation is unnecessary and can actually be omitted.

Exercises

1. For any two sets X and Y of integers, the Minkowski sum $X + Y$ is the set of all pairwise sums $\{x + y \mid x \in X, y \in Y\}$.

   (a) Describe an analyze and algorithm to compute the number of elements in $X + Y$ in $O(n^2 \log n)$ time. *[Hint: The answer is **not** always n^2.]*

   (b) Describe and analyze an algorithm to compute the number of elements in $X + Y$ in $O(M \log M)$ time, where M is the largest absolute value of any element of $X \cup Y$. *[Hint: What's this lecture about?]*

2. Suppose we are given a bit string $B[1..n]$. A triple of distinct indices $1 \le i < j < k \le n$ is called a ***well-spaced triple*** in B if $B[i] = B[j] = B[k] = 1$ and $k - j = j - i$.

   (a) Describe a brute-force algorithm to determine whether B has a well-spaced triple in $O(n^2)$ time.

   (b) Describe an algorithm to determine whether B has a well-spaced triple in $O(n \log n)$ time. *[Hint: Hint.]*

   (c) Describe an algorithm to determine the *number* of well-spaced triples in B in $O(n \log n)$ time.

3. (a) Describe an algorithm that determines whether a given set of n integers contains two elements whose sum is zero, in $O(n \log n)$ time.

   (b) Describe an algorithm that determines whether a given set of n integers contains *three* elements whose sum is zero, in $O(n^2)$ time.

   (c) Now suppose the input set X contains only integers between $-10000n$ and $10000n$. Describe an algorithm that determines whether X contains three elements whose sum is zero, in $O(n \log n)$ time. *[Hint: Hint.]*

4. Describe an algorithm that applies the bit-reversal permutation to an array $A[1..n]$ in $O(n)$ time when n is a power of 2.

`BIT-`	$\Longleftrightarrow$	`BTI-`
`REVERSAL`	$\Longleftrightarrow$	`RRVAESEL`
`BUTTERFLYNETWORK`	$\Longleftrightarrow$	`BYEWTEFRUNROTTLK`
`THISISTHEBITREVERSALPERMUTATION!`	$\Longleftrightarrow$	`TREUIPRIIAIATRVNHSBTSEEOSLTTHME!`

5. The FFT algorithm we described in this lecture is limited to polynomials with 2^k coefficients for some integer k. Of course, we can always pad the coefficient vector with zeros to force it into this form, but this padding artificially inflates the input size, leading to a slower algorithm than necessary.

   Describe and analyze a similar DFT algorithm that works for polynomials with 3^k coefficients, by splitting the coefficient vector into three smaller vectors of length 3^{k-1}, recursively computing the DFT of each smaller vector, and correctly combining the results.

6. Fix an integer k. For any two k-bit integers i and j, let $\boldsymbol{i \wedge j}$ denote their bitwise AND, and let $\boldsymbol{\Sigma(i)}$ denote the number of 1s in the binary expansion of i. For example, when $k = 4$, we have $10 \wedge 7 = \texttt{1010} \wedge \texttt{0111} = \texttt{0010} = 2$ and $\Sigma(7) = \Sigma(\texttt{0111}) = 3$.

   The kth ***Sylvester-Hadamard matrix*** $\boldsymbol{H_k}$ is a $2^k \times 2^k$ matrix indexed by k-bit integers, each of whose entries is either $+1$ or -1, defined as follows:

$$H_k[i, j] = (-1)^{\Sigma(i \wedge j)}$$

   For example:

$$H_3 = \begin{bmatrix} +1 & +1 & +1 & +1 & +1 & +1 & +1 & +1 \\ +1 & -1 & +1 & -1 & +1 & -1 & +1 & -1 \\ +1 & +1 & -1 & -1 & +1 & +1 & -1 & -1 \\ +1 & -1 & -1 & +1 & +1 & -1 & -1 & +1 \\ +1 & +1 & +1 & +1 & -1 & -1 & -1 & -1 \\ +1 & -1 & +1 & -1 & -1 & +1 & -1 & +1 \\ +1 & +1 & -1 & -1 & -1 & -1 & +1 & +1 \\ +1 & -1 & -1 & +1 & -1 & +1 & +1 & -1 \end{bmatrix}$$

   (a) Prove that the matrix H_k can be decomposed into four copies of H_{k-1} as follows:

$$H_k = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix}$$

   (b) Prove that $H_k \cdot H_k = 2^k \cdot I_k$, where I is the $2^k \times 2^k$ identity matrix.

(c) For any vector $\vec{x} \in \mathbb{R}^{2^k}$, the product $H_k\vec{x}$ is called the ***Walsh-Hadamard transform*** of $\vec{x}$. Describe an algorithm to compute the Walsh-Hadamard transform in $O(n \log n)$ time, given the integer k and a vector of $n = 2^k$ integers as input.

♥7. The ***discrete Hartley transform*** of a vector $\vec{x} = (x_0, x_1, \ldots, x_{n-1})$ is another vector $\vec{X} = (X_0, X_1, \ldots, X_{n-1})$ defined as follows:

$$X_j = \sum_{i=0}^{n-1} x_i \left(\cos\left(\frac{2\pi}{n} ij\right) + \sin\left(\frac{2\pi}{n} ij\right)\right)$$

Describe an algorithm to compute the discrete Hartley transform in $O(n \log n)$ time when n is a power of 2.

8. Suppose n is a power of 2. Prove that if we recursively apply the FACTORFFT algorithm, factoring of n into *arbitrary* smaller powers of two, the resulting algorithm still runs in $O(n \log n)$ time.

9. Let $f : \mathbb{Z} \to \mathbb{Z}$ be *any* function such that $2 \le f(n) \le \sqrt{n}$ for all $n \ge 4$. Prove that the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n \le 4 \\ f(n) \cdot T\left(\dfrac{n}{f(n)}\right) + \dfrac{n}{f(n)} \cdot T(f(n)) + O(n) & \text{otherwise} \end{cases}$$

has the solution $T(n) = O(n \log n)$. For example, setting $f(n) = 2$ for all n gives us the standard mergesort/FFT recurrence.

10. Although the radix-2 FFT algorithm and the more general recursive factoring algorithm both run in $O(n \log n)$ time, the latter algorithm is more efficient in practice (when n is large) due to caching effects.

Consider an idealized two-level memory model, which has an arbitrarily large but slow *main memory* and a significantly faster *cache* that holds the C most recently accessed memory items. The running time of an algorithm in this model is dominated by the number of *cache misses*, when the algorithm needs to access a value in memory that it not stored in the cache.

The number of cache misses seen by the radix-2 FFT algorithm obeys the following recurrence:

$$M(n) \le \begin{cases} 2M(n/2) + O(n) & \text{if } n > C \\ O(n) & \text{if } n \le C \end{cases}$$

If the input array is too large to fit in the cache, then *every* memory access in both the initial and final for-loops will cause a cache miss, and the recursive calls will cause their own cache misses. But if the input array is small enough to fit in cache, the

initial for-loop loads the input into the cache, but there are no more cache misses, even inside the recursive calls.

(a) Solve the previous recurrence for $M(n)$, as a function of both n and C. To simplify the analysis, assume both n and C are powers of 2.

(b) Suppose we always recursively call FactorFFT with $p = C$ and $q = n/C$. How many cache misses does the resulting algorithm see, as a function of n and C? To simplify the analysis, assume $n = C^k$ for some integer k. (In particular, assume C is a power of 2.)

(c) Unfortunately, it is not always possible for a program (or a compiler) to determine the size of the cache. Suppose we always recursively call FactorFFT with $p = q = \sqrt{n}$. How many cache misses does the resulting algorithm see, as a function of n and C? To simplify the analysis, assume $n = C^{2^k}$ for some integer k.

© 2018 Jeff Erickson

Wouldn't the sentence "I want to put a hyphen between the words Fish and And and And and Chips in my Fish-And-Chips sign." have been clearer if quotation marks had been placed before Fish, and between Fish and and, and and and And, and And and and, and and and And, and And and and, and and and Chips, as well as after Chips?[1]

— Martin Gardner, *Aha! Insight* (1978)

BMO: *I finished with my computations! As I thought, it's a real language they are speaking based on intonations! [speaking to Chips and Ice Cream] Chips? Chips chips chips! Ice cream?*
Chips: *Chips chips chips!*
Ice Cream: *Ice cream ice cream!*

— "Chips and Ice Cream", *Adventure Time* season 6, episode 34 (April 30, 2015)

For every polynomial-time algorithm you have,
there is an exponential algorithm that I would rather run.

— Alan Perlis (first Turing Award winner)

B

Faster Exponential Algorithms

[Read Chapters 2, 3, and 12 first.]
Status: Unfinished.

There are many useful problems—including but not limited to the set of all NP-hard problems—for which the fastest algorithm known runs in exponential time. Moreover, there is *very* strong evidence (but alas, no proof) that it is *impossible* to solve these problems in less than exponential time—it's not that we're all stupid; the problems really are that hard!

But sometimes we have to solve these problems anyway. In these circumstances, it is important to ask: *Which* exponential? An algorithm that runs in $O(2^n)$ time, while still

[0] If you ever decide to read this sentence out loud, be sure to pause briefly between 'Fish and and' and 'and and and And', 'and and and And' and 'and And and and', 'and And and and' and 'and and and And', 'and and and And' and 'and And and and', and 'and And and and' and 'and and and Chips'!

Did you notice the punctuation I carefully inserted between 'Fish and and' and 'and', 'and' and 'and and and And', 'and and and And' and 'and and and And', 'and and and And' and 'and', 'and' and 'and And and and', 'and And and and' and 'and And and and', 'and And and and' and 'and', 'and' and 'and and and And', 'and and and And' and 'and and and And', 'and and and And' and 'and', 'and' and 'and And and and', 'and And and and' and 'and', 'and' and 'and And and and', 'and And and and' and 'and', and 'and' and 'and and and Chips'?

© 2018 Jeff Erickson CC BY NC SA http://algorithms.wtf

unusable for large instances, is still significantly better than an algorithm that runs in $O(4^n)$ time; an algorithm that runs in $O(1.5^n)$ or $O(1.25^n)$ is better still.

The most straightforward method to find an optimal solution to an NP-hard problem to recursively generate *all* possible solutions via recursive backtracking and check each one: all satisfying assignments, or all vertex colorings, or all subsets, or all permutations, or whatever. However, many (perhaps even most) NP-hard problems have some additional structure that allows us to prune away most of the branches of the recursion tree, thereby drastically reducing the running time.

B.1 3Sat

Let's consider the mother of all NP-hard problems: 3Sat. Given a boolean formula in conjunctive normal form, with at most three literals in each clause, our task is to determine whether any assignment of values of the variables makes the formula true. Yes, this problem is NP-hard, which means that a polynomial algorithm is almost certainly impossible. Too bad; we have to solve the problem anyway.

The trivial solution is to try every possible assignment. We'll evaluate the running time of our 3Sat algorithms in terms of two variables: n is the number of variables in the input formula, and m is the number of clauses. There are 2^n possible assignments, and we can evaluate each assignment in $O(m)$ time, so the overall running time is $O(2^n m)$. Assuming no clause appears in the input formula more than once—a condition that we can easily enforce in polynomial time—then $m = O(n^3)$, so the running time is at most $O(2^n n^3)$.

Polynomial factors like n^3 are essentially noise when the overall running time is exponential, so from now on I'll write $\text{poly}(n)$ to represent any polynomial function of n; in other words, $\text{poly}(n) := n^{O(1)}$. For example, the trivial algorithm for 3Sat runs in time $O(2^n \text{poly}(n))$.

One Clause

Recall that all 3CNF formulas have the following recursive structure

A 3CNF formula is either nothing or (a clause with three literals) $\wedge$ (a 3CNF formula)

Suppose we want to decide whether some 3CNF formula Φ with n variables is satisfiable. Of course this is trivial if Φ is the empty formula, so suppose

$$\Phi = (x \vee y \vee z) \wedge \Phi'$$

for some literals x, y, z and some 3CNF formula Φ'. By distributing the $\wedge$ across the $\vee$s, we can rewrite Φ as follows:

$$\Phi = (x \wedge \Phi') \vee (y \wedge \Phi') \vee (z \wedge \Phi')$$

For any boolean formula Ψ and any literal x, let $\boldsymbol{\Psi|x}$ (pronounced "sigh given eks") denote the simpler boolean formula obtained by setting x to TRUE and simplifying as much as possible. If Ψ has n variables, then $\boldsymbol{\Psi|x}$ has at most $n-1$ variables.

It's not hard to prove by induction (hint, hint) that $x \wedge \Psi = x \wedge (\Psi|x)$, and therefore

$$\Phi = (x \wedge (\Phi'|x)) \ \vee \ (y \wedge (\Phi'|y)) \ \vee \ (z \wedge (\Phi'|z)).$$

Thus, for any satisfying assignment for Φ, either

- x is true and $\Phi'|x$ is satisfiable, or
- y is true and $\Phi'|y$ is satisfiable, or
- z is true and $\Phi'|z$ is satisfiable.

Each of the smaller formulas $\Phi'|x$, $\Phi'|y$, and $\Phi'|z$ has at most $n-1$ variables. If we recursively check all three possibilities, we get the running time recurrence

$$T(n) \leq 3T(n-1) + \text{poly}(n),$$

whose solution is $O(3^n \text{poly}(n))$. So we've actually done *worse!*

But these three recursive cases are not mutually exclusive! If $\Phi'|x$ is *not* satisfiable, then x *must* be false in any satisfying assignment for Φ. So instead of recursively checking $\Phi'|y$ in the second step, we can check the even simpler formula $\Phi'|\bar{x}y = (\Phi'|\bar{x})|y$. Similarly, if $\Phi'|\bar{x}y$ is not satisfiable, then we know that y must be false in any satisfying assignment, so we can recursively check $\Phi'|\bar{x}\bar{y}z$ in the third step.

```
3SAT(Φ):
  if Φ = ∅
      return TRUE
  (x ∨ y ∨ z) ∧ Φ' ← Φ
  if 3SAT(Φ'|x)
      return TRUE
  if 3SAT(Φ'|x̄y)
      return TRUE
  return 3SAT(Φ'|x̄ȳz)
```

The running time off this algorithm obeys the recurrence

$$T(n) = T(n-1) + T(n-2) + T(n-3) + \text{poly}(n),$$

where $\text{poly}(n)$ denotes the polynomial time required to simplify boolean formulas, handle control flow, move stuff into and out of the recursion stack, and so on.

How do we solve this recurrence? It isn't simple enough to let us guess an accurate solution and check it by induction. Fortunately, there is a standard mechanical solution for all backtracking recurrences of the form

$$T(n) \leq \text{poly}(n) + \sum_{i=1}^{k} T(n-a_i),$$

where $k, a_1, a_2, \ldots, a_k$ are constants. Assume without loss of generality that the constants a_i are indexed in non-increasing order: $a_1 \le a_2 \le \cdots \le a_k$. The fundamental theorem of algebra implies that the ***characteristic polynomial***

$$r^{a_k} - \sum_{i=1}^{k} r^{a_k - a_i}$$

(where r is a formal variable) has exactly k complex roots, some of which may be equal. Let λ be the root with largest magnitude. Then the solution to the recurrence is $\boldsymbol{T(n) = O(\lambda^n \operatorname{poly}(n))}$. This technique is described in more detail in the appendix; in particular, we can actually derived *exact closed form* solutions to backtracking-style recurrences from the complete sequence of characteristic roots and base cases, if the polynomial terms and base cases are specified exactly.

The recurrence $T(n) = T(n-1) + T(n-2) + T(n-3) + \operatorname{poly}(n)$ has characteristic polynomial $r^3 - r^2 - r - 1$, whose roots are approximately

$$1.83928676, \quad -0.419643337 + 0.60629073i, \quad -0.419643337 - 0.60629073i.$$

(Thank you, Wolfram Alpha!) The first root has larger magnitude than the other two, so we get the solution

$$T(n) = O(\lambda^n \operatorname{poly}(n)) = \boldsymbol{O(1.83929^n)}.$$

(Notice that we cleverly eliminated the polynomial noise by ever so slightly increasing the base of the exponent.)

Pure Literals

We can improve this algorithm further by eliminating *pure* literals from the formula before recursing. A literal x is ***pure*** in if it appears in the formula Φ but its negation $\bar{x}$ does not. It's not hard to prove (hint, hint) that if Φ has a satisfying assignment, then it has a satisfying assignment where every pure literal is true. If $\Phi = (x \vee y \vee z) \wedge \Phi'$ has no pure literals, then some clause in Φ contains the literal $\bar{x}$, so we can write

$$\Phi = (x \vee y \vee z) \wedge (\bar{x} \vee u \vee v) \wedge \Phi'$$

for some literals u and v (each of which might be y, $\bar{y}$, z, or $\bar{z}$). It follows that the first recursive formula $\Phi|x$ contains the clause $(u \vee v)$. We can recursively eliminate the variables u and v just as we eliminated the variables y and x in the second and third cases of our previous algorithm:

$$\Phi|x = (u \vee v) \wedge \Phi'|x = (u \wedge \Phi'|xu) \vee (v \wedge \Phi'|x\bar{u}v).$$

Our new faster algorithm is shown in Figure **??**. The running time $T(n)$ of this algorithm satisfies the recurrence

$$T(n) = \operatorname{poly}(n) + \max \begin{Bmatrix} T(n-1) \\ 2T(n-2) + 2T(n-3) \end{Bmatrix}$$

```
FASTER3SAT(Φ):
  if Φ = ∅
      return TRUE
  if Φ has a pure literal x
      return FASTER3SAT(Φ|x)
  (x ∨ y ∨ z) ∧ (x̄ ∨ u ∨ v) ∧ Φ′ ← Φ
  if FASTER3SAT(Φ|xu)
      return TRUE
  if FASTER3SAT(Φ|xūv)
      return TRUE
  if FASTER3SAT(Φ|x̄y)
      return TRUE
  return FASTER3SAT(Φ|x̄ȳz)
```

Figure B.1. A faster backtracking algorithm for 3SAT.

To solve this recurrence, we solve each case separately and take the larger of the two solutions. Here, the comparison is easy; the first case yields a *polynomial* bound, while the second case clearly yields an exponential bound, so for purposes of worst-case analysis, we can simplify the recurrence to $T(n) = O(\text{poly}(n)) + 2T(n-2) + 2T(n-3)$. The solution to this simpler recurrence is

$$T(n) = O(\lambda^n \text{ poly}(n)) = \boldsymbol{O(1.76930^n)},$$

where $\lambda \approx 1.76929235$ is the largest root of the characteristic polynomial $r^3 - 2r - 2$.

Local Search

We can get an even faster algorithm using a different backtracking strategy, called ***local search***. Instead of constructing a satisfying assignment recursively, we search for a path through the set of all assignments that ends at a satisfying assignment.

The ***Hamming distance*** between two different assignments is the number of variables where the two assignments differ.[2] Suppose we could somehow magically find an assignment that is *almost* satisfying, meaning that its Hamming distance from some satisfying assignment is small. Then instead of searching over the entire space of 2^n assignments, we only need to look at the assignments that are close to the assignment we already have. Moreover, we can guide the search by considering only the clauses that are *not* satisfied by our current assignment.

To make this concrete, let A be our current assignment, and suppose there is a satisfying assignment A^* that has Hamming distance r from A. Consider any clause of Φ

[2]Richard Hamming was an early giant of computer and information science, who earned PhD in mathematics at Illinois in 1942, programmed IBM calculating machines for the Manhattan project in 1945, and then went on to create the first error-correcting codes. Arguably the greatest recognition of his achievements is that his name is attached to the number of indices where two bit vectors differ.

that is *not* satisfied by A; at least one the three variables in that clause has a different value in A^*. Thus, if we change that particular variable's value, the resulting assignment has Hamming distance $r-1$ from A^*. Of course, we don't know which variable to change, so we have to try all three of them. And we don't know the target assignment A^*, so we just check at every stage of recursion whether our current assignment is satisfying. The resulting backtracking algorithm, shown in Figure **??**, runs in $O(3^r \operatorname{poly}(n))$ time.

```
Local3Sat(Φ, A, r):
  if A satisfies Φ
      return True
  if r = 0
      return False
  C ← any clause of Φ that A does not satisfy
  x, y, z ← variables of C
  A[x] ← ¬A[x]
  if Local3Sat(Φ, A, r − 1) = True
      return True
  A[x] ← ¬A[x]
  A[y] ← ¬A[y]
  if Local3Sat(Φ, A_y, r − 1) = True
      return True
  A[y] ← ¬A[y]
  A[z] ← ¬A[z]
  return Local3Sat(Φ, A_z, r − 1)
```

Figure B.2. A local search algorithm for 3Sat.

It must be emphasized that the local search algorithm is *not* explicitly considering every assignment within Hamming distance r of the starting assignment A.

Now we apply the following trivial observation: In any satisfying truth assignment, either at least half the variables are True or at least half the variables are False. Thus, *every* satisfying assignment has Hamming distance at most $n/2$ from either the all-True assignment or the all-False assignment. So using the local search strategy, we can solve 3Sat in $\boldsymbol{O(3^{n/2}\operatorname{poly}(n)) = O(1.73206^n)}$ ***time***.

♥Random Starting Assignments

We can improve the local search algorithm further, at the expense of a small probability of a false negative, by repeatedly choosing *random* starting assignments and searching within a small radius of each.

First let's do some preliminary math. Let $\boldsymbol{V(r)}$ denote the number of assignments within Hamming distance r of any fixed assignment. Suppose we run the local search algorithm starting with a random assignment; let $P(r)$ denote the probability that the algorithm finds a satisfying assignment (assuming one exists). A simple counting

argument implies that $\boldsymbol{P(r) \geq V(r)/2^n}$. If we choose $N(r)$ random assignments and run the local search algorithm starting at each one, the probability of *not* finding a satisfying assignment (if one exists) is *at most* $(1-P(r))^{N(r)}$. Thus, if we set $\boldsymbol{N(r) = n2^n/V(r)}$, this failure probability is at most

$$(1-P(r))^{n2^n/V(r)} \leq \left(\left(1-\frac{V(r)}{2^n}\right)^{2^n/V(r)}\right)^n \leq e^{-n}.$$

The overall running time of our algorithm is

$$O(N(r)\,3^r \operatorname{poly}(n)) \;=\; O\left(\frac{n}{P(r)} \cdot 3^r \operatorname{poly}(n)\right) \;=\; O\left(\frac{2^n 3^r}{V(r)} \operatorname{poly}(n)\right).$$

To complete the analysis, we need to estimate the volume $V(r)$ and choose an appropriate value of r. For any constant $0 < \alpha < 1$, Stirling's approximation $n! \sim \sqrt{2\pi n}\,(n/e)^n$ implies the lower bound

$$V(\alpha n) \;=\; \sum_{i=0}^{r} \binom{n}{i} \;=\; \Omega\big(H(\alpha)^n/\sqrt{n}\big)$$

where $H(\alpha)$ is the *binary entropy* function:

$$H(\alpha) = \left(\frac{1}{\alpha}\right)^{\alpha} \left(\frac{1}{1-\alpha}\right)^{1-\alpha}.$$

Thus, if we set $r = \alpha n$, the running time of our algorithm becomes

$$O\left(\frac{2^n 3^{\alpha n}}{H(\alpha)^n} \operatorname{poly}(n)\right) \;=\; O\Big(\big(2(3\alpha)^{\alpha}(1-\alpha)^{1-\alpha}\big)^n \operatorname{poly}(n)\Big)$$

We can find the value of α that minimizes the base of the exponential part of this upper bound using a bit of calculus.

$$\begin{aligned}
\frac{d}{d\alpha} \ln\big((3\alpha)^{\alpha}(1-\alpha)^{1-\alpha}\big) &= \frac{d}{d\alpha}\big(\alpha \ln(3\alpha) + (1-\alpha)\ln(1-\alpha)\big) \\
&= \ln(3\alpha) - \ln(1-\alpha) = 0 \\
\implies \alpha &= 1/4.
\end{aligned}$$

Finally, if we set $\alpha = 1/4$ (or equivalently, $r = n/4$), the algorithm runs in

$$O\Big(\big(2(3/4)^{1/4}(3/4)^{3/4}\big)^n \operatorname{poly}(n)\Big) \;=\; \boldsymbol{O\big((3/2)^n \operatorname{poly}(n)\big)}\ \textbf{\textit{time}}.$$

Further Extensions

Naturally, these approaches can be extended much further. Since 1998, at least fifteen different 3Sat algorithms have been published, each improving the running time by a small amount; while a few of these improvements use pure backtracking, most of them are based on some variant of local search. For example, in 1999, Uwe Schöning the running time of the randomized local search algorithm to $O((4/3)^n \operatorname{poly}(n))$, still with an exponentially small error probability, by replacing the backtracking search with a random walk of length $3n$. As of 2016, the fastest deterministic algorithm for 3Sat runs in $O(1.3303^n)$ time[3], and the fastest *randomized* algorithm runs in $O(1.30704^n)$ expected time[4], but there is good reason to believe that these are *not* the best possible.

There is a thriving industry of practical Sat solvers, which combine backtracking, local search, random restarts, and other more sophisticated algorithmic techniques with domain-specific heuristics. With these tools in hand, most real-world instances of Sat are actually easy to solve in practice; modern Sat solvers routinely resolve instances with millions of variables and clauses. Sat solvers have become an indispensable tool in several areas of computing, including hardware and software verification, operations research, and computational biology. The practically of Sat does not contradict the fact that the fastest algorithms require exponential time *in the worst case*; it just means that *the worst case* is rare in practice. Unfortunately, a proper discussion of practical techniques for Sat is beyond both the scope of this text and the expertise of its author.

B.2 Maximum Independent Set

Now suppose we are given an undirected graph G and are asked to find the size of the *largest independent set*, that is, the largest subset of the vertices of G with no edges between them. Once again, we have an obvious recursive algorithm: Try every subset of nodes, and return the largest subset with no edges. Expressed recursively, we obtain the following algorithm, where $N(v)$ denotes the *neighborhood* of v: the set containing v and all of its neighbors.

```
MaximumIndSetSize(G):
  if G = ∅
      return 0
  else
      v ← any node in G
      withv ← 1 + MaximumIndSetSize(G \ N(v))
      withoutv ← MaximumIndSetSize(G \ {v})
      return max{withv, withoutv}.
```

[3]Kazuhisa Makino, Suguru Tamaki, Masaki Yamamoto. Derandomizing the HSSW algorithm for 3-SAT. *Algorithmica* 67(2):112–124 (2013)

[4]Timon Hertli. 3-SAT faster and simpler: Unique-SAT Bounds for PPSZ gold in general. *SIAM J. Comput.* 43(2):718–729 (2014).

Our algorithm exploits the fact that if an independent set contains v, then by definition it contains none of v's neighbors. In the worst case, v has no neighbors, so $G \setminus \{v\} = G \setminus N(v)$. Thus, the running time of this algorithm satisfies the recurrence $T(n) = 2T(n-1) + \text{poly}(n) = O(2^n \text{poly}(n))$. Surprise, surprise.

This algorithm is mirroring a crude recursive upper bound for the number of *maximal* independent sets in a graph; an independent set is maximal if every vertex in G is either already in the set or a neighbor of a vertex in the set. If the graph is non-empty, then every maximal independent set either includes or excludes each vertex. Thus, the number of maximal independent sets satisfies the recurrence $M(n) \leq 2M(n-1)$, with base case $M(1) = 1$. We can easily guess and confirm the solution $M(n) \leq 2^n - 1$. The only subset that we aren't counting with this upper bound is the empty set!

Smarter Case Analysis

We can speed up our algorithm by making several careful modifications to avoid the worst case of the running-time recurrence.

Degree 0. If any vertex v has no neighbors, then $N(v) = \{v\}$, and both recursive calls consider a graph with $n-1$ nodes. But in this case, v is in *every* maximal independent set, so one of the recursive calls is redundant. On the other hand, if v has at least one neighbor, then $G \setminus N(v)$ has at most $n-2$ nodes. So now we have the following recurrence.

$$\begin{aligned} T(n) &\leq \text{poly}(n) + \max \left\{ \begin{array}{c} T(n-1) \\ \boldsymbol{T(n-1) + T(n-2)} \end{array} \right\} \\ &= O(1.61804^n) \end{aligned}$$

As before, the upper bound is derived by solving each case separately using characteristic polynomials and taking the larger of the two solutions. The first case gives us $T(n) = \text{poly}(n)$; the second case yields our old friends the Fibonacci numbers. Here and in later multi-case recurrences, I've typeset the worst case of the recurrence in **bold**.

Degree 1. We can improve this bound even more by examining the new worst case: some vertex v has exactly one neighbor w. In this case, either v or w appears in every maximal independent set. However, given any independent set that includes w, removing w and adding v creates another independent set of the same size. It follows that *some maximum independent set includes* v, so we don't need to search the graph $G \setminus \{v\}$, and the $G \setminus N(v)$ has at most $n-2$ nodes. On the other hand, if the degree of v is at least 2, then $G \setminus N(v)$ has at most $n-3$ nodes.

$$
\begin{aligned}
T(n) &\le \text{poly}(n) + \max\left\{\begin{array}{c} T(n-1) \\ T(n-2) \\ \boldsymbol{T(n-1)+T(n-3)} \end{array}\right\} \\
&= O(1.46558^n)
\end{aligned}
$$

The base of the exponent is the largest root of the characteristic polynomial $r^3 - r^2 - 1$.

Degree $\geq$ 3. Now the worst-case is a graph where every node has degree at least 2; we split this worst case into two subcases. If G has a node v with degree 3 or more, then $G \setminus N(v)$ has at most $n-4$ nodes. Otherwise (since we have already considered nodes of degree 0 and 1), *every* node in G has degree 2. Let u, v, w be a path of three nodes in G (possibly with u adjacent to w). In any maximal independent set, either v is present and u, w are absent, or u is present and its two neighbors are absent, or w is present and its two neighbors are absent. In all three cases, we recursively count maximal independent sets in a graph with $n-3$ nodes.

$$
\begin{aligned}
T(n) &\le \text{poly}(n) + \max\left\{\begin{array}{c} T(n-1) \\ T(n-2) \\ T(n-1)+T(n-4) \\ \boldsymbol{3T(n-3)} \end{array}\right\} \\
&= O(3^{n/3}\,\text{poly}(n)) \\
&= O(1.44225^n)
\end{aligned}
$$

The base of the exponent is $\sqrt[3]{3}$, the largest root of the characteristic polynomial $r^3 - 3$. The third case would give us a bound of $O(1.38028^n)$, where the base is the largest root of the characteristic polynomial $r^4 - r^3 - 1$.

Degree 2. Now the worst case for our algorithm is a graph with an extraordinarily special structure: *Every node has degree 2*. In other words, every component of G is a cycle. It is easy to prove that the largest independent set in a cycle of length k has size $\lfloor k/2 \rfloor$. So we can handle this case directly in polynomial time, with no recursion at all!

$$
\begin{aligned}
T(n) &\le \text{poly}(n) + \max\left\{\begin{array}{c} T(n-1) \\ T(n-2) \\ \boldsymbol{T(n-1)+T(n-4)} \end{array}\right\} \\
&= O(1.38028^n)
\end{aligned}
$$

Again, the base of the exponential running time is the largest root of the characteristic polynomial $r^4 - r^3 - 1$. The final algorithm after all these improvements is shown in Figure **??**.

```
MaximumIndSetSize(G):
  if G = ∅
      return 0
  else if G has a node v with degree 0 or 1
      return 1 + MaximumIndSetSize(G \ N(v))        ⟨⟨≤ n − 1⟩⟩
  else if G has a node v with degree greater than 2
      withv ← 1 + MaximumIndSetSize(G \ N(v))       ⟨⟨≤ n − 4⟩⟩
      withoutv ← MaximumIndSetSize(G \ {v})         ⟨⟨≤ n − 1⟩⟩
      return max{withv, withoutv}
  else ⟨⟨every node in G has degree 2⟩⟩
      total ← 0
      for each component of G
          k ← number of vertices in the component
          total ← total + ⌊k/2⌋
      return total
```

Figure B.3. A fast backtracking algorithm for MaxIndependentSet

Further Improvements

As with 3Sat, there are faster but increasingly complex algorithms for MaxIndependent-Set. For example, in 1977, Bob Tarjan and Anthony Trojanowski published a complex backtracking algorithm that considers several dozen cases, each with its own running time recurrence. Most of these cases are clearly dominated by others; for example, even without solving the recurrences, it's easy to see that the recurrence $T(n) \leq \text{poly}(n) + T(n-4) + T(n-5)$ has a smaller solution than $T(n) \leq \text{poly}(n) + T(n-3) + T(n-5)$. After discarding all such dominated cases, Tarjan and Trojanowski are left with the rather impressive recurrence shown in Figure **??**. The solution to this recurrence is $T(n) = O(1.25603^2) = O(2^{n/3})$, where the base of the exponent is the largest root of the characteristic polynomial $r^{10} - r^8 - r^2 - 2$ of the third case (in bold).

As of 2016, the fastest published algorithm for computing maximum independent sets runs in $O(1.2114^n)$ time[5]. However, in an unpublished technical report, Mike Robson describes a *computer-generated* algorithm that runs in $O(2^{n/4}\,\text{poly}(n)) = O(1.1889^n)$ time; just the description of this algorithm requires more than 15 pages.[6]

B.3 Dynamic Programming

Warmup: Bellman-Held-Karp TSP in $O(2^n\,\text{poly}(n))$ time. (Leave as exercise?) ★★★

★★★

[5]Nicolas Bourgeois, Bruno Escoffier, Vangelis Paschos, and Johan M. M. can Rooij. Fast algorithms for max independent set. *Algorithmica* 62(1):382–415, 2010.

[6]John Michael Robson. Finding a maximum independent set in time $O(2^{n/4})$. Technical report 1251-01, LaBRI, 2001. ⟨http://www.labri.fr/perso/robson/mis/techrep.ps⟩.

$$T(n) \le \text{poly}(n) + \max \left\{ \begin{array}{c} T(n-2)+T(n-6) \\ T(n-2)+2T(n-8) \\ \boldsymbol{T(n-2)+T(n-8)+2T(n-10)} \\ T(n-3)+T(n-5) \\ T(n-4)+T(n-6)+T(n-8) \\ T(n-4)+2T(n-7) \\ T(n-4)+T(n-8)+3T(n-11) \\ T(n-4)+2T(n-8)+T(n-10) \\ T(n-4)+T(n-9)+2T(n-12) \\ T(n-4)+4T(n-10) \\ T(n-5)+4T(n-9) \\ T(n-5)+6T(n-11)+4T(n-14)+T(n-17) \end{array} \right\}$$

Figure B.4. The (simplified!) running time recurrence for Tarjan and Trojanowski's independent set algorithm.

3-coloring in $O(2^n \text{ poly}(n))$ time. Let $OPT(X) =$ True if there is a 3-coloring in which every vertex in X is red. Then

$$OPT(X) = \begin{cases} \text{False} & \text{if } X \text{ is not an independent set} \\ \text{True} & \text{if } G[V \setminus X] \text{ is bipartite} \\ \bigvee_{v \in V \setminus X} OPT(X \cup \{v\}) & \text{otherwise} \end{cases}$$

Lawlar improved the running time to $O(3^{n/3} \text{ poly}(n))$ by observing that we only need to care about $OPT(X)$ when X is a *maximal* independent set. There are $O(3^{n/3})$ maximal independent sets, which can be listed in $O(3^{n/3} \text{ poly}(n))$ time.

Alternatively, 3-coloring in $O(2^n \text{ poly}(n))$ time by backtracking through the vertices in some fixed whatever-first order. Each vertex must be colored differently from is parent (and possibly other vertices) and without loss of generality, the first tow nodes are red and green, respectively. So we consider only 2^{n-2} possible colorings, but memoization doesn't help. (Can we do better by looking in max-adjacency order?)

★★★

Lawler's algorithm for chromatic number in $O((1+3^{1/3})^n \text{ poly } n)$ time: Let $OPT(X)$ denote the chromatic number of the subgraph $G[X]$ induced by the vertex subset $X \subseteq V$. Then

$$OPT(X) = \begin{cases} 0 & \text{if } X = \varnothing \\ 1 + \min \left\{ OPT(X \setminus I) \;\middle|\; \text{maximal ind. set } I \text{ in } G[X] \right\} & \text{otherwise} \end{cases}$$

There are $O(3^{k/3})$ maximal independent sets of size k, which can be listed in $O(3^{k/3} \text{ poly}(n))$ time.

★★★

Other techniques to include?

- Horowitz-Sahni partitioning to solve SubsetSum in $O(2^{n/2}\,\mathrm{poly}(n))$ time? (See also: 3SUM)
- Decision via counting via Inclusion-Exclusion?

Exercises

1. (a) Prove that any n-vertex graph has at most $3^{n/3}$ maximal independent sets. *[Hint: Modify the MaximumIndSetSize algorithm so that it lists all maximal independent sets.]*

   (b) Describe an n-vertex graph with exactly $3^{n/3}$ maximal independent sets, for every integer n that is a multiple of 3.

2. (a) Describe an algorithm for kSat whose running time satisfies the recurrence

$$T(n) = 2T(n-1) - T(n-k+1) + \mathrm{poly}(n).$$

   (b) Describe an algorithm for kSat whose running time satisfies the recurrence

$$T(n) = T(n-1) + 2T(n-2) - 2T(n-k+1) + \mathrm{poly}(n).$$

   [Hint: Modify the backtracking algorithms for 3Sat described in the text. No, your algorithm will not make negative recursive calls.]

♥3. Describe an algorithm that solves 3Sat in $O(\phi^n\,\mathrm{poly}(n))$ time, where $\phi = (1+\sqrt{5})/2 \approx 1.618034$. *[Hint: Prove that in each recursive call, either you have just eliminated a pure literal, or the formula has a clause with at most two literals. What recurrence leads to this running time?]*

4. Describe an algorithm to determine whether a graph is 4-colorable in $O(2^n\,\mathrm{poly}(n))$ time. *[Hint: Consider red and green together, but separately from blue and yellow.]*

5. (a) Prove the following upper bound for all constant $0 < \alpha < 1$:

$$\binom{n}{\alpha n} \le \sum_{i=0}^{\alpha n} \binom{n}{i} < \left(\left(\frac{1}{\alpha}\right)^{\alpha}\left(\frac{1}{1-\alpha}\right)^{1-\alpha}\right)^n$$

   [Hint: Expand the expression $(\alpha + (1-\alpha))^n = 1$ using the binomial theorem.]

   (b) Prove the following lower bound for all constant $0 < \alpha < 1$:

$$\binom{n}{\alpha n} \ge \left(\left(\frac{1}{\alpha}\right)^{\alpha}\left(\frac{1}{1-\alpha}\right)^{1-\alpha}\right)^n \cdot \Omega(1/\sqrt{n})$$

   [Hint: Use Stirling's approximation: $\sqrt{2\pi n}\,(n/e)^n \le n! \le 2\sqrt{\pi n}\,(n/e)^n$.]

Need a few more exercises.

★★★

© 2018 Jeff Erickson http://algorithms.wtf

Those who cannot remember the past are condemned to repeat it.

— Jorge Agustín Nicolás Ruiz de Santayana y Borrás, *The Life of Reason, Book I: Introduction and Reason in Common Sense* (1905)

Those who cannot remember the past are condemned to repeat it.

— Jorge Agustín Nicolás Ruiz de Santayana y Borrás, *The Life of Reason, Book I: Introduction and Reason in Common Sense* (1905)

Those who cannot remember the past are condemned to repeat it.

— Jorge Agustín Nicolás Ruiz de Santayana y Borrás, *The Life of Reason, Book I: Introduction and Reason in Common Sense* (1905)

C

Dynamic Programming for Formal Languages and Automata

[Read Chapter 3 first.]

Status: Unfinished

This chapter describes dynamic programming algorithms for several problems involving formal languages and finite-state automata. Although I have strived for a self-contained presentation, this material will likely make sense only to people who are already somewhat familiar with formal languages *and* dynamic programming.

C.1 DFA Minimization

Write this. ★★★

C.2 NFA Acceptance

Recall that a ***nondeterministic finite-state automaton***—or ***NFA*** for short—can be described as a directed graph, whose edges are called *states* and whose edges have *labels*

drawn from a finite set Σ called the *alphabet*. Every NFA has a designated *start* state and a subset of *accepting* states. Every walk in this graph has a label, which is a string formed by concatenating the labels of the edges in the walk. A string w is *accepted* by an NFA if and only if there is a walk from the start state to one of the accepting states whose label is w.

More formally (or at least, more symbolically), an NFA consists of a finite set Q of states, a start state $s \in Q$, a set of accepting states $A \subseteq Q$, and a transition function $\delta : Q \times \Sigma \to 2^Q$. We recursively extend the transition function to a function $\delta^* : Q \times \Sigma^* \to 2^Q$ over strings by defining

$$\delta^*(q, w) = \begin{cases} \{q\} & \text{if } w = \varepsilon, \\ \displaystyle\bigcup_{r \in \delta(q,a)} \delta^*(r, x) & \text{if } w = ax \text{ for some } a \in \Sigma \text{ and } x \in \Sigma^* \end{cases}$$

Our NFA ***accepts*** string w if and only if the set $\delta^*(s, w)$ contains at least one accepting state.

We can express this acceptance criterion more directly in terms of the original transition function δ as follows. Let $\mathit{Accepts?}(q, w) = \textsc{True}$ if our NFA would accept string w if it started in state q (instead of the usual start state s), and $\mathit{Accepts?}(q, w) = \textsc{False}$ otherwise. The function *Accepts?* has the following recursive definition:

$$\mathit{Accepts?}(q, w) := \begin{cases} \textsc{True} & \text{if } w = \varepsilon \text{ and } q \in A \\ \textsc{False} & \text{if } w = \varepsilon \text{ and } q \notin A \\ \displaystyle\bigvee_{r \in \delta(q,a)} \mathit{Accepts?}(r, x) & \text{if } w = ax \text{ for some } a \in \Sigma \text{ and } x \in \Sigma^* \end{cases}$$

Then our NFA accepts w if and only if $\mathit{Accepts?}(s, w) = \textsc{True}$.

Backtracking

In the magical world of non-determinism, we can imagine that the NFA always makes the right decision when faced with multiple transitions, or perhaps spawns off an independent parallel thread for each possible choice. Alas, real computers are neither clairvoyant nor (despite the increasing use of multiple cores) infinitely parallel. To simulate the NFA's behavior directly, we must recursively explore the consequences of each choice explicitly.

Before we can turn our recursive definition(s) into an algorithm, we need to nail down the precise input representation. Let's assume, without loss of generality, that the alphabet is $\Sigma = \{1, 2, \ldots, |\Sigma|\}$, the state set is $Q = \{1, 2, \ldots, |Q|\}$, the start state is state 1, and our input consists of three arrays:

- A boolean array $A[1 .. |Q|]$, where $A[q] = \textsc{True}$ if and only if $q \in A$.

- A boolean array $\delta[1..|Q|, 1..|\Sigma|, 1..|Q|]$, where $\delta[p, a, q] = \textsc{True}$ if and only if $p \in \delta(q, a)$.
- An array $w[1..n]$ of symbols, representing the input string.

Fixing the input string $w[1..n]$ lets us simplify the definition of the acceptance function slightly. For any state q and index i, define $\mathit{Accepts?}(q, i) = \textsc{True}$ if the NFA accepts the suffix $w[i..n]$ starting in state q, and $\mathit{Accepts?}(q, i) = \textsc{False}$ otherwise.

$$\mathit{Accepts?}(q, i) := \begin{cases} \textsc{True} & \text{if } i > n \text{ and } q \in A \\ \textsc{False} & \text{if } i > n \text{ and } q \notin A \\ \displaystyle\bigvee_{r \in \delta(q,a)} \mathit{Accepts?}(r, i+1) & \text{otherwise} \end{cases}$$

In particular, the NFA accepts w if and only if $\mathit{Accepts?}(s, 1) = \textsc{True}$. This is only a minor notational change from the previous definition, but it makes the recursive calls more efficient (passing just an integer instead of a string) and eventually easier to memoize. Our new recursive definition translates directly into the following backtracking algorithm.

```
ACCEPTS?(q, i):
  if i > n
      return A[q]
  for r ← 1 to |Q|
      if δ[q, w[i], r] and ACCEPTS?(r, i + 1)
          return TRUE
  return FALSE
```

Dynamic Programming

The previous algorithm runs in $O(|Q|^n)$ time in the worst case; fortunately, everything is set up to quickly derive a faster dynamic programming solution. The *Accepts?* function can be memoized into a two-dimensional array $\mathit{Accepts?}[1..|Q|, 1..n+1]$. Each entry $\mathit{Accepts?}[q, i]$ depends only on entries of the form $\mathit{Accepts?}[r, i+1]$, so we can fill the memoization table by considering indices i in decreasing order in the outer loop, and states q in arbitrary order in the inner loop. Evaluating each entry $\mathit{Accepts?}[q, i]$ requires $O(|Q|)$ time, using an even deeper loop over all states r, and there are $O(n|Q|)$ such entries. Thus, the entire dynamic programming algorithm requires $\boldsymbol{O(n|Q|^2)}$ ***time***.

```
NFAAccepts?(A[1..|Q|], δ[1..|Q|, 1..|Σ|, 1..|Q|], w[1..n]):
  for q ← 1 to |Q|
      Accepts?[q, n+1] ← A[q]
  for i ← n down to 1
      for q ← 1 to |Q|
          Accepts?[q, i] ← False
          for r ← 1 to |Q|
              if δ[q, w[i], r] and Accepts?[r, i+1]
                  Accepts?[q, i] ← True
  return Accepts?[1, 1]
```

C.3 Regular Expression Matching

★★★

Write this. Assume given an expression tree.

C.4 CFG Parsing

Another problem from formal languages that can be solved via dynamic programming is the ***parsing*** problem for context-free languages. Given a string w and a context-free grammar G, does w belong to the language generated by G? Recall that a ***context-free grammar*** over the alphabet Σ consists of a finite set Γ of *non-terminals* (disjoint from Σ) and a finite set of *production rules* of the form $A \to w$, where A is a nonterminal and w is a string over $\Sigma \cup \Gamma$. The ***length*** of a context free grammar is the number of production rules.

Real-world applications of parsing normally require more information than just a single bit. For example, compilers require parsers that output a *parse tree* of the input code; some natural language applications require the *number* of distinct parse trees for a given string; others assign probabilities to the production rules and then ask for the *most likely* parse tree for a given string. However, once we have an algorithm for the decision problem, it it not hard to extend it to answer these more general questions.

For any nonterminal A and any string x, define $Gen?(A, x) = \text{True}$ if x can be derived from A and $Gen?(A, x) = \text{False}$ otherwise. At first glance, it seems that the production rules of the CFL immediately give us a (rather complicated) recursive definition for this function. Unfortunately, there are a few subtle problems.[1]

- Consider the context-free grammar $S \to \varepsilon \mid SS \mid (S)$ that generates all properly balanced strings of parentheses. The most straightforward recursive algorithm for $Gen?(S, w)$ recursively checks whether $x \in L(S)$ and $y \in L(S)$, for *every* possible partition $w = x \bullet y$, *including the trivial partitions* $w = \varepsilon \bullet w$ *and* $w = w \bullet \varepsilon$. Thus, $Gen?(S, w)$ can call itself, leading to an infinite loop.

[1]Similar subtleties arise in induction proofs about context-free grammars.

- Consider another grammar that includes the productions $S \to A$, $A \to B$, and $B \to S$, possibly among others. The "obvious" recursive algorithm for $Gen?(S, w)$ must call $Gen?(A, w)$, which calls $Gen?(B, w)$, which calls $Gen?(S, w)$, and we are again in an infinite loop.

To avoid these issues, we will make the simplifying assumption that our input grammar is in ***Chomsky normal form***, which means that it has the following special structure:

- The starting non-terminal S does not appear on the right side of any production rule.
- The grammar *may* include the production rule $S \to \varepsilon$, where S is the starting non-terminal, but does not contain the rule $A \to \varepsilon$ for any other non-terminal $A \neq S$.
- Otherwise, every production rule has the form $A \to BC$ (two non-terminals) or $A \to a$ (one terminal).

Any context-free grammar can be converted into an equivalent grammar Chomsky normal form; moreover, if the original grammar has length L, an equivalent CFG grammar of length $O(L^2)$ can be computed in $O(L^2)$ time. The conversion algorithm is fairly complex, and we haven't yet seen all the algorithmic tools needed to understand it; for purposes of this chapter, it's enough to know that such an algorithm exists. For example, the language of all properly balanced strings of parentheses is generated by the CNF grammar

$$S \to \varepsilon \mid AA \mid BC \qquad A \to AA \mid BC \qquad B \to LA \qquad C \to RA \qquad L \to (\qquad R \to)$$

⟨⟨This is incorrect.⟩⟩ ⊲⊲⊲⊲⊲

With this simplifying assumption in place, the function *Gen?* has a relatively straightforward recursive definition.

$$Gen?(A, x) = \begin{cases} \textsc{True} & \text{if } |x| \leq 1 \text{ and } A \to x \\ \textsc{False} & \text{if } |x| \leq 1 \text{ and } A \not\to x \\ \displaystyle\bigvee_{A \to BC} \bigvee_{y \bullet z = x} Gen?(B, y) \wedge Gen?(C, z) & \text{otherwise} \end{cases}$$

The first two cases take care of terminal productions $A \to a$ and the ε-production $S \to \varepsilon$ (if the grammar contains it). Here the notation $A \not\to x$ means that $A \to x$ is *not* a production rule in the given grammar. In the third case, for all production rules $A \to BC$, and for all ways of splitting x into a *non-empty* prefix y and a *non-empty* suffix z, we recursively check whether $y \in L(B)$ and $z \in L(C)$. Because we pass strictly smaller strings in the second argument of these recursive calls, every branch of the recursion tree eventually terminates.

This recurrence was transformed into a dynamic programming algorithm by Tadao Kasami in 1965, and again independently by Daniel Younger in 1967, and again independently by John Cocke in 1970, so of course the resulting algorithm is known as "Cocke-Younger-Kasami", or more commonly ***the CYK algorithm***, with the names listed in *reverse* chronological order.

We can derive the CYK algorithm from the previous recurrence as follows. As usual for recurrences involving strings, we modify the function to accept index arguments instead of strings, to ease memoization. Fix the input string w, and then let $Gen?(A, i, j) = \textsc{True}$ if and only if the substring $w[i\,..\,j]$ can be derived from non-terminal A. Now our earlier recurrence can be rewritten as follows:

$$Gen?(A, i, j) = \begin{cases} \textsc{True} & \text{if } i = j \text{ and } A \to w[i] \\ \textsc{False} & \text{if } i = j \text{ and } A \not\to w[i] \\ \displaystyle\bigvee_{A \to BC} \bigvee_{k=i}^{j-1} Gen?(B, i, k) \wedge Gen?(C, k+1, j) & \text{otherwise} \end{cases}$$

Then w lies in the language of the grammar if and only if either $Gen?(A, 1, n) = \textsc{True}$, or $w = \varepsilon$ and the grammar includes the production $S \to \varepsilon$.

We can memoize the function *Gen?* into a three-dimensional boolean array $Gen[1\,..\,|\Gamma|, 1\,..\,n, 1\,..\,n]$, where the first dimension is indexed by the non-terminals Γ in the input grammar. Each entry $Gen[A, i, j]$ in this array depends on entries of the form $Gen[\,\cdot\,, i, k]$ for some $k < j$, or $Gen[\,\cdot\,, k+1, j]$ for some $k \geq i$. Thus, we can fill the array by increasing j and decreasing i in two outer loops, and considering non-terminals A in arbitrary order in the inner loop. The resulting dynamic programming algorithm runs in $\boldsymbol{O(n^3 L)}$ ***time***, where L is the length of the input grammar.

```
CYKParse(w, G):
  if w = ε
      if G contains the production S → ε
          return True
      else
          return False
  for i ← 1 to n
      for all non-terminals A
          if G contains the production A → w[i]
              Gen[A, i, i] ← True
          else
              Gen[A, i, i] ← False
  for j ← 1 to n
      for i ← n down to j + 1
          for all non-terminals A
              Gen[A, i, j] ← False
          for all production rules A → BC
              for k ← i to j − 1
                  if Gen[B, i, k] and Gen[C, k + 1, j]
                      Gen[A, i, j] ← True
  return Gen[S, 1, n]
```

Exercises

Add some exercises! ★★★

It is a very sad thing that nowadays there is so little useless information.

— Oscar Wilde, "A Few Maxims for the Instruction Of The Over-Educated" (1894)

Ninety percent of science fiction is crud. But then, ninety percent of everything is crud, and it's the ten percent that isn't crud that is important.

— [Theodore] Sturgeon's Law (1953)

Dis-moi ce que tu manges, je te dirai ce que tu es.

— Jean Anthelme Brillat-Savarin, *Physiologie du Gout* (1825)

D

Advanced Dynamic Programming

[Read Chapter 3 first.]

Status: Unfinished.

Replace Hirschberg with Chowdhury-Ramachandran

Four Russians?

Dynamic programming is a powerful technique for efficiently solving recursive problems, but it's hardly the end of the story. In many cases, once we have a basic dynamic programming algorithm in place, we can make further improvements to bring down the running time or the space usage. We saw one example in the Fibonacci number algorithm. Buried inside the naïve iterative Fibonacci algorithm is a recursive problem—computing a power of a matrix—that can be solved more efficiently by dynamic programming techniques—in this case, repeated squaring.

D.1 Saving Space: Divide and Conquer

Just as we did for the Fibonacci recurrence, we can reduce the space complexity of our edit distance algorithm from $O(mn)$ to $O(m+n)$ by only storing the current and previous rows of the memoization table. This "sliding window" technique provides an easy space improvement for most (but *not* all) dynamic programming algorithm.

Unfortunately, this technique seems to be useful only if we are interested in the *cost* of the optimal edit sequence, not if we want the optimal edit sequence itself. By throwing away most of the table, we apparently lose the ability to walk backward through the table to recover the optimal sequence.

Fortunately for memory-misers, in 1975 Dan Hirschberg discovered a simple divide-and-conquer strategy that allows us to compute the optimal edit sequence in $O(mn)$ time, using just $O(m+n)$ space. The trick is to record not just the edit distance for each pair of prefixes, but also a single position in the middle of the optimal editing sequence for that prefix. Specifically, any optimal editing sequence that transforms $A[1..m]$ into $B[1..n]$ can be split into two smaller editing sequences, one transforming $A[1..m/2]$ into $B[1..h]$ for some integer h, the other transforming $A[m/2+1..m]$ into $B[h+1..n]$.

To compute this breakpoint h, we define a second function $\mathit{Half}(i,j)$ such that some optimal edit sequence from $A[1..i]$ into $B[1..j]$ contains an optimal edit sequence from $A[1..m/2]$ to $B[1..\mathit{Half}(i,j)]$. We can define this function recursively as follows:

$$\mathit{Half}(i,j) = \begin{cases} \infty & \text{if } i < m/2 \\ j & \text{if } i = m/2 \\ \mathit{Half}(i-1,j) & \text{if } i > m/2 \text{ and } \mathit{Edit}(i,j) = \mathit{Edit}(i-1,j)+1 \\ \mathit{Half}(i,j-1) & \text{if } i > m/2 \text{ and } \mathit{Edit}(i,j) = \mathit{Edit}(i,j-1)+1 \\ \mathit{Half}(i-1,j-1) & \text{otherwise} \end{cases}$$

(Because there there may be more than one optimal edit sequence, this is not the only correct definition.) A simple inductive argument implies that $\mathit{Half}(m,n)$ is indeed the correct value of h. We can easily modify our earlier algorithm so that it computes $\mathit{Half}(m,n)$ at the same time as the edit distance $\mathit{Edit}(m,n)$, all in $O(mn)$ time, using only $O(m)$ space.

Edit		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	**0**	1	2	3	4	5	6	7	8
L	2	1	**0**	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	**4**	5	6
R	4	3	2	2	2	**2**	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	**3**	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	**4**	5	6
I	9	8	7	7	7	7	**6**	5	5	6
C	10	9	8	8	8	8	7	6	6	6

Half		A	L	G	O	R	I	T	H	M
	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
A	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
L	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
T	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
R	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
U	0	1	2	3	4	5	6	7	8	9
I	0	1	2	3	4	5	**5**	5	5	5
S	0	1	2	3	4	5	5	5	5	5
T	0	1	2	3	4	5	5	**5**	5	5
I	0	1	2	3	4	5	**5**	5	5	5
C	0	1	2	3	4	5	5	5	5	5

Finally, to compute the optimal editing sequence that transforms A into B, we recursively compute the optimal sequences transforming $A[1..m/2]$ into $B[1..\mathit{Half}(m,n)]$ and transforming $A[m/2+1..m]$ into $B[\mathit{Half}(m,n)+1..n]$. The recursion bottoms out

when one string has only constant length, in which case we can determine the optimal editing sequence in linear time using our old dynamic programming algorithm. The running time of the resulting algorithm satisfies the following recurrence:

$$T(m,n) = \begin{cases} O(n) & \text{if } m \leq 1 \\ O(m) & \text{if } n \leq 1 \\ O(mn) + T(m/2,h) + T(m/2,n-h) & \text{otherwise} \end{cases}$$

It's easy to prove inductively that $T(m,n) = O(mn)$, no matter what the value of h is. Specifically, the entire algorithm's running time is at most twice the time for the initial dynamic programming phase.

$$\begin{aligned} T(m,n) &\leq \alpha mn + T(m/2,h) + T(m/2,n-h) \\ &\leq \alpha mn + 2\alpha mh/2 + 2\alpha m(n-h)/2 && \text{[inductive hypothesis]} \\ &= 2\alpha mn \end{aligned}$$

A similar inductive argument implies that the algorithm uses only $O(n+m)$ space.

Compare with Chowdhury and Ramachandran [SODA 2006]: Compute values on boundary via divide-and-conquer (instead of by sweeping). Reconstruct sequence recursively as well. Subproblem: given a submatrix with values on top/left boundary, and target indices on bottom/right boundary, report LCS leading to target and top/left index.

Also, this isn't actually how Hirschberg described "Hirschberg's algorithm".

★★★

Hirschberg's divide-and-conquer trick can be applied to almost any dynamic programming problem to obtain an algorithm to construct an optimal *structure* (in this case, the cheapest edit sequence) within the same space and time bounds as computing the *cost* of that optimal structure (in this case, edit distance). For this reason, we will almost always ask you for algorithms to compute the cost of some optimal structure, not the optimal structure itself.

D.2 Saving Time: Sparseness

In many applications of dynamic programming, we are faced with instances where almost every recursive subproblem will be resolved exactly the same way. We call such instances *sparse*. For example, we might want to compute the edit distance between two strings that have few characters in common, which means there are few "free" substitutions anywhere in the table. Most of the table has exactly the same structure. If we can reconstruct the entire table from just a few key entries, then why compute the entire table?

To better illustrate how to exploit sparseness, let's consider a simplification of the edit distance problem, in which substitutions are not allowed (or equivalently, where a

substitution counts as two operations instead of one). Now our goal is to maximize the number of "free" substitutions, or equivalently, to find the *longest common subsequence* of the two input strings.

Fix the two input strings $A[1..n]$ and $B[1..m]$. For any indices i and j, let $LCS(i,j)$ denote the length of the longest common subsequence of the prefixes $A[1..i]$ and $B[1..j]$. This function can be defined recursively as follows:

$$LCS(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS(i-1, j-1) + 1 & \text{if } A[i] = B[j] \\ \max\{LCS(i, j-1),\ LCS(i-1, j)\} & \text{otherwise} \end{cases}$$

This recursive definition directly translates into an $O(mn)$-time dynamic programming algorithm.

LCS	«	A	L	G	O	R	I	T	H	M	S	»
«	**0**	0	0	0	0	0	0	0	0	0	0	0
A	0	**1**	1	1	1	1	1	1	1	1	1	1
L	0	1	**2**	2	2	2	2	2	2	2	2	2
T	0	1	2	2	2	2	2	**3**	3	3	3	3
R	0	1	2	2	2	**3**	3	3	3	3	3	3
U	0	1	2	2	3	3	3	3	3	3	3	3
I	0	1	2	2	3	3	**4**	4	4	4	4	4
S	0	1	2	2	3	3	4	4	4	4	**5**	5
T	0	1	2	2	3	3	4	**5**	5	5	5	5
I	0	1	2	2	3	3	**4**	5	5	5	5	5
C	0	1	2	2	3	3	4	5	5	5	5	5
»	0	1	2	2	3	3	4	5	5	5	5	**6**

Figure D.1. The *LCS* memoization table for the strings ALGORITHMS and ALTRUISTIC; the brackets « and » are sentinel characters. Match points are indicated in red.

Call an index pair (i,j) a ***match point*** if $A[i] = B[j]$. In some sense, match points are the only "interesting" locations in the memoization table. Given a list of the match points, we can reconstruct the entire table using the following recurrence:

$$LCS(i,j) = \begin{cases} 0 & \text{if } i = j = 0 \\ \max\left\{LCS(i',j') \;\middle|\; A[i'] = B[j'] \text{ and } i' < i \text{ and } j' < j\right\} + 1 & \text{if } A[i] = B[j] \\ \max\left\{LCS(i',j') \;\middle|\; A[i'] = B[j'] \text{ and } i' \le i \text{ and } j' \le j\right\} & \text{otherwise} \end{cases}$$

(Notice that the inequalities are strict in the second case, but not in the third.) To simplify boundary issues, we add unique sentinel characters $A[0] = B[0]$ and $A[m+1] = B[n+1]$ to both strings (brackets « and » in the example above). These sentinels ensure that the sets on the right side of the recurrence equation are non-empty, and that we *only* have to consider match points to compute $LCS(m,n) = LCS(m+1, n+1) - 1$.

If there are K match points, we can actually compute them in $O(m \log m + n \log n + K)$ time. Sort the characters in each input string, remembering the original index of each character, and then essentially merge the two sorted arrays, as follows:

```
FindMatches(A[1..m], B[1..n]):
  for i ← 1 to m:  I[i] ← i
  for j ← 1 to n:  J[j] ← j

  sort A and permute I to match
  sort B and permute J to match

  i ← 1;  j ← 1
  while i < m and j < n
      if A[i] < B[j]
          i ← i + 1
      else if A[i] > B[j]
          j ← j + 1
      else                          ⟨⟨Found a match!⟩⟩
          ii ← i
          while A[ii] = A[i]
              jj ← j
              while B[jj] = B[j]
                  report (I[ii], J[jj])
                  jj ← jj + 1
              ii ← i + 1
          i ← ii;  j ← jj
```

To efficiently evaluate our modified recurrence, we once again turn to dynamic programming. We consider the match points in lexicographic order—the order they would be encountered in a standard row-major traversal of the $m \times n$ table—so that when we need to evaluate $LCS(i, j)$, all match points (i', j') with $i' < i$ and $j' < j$ have already been evaluated.

```
SparseLCS(A[1..m], B[1..n]):
  Match[1..K] ← FindMatches(A, B)
  Match[K + 1] ← (m + 1, n + 1)        ⟨⟨Add end sentinel⟩⟩
  Sort M lexicographically

  for k ← 1 to K
      (i, j) ← Match[k]
      LCS[k] ← 1                       ⟨⟨From start sentinel⟩⟩
      for ℓ ← 1 to k − 1
          (i′, j′) ← Match[ℓ]
          if i′ < i and j′ < j
              LCS[k] ← min{LCS[k], 1 + LCS[ℓ]}
  return LCS[K + 1] − 1
```

The overall running time of this algorithm is $O(m \log m + n \log n + K^2)$. So as long as $K = o(\sqrt{mn})$, this algorithm is actually faster than naïve dynamic programming. With

some additional work, the running time can be further improved to $O(m \log m + n \log n + K \log K)$.

D.3 Saving Time: Monotonicity

Recall the optimal binary search tree problem from the previous lecture. Given an array $F[1..n]$ of access frequencies for n items, the problem it to compute the binary search tree that minimizes the cost of all accesses. A relatively straightforward dynamic programming algorithm solves this problem in $O(n^3)$ time.

As for longest common subsequence problem, the algorithm can be improved by exploiting some structure in the memoization table. In this case, however, the relevant structure isn't in the table of costs, but rather in the table used to reconstruct the actual optimal tree. Let $OptRoot[i, j]$ denote the index of the root of the optimal search tree for the frequencies $F[i..j]$; this is always an integer between i and j. Donald Knuth proved the following nice monotonicity property for optimal subtrees: If we move either end of the subarray, the optimal root moves in the same direction or not at all. More formally:

$$\boxed{OptRoot[i, j-1] \leq OptRoot[i, j] \leq OptRoot[i+1, j] \text{ for all } i \text{ and } j.}$$

In other words, every row and column in the array $OptRoot[1..n, 1..n]$ is sorted. This (nontrivial!) observation suggests the following more efficient algorithm:

```
FasterOptimalSearchTree(f[1..n]):
  InitF(f[1..n])
  for i ← 1 downto n
      OptCost[i, i−1] ← 0
      OptRoot[i, i−1] ← i
  for d ← 0 to n
      for i ← 1 to n
          ComputeCostAndRoot(i, i+d)
  return OptCost[1, n]
```

```
ComputeCostAndRoot(i, j):
  OptCost[i, j] ← ∞
  for r ← OptRoot[i, j−1] to OptRoot[i+1, j]
      tmp ← OptCost[i, r−1] + OptCost[r+1, j]
      if OptCost[i, j] > tmp
          OptCost[i, j] ← tmp
          OptRoot[i, j] ← r
  OptCost[i, j] ← OptCost[i, j] + F[i, j]
```

It's not hard to see that the loop index r increases monotonically from 1 to n during each iteration of the *outermost* for loop of FasterOptimalSearchTree. Consequently, the total cost of all calls to ComputeCostAndRoot is only $O(n^2)$.

If we formulate the problem slightly differently, this algorithm can be improved even further. Suppose we require the optimum *external* binary tree, where the keys $A[1..n]$ are all stored at the leaves, and intermediate pivot values are stored at the internal nodes. An algorithm discovered by Ching Hu and Alan Tucker[1] computes the optimal binary search tree in this setting in only $O(n \log n)$ time!

D.4 Saving Time: More Monotonicity

Knuth's algorithm can be significantly generalized by considering a more subtle form of monotonicity in the *cost* array. A common (but often implicit) operation in many dynamic programming algorithms is finding the minimum element in every row of a two-dimensional array. For example, consider a single iteration of the outer loop in FasterOptimalSearchTree, for some fixed value of d. Define an array M by setting

$$M[i,r] = \begin{cases} OptCost[i,r-1] + OptCost[r+1,i+d] & \text{if } i \le r \le i+d \\ \infty & \text{otherwise} \end{cases}$$

Each call to ComputeCostAndRoot$(i, i+d)$ computes the smallest element in the ith row of this array M.

Let $M[1..m, 1..n]$ be an arbitrary two-dimensional array. We say that M is ***monotone*** if the leftmost smallest element in any row is either directly above or to the left of the leftmost smallest element in any later row. To state this condition more formally, let $LM(i)$ denote the index of the smallest item in the ith row $M[i, \cdot]$; if there is more than one smallest item, choose the one furthest to the left. Then M is monotone if and only if $LM(i) \le LM(i+1)$ for every index i. For example, the following 5×5 array is monotone (as shown by the hilighted row minima).

$$\begin{bmatrix} 12 & 21 & 38 & 76 & 27 \\ 74 & 14 & 14 & 29 & 60 \\ 21 & 8 & 25 & 10 & 71 \\ 68 & 45 & 29 & 15 & 76 \\ 97 & 8 & 12 & 2 & 6 \end{bmatrix}$$

Given a monotone $m \times n$ array M, we can compute the array $LM[i]$ containing the index of the leftmost minimum element in every row as follows. We begin by recursively

[1] T. C. Hu and A. C. Tucker, Optimal computer search trees and variable length alphabetic codes, *SIAM J. Applied Math.* 21:514–532, 1971. For a slightly simpler algorithm with the same running time, see A. M. Garsia and M. L. Wachs, A new algorithms for minimal binary search trees, *SIAM J. Comput.* 6:622–642, 1977. The original correctness proofs for both algorithms are rather intricate; for simpler proofs, see Marek Karpinski, Lawrence L. Larmore, and Wojciech Rytter, Correctness of constructing optimal alphabetic trees revisited, *Theoretical Computer Science*, 180:309-324, 1997.

computing the leftmost minimum elements in all odd-indexed rows of M. Then for each even index $2i$, the monotonicity of M implies the bounds

$$LM[2i-1] \le LM[2i] \le LM[2i+1],$$

so we can compute $LM[2i]$ by brute force by searching only within that range of indices. This search requires exactly $LM[2i+1]-LM[2i-1]$ comparisons, because finding the minimum of k numbers requires $k-1$ comparisons. In particular, if $LM[2i-1] = LM[2i+1]$, then we don't need to perform any comparisons on row $2i$. Summing over all even indices, we find that the total number of comparisons is

$$\sum_{i=1}^{m/2} LM[2i+1]-LM[2i-1] \;=\; LM[m+1]-LM[1] \;\le\; n.$$

We also need to spend constant time on each row, as overhead for the main loop, so the total time for our algorithm is $O(n+m)$ plus the time for the recursive call. Thus, the running time satisfies the recurrence

$$T(m,n) = T(m/2,n) + O(n+m),$$

which implies that our algorithm runs in $\boldsymbol{O(m+n\log m)}$ ***time***.

Alternatively, we could use the following divide-and-conquer procedure, similar in spirit to Hirschberg's divide-and-conquer algorithm. Compute the middle leftmost-minimum index $h = LM[m/2]$ by brute force in $O(n)$ time, and then recursively find the leftmost minimum entries in the following submatrices:

$$M\,[1\,..\,m/2-1,\ 1\,..\,h] \qquad M\,[m/2+1\,..\,m,\ h\,..\,n]$$

The worst-case running time $T(m,n)$ for this algorithm obeys the following recurrence (after removing some irrelevant ± 1s from the recursive arguments, as usual):

$$T(m,n) = \begin{cases} 0 & \text{if } m < 1 \\ O(n) + \max\limits_{k}\big(T(m/2,k) + T(m/2,n-k)\big) & \text{otherwise} \end{cases}$$

The recursion tree for this recurrence is a balanced binary tree of depth $\log_2 m$, and therefore with $O(m)$ nodes. The total number of *comparisons* performed at each level of the recursion tree is $O(n)$, but we also need to spend at least constant time at each node in the recursion tree. Thus, this divide-and-conquer formulation also runs in $\boldsymbol{O(m+n\log m)}$ ***time***.

In fact, these two algorithms are morally identical. Both algorithms examine the same subset of array entries and perform the same pairwise comparisons, although in different orders. Specifically, the divide-and-conquer algorithm performs the usual depth-first traversal of its recursion tree. The even-odd algorithm actually performs a *breadth*-first traversal of the exactly the same recursion tree, handling each level of the recursion tree in a single for-loop. The breadth-first formulation of the algorithm will prove more useful in the long run.

D.5 Saving More Time: Total Monotonicity

A more general technique for exploiting structure in dynamic programming arrays was discovered by Alok Aggarwal, Maria Klawe, Shlomo Moran, Peter Shor, and Robert Wilber in 1987; their algorithm is now universally known as ***SMAWK*** (pronounced "smoke") after their suitably-permuted initials.

SMAWK requires a stricter form of monotonicity in the input array. We say that M is ***totally monotone*** if the subarray defined by any subset of (not necessarily consecutive) rows and columns is monotone. For example, the following 5×5 array is monotone (as shown by the highlighted row minima), but not totally monotone (as shown by the gray 2×2 subarray).

$$\begin{bmatrix} 12 & 21 & 38 & 76 & 27 \\ 74 & 14 & 14 & 29 & 60 \\ 21 & 8 & 25 & 10 & 71 \\ 68 & 45 & 29 & 15 & 76 \\ 97 & 8 & 12 & 2 & 6 \end{bmatrix}$$

On the other hand, the following array is totally monotone:

$$\begin{bmatrix} 12 & 21 & 38 & 76 & 89 \\ 47 & 14 & 14 & 29 & 60 \\ 21 & 8 & 20 & 10 & 71 \\ 68 & 16 & 29 & 15 & 76 \\ 97 & 8 & 12 & 2 & 6 \end{bmatrix}$$

Given a totally monotone $m \times n$ array as input, SMAWK finds the leftmost smallest element of every row in only $\boldsymbol{O(n+m)}$ ***time***.

The Monge property

Before we go into the details of the SMAWK algorithm, it's useful to consider the most common special case of totally monotone matrices. A ***Monge array***[2] is a two-dimensional array M where

$$M[i,j] + M[i',j'] \leq M[i,j'] + M[i',j]$$

for all row indices $i < i'$ and all column indices $j < j'$. This inequality is sometimes called the *Monge property* or the *quadrangle inequality* or *submodularity*.

Lemma: *Every Monge array is totally monotone.*

[2]Monge arrays are named after Gaspard Monge, a French geometer who was one of the first to consider problems related to flows and cuts, in his 1781 *Mémoire sur la Théorie des Déblais et des Remblais*, which (in context) should be translated as "Treatise on the Theory of Holes and Dirt".

Proof: Let M be a two-dimensional array that is *not* totally monotone. Then there must be row indices $i < i'$ and column indices $j < j'$ such that $M[i,j] > M[i,j']$ and $M[i',j] \leq M[i',j']$. These two inequalities imply that $M[i,j]+M[i',j'] > M[i,j']+M[i',j]$, from which it follows immediately that M is *not* Monge. □

Monge arrays have several useful properties that make identifying them easier. For example, an easy inductive argument implies that we do not need to check every quadruple of indices; in fact, we can determine whether an $m \times n$ array is Monge in just $O(mn)$ time.

Lemma: *An array M is Monge if and only if $M[i,j]+M[i+1,j+1] \leq M[i,j+1]+M[i+1,j]$ for all indices i and j.*

Monge arrays arise naturally in geometric settings; the following canonical example of a Monge array was suggested by Monge himself in 1781. Fix two parallel lines ℓ and ℓ' in the plane. Let $p_1, p_2, \ldots, p_m$ be points on ℓ, and let $q_1, q_2, \ldots, q_n$ be points on ℓ', with each set indexed in order along their respective lines. Let $M[1..m, 1..n]$ be the array of Euclidean distances between these points:

$$M[i,j] := |p_i q_j| = \sqrt{(p_i.x - q_i.x)^2 + (p_i.y - q_i.y)^2}$$

An easy argument with the triangle inequality implies that this array is Monge. Fix arbitrary indices $i < i'$ and $j < j'$, and let x denote the intersection point of segments $p_i q_{j'}$ and $p_{i'} q_j$.

$$\begin{aligned}
M[i,j]+M[i',j'] &= |p_i q_j| + |p_{i'} q_{j'}| \\
&\leq |p_i x| + |x q_j| + |p_{i'} x| + |x q_{j'}| && \text{[triangle inequality]} \\
&= \big(|p_i x| + |x q_{j'}|\big) + \big(|p_{i'} x| + |x q_j|\big) \\
&= |p_i q_{j'}| + |p_{i'} q_j| \\
&= M[i,j'] + M[i',j]
\end{aligned}$$

Figure D.2. The cheapest way to move two specks of dirt (on the left) into two tiny holes (on the right). From Monge's 1781 treatise on the theory of holes and dirt.

There are also several easy ways to construct and combine Monge arrays, either from scratch or by manipulating other Monge arrays.

Lemma: *The following arrays are Monge:*

(a) *Any array with constant rows.*
(b) *Any array with constant columns.*
(c) *Any array that is all* 0*s except for an upper-right rectangular block of* 1*s.*
(d) *Any array that is all* 0*s except for an lower-left rectangular block of* 1*s.*
(e) *Any positive multiple of any Monge array.*
(f) *The sum of any two Monge arrays.*
(g) *The transpose of any Monge array.*

Each of these properties follows from straightforward definition chasing. For example, to prove part (f), let X and Y be arbitrary Monge arrays, and let $Z = X + Y$. For all row indices $i < i'$ and column indices $j < j'$, we immediately have

$$\begin{aligned} Z[i,j] + Z[i',j'] &= X[i,j] + X[i',j'] + Y[i,j] + Y[i',j'] \\ &\le X[i',j] + X[i,j'] + Y[i',j] + Y[i,j'] \\ &= Z[i',j] + Z[i,j']. \end{aligned}$$

The other properties have similarly elementary proofs.

In fact, the properties listed in the previous lemma precisely characterize Monge arrays: ***Every*** Monge array (including the geometric example above) is a positive linear combination of row-constant, column-constant, and upper-right block arrays. (This characterization was proved independently by Rudolf and Woeginger in 1995, Bein and Pathak in 1990, Burdyok and Trofimov in 1976, and possibly others.)

D.6 The SMAWK algorithm

The SMAWK algorithm alternates between two different subroutines, one for "tall" arrays with more rows than columns, the other for "wide" arrays with more columns than rows. The "tall" subroutine is just the divide-and-conquer algorithm described in the previous section—recursively compute the leftmost row minima in every other row, and then fill in the remaining row minima in $O(n+m)$ time. The secret to SMAWK's success is its handling of "wide" arrays.

For any row index i and any two column indices $p < q$, the definition of total monotonizity implies the following observations:

- If $M[i,p] \le M[i,q]$, then for all $h \le i$, we have $M[h,p] \le M[h,q]$ and thus $LM[h] \ne q$.
- If $M[i,p] > M[i,q]$, then for all $j \ge i$, we have $M[j,p] > M[j,q]$ and thus $LM[j] \ne p$.

Call an array entry $M[i,j]$ ***dead*** if we have enough information to conclude that $LM[i] \ne j$. Then after we compare any two entries in the same row, either the left entry

and everything below it is dead, or the right entry and everything above it is dead.

The following algorithm maintains a stack of column indices in an array $S[1..m]$ (yes, really m, the number of *rows*) and an index t indicating the number of indices on the stack.

```
REDUCE(M[1..m, 1..n]):
  t ← 1
  S[t] ← 1
  for k ← 1 to n
      while t > 0 and M[t, S[t]] ≥ M[t, k]
          t ← t − 1        ⟨⟨pop⟩⟩
      if t < m
          t ← t + 1
          S[t] ← k         ⟨⟨push k⟩⟩
  return S[1..t]
```

This algorithm maintains three important invariants:

- $S[1..t]$ is sorted in increasing order.
- For all $1 \le j \le t$, the top $j-1$ entires of column $S[j]$ are dead.
- If $j < k$ and j is not on the stack, then *every* entry in column j is dead.

The first invariant follows immediately from the fact that indices are pushed onto the stack in increasing order. The second and third are more subtle, but both follow inductively from the total monotonicity of the array. The following figure shows a typical state of the algorithm when it is about to compare the entries $M[t, S[t]]$ and $M[t, k]$ (indicated by stars). Dead entries are indicated in gray.

If $M[t, S[t]] < M[t, k]$, then $M[t, k]$ and everything above it is dead, so we can safely push column k onto the stack (unless we already have $t = n$, in which case every entry in column k is dead) and then increment k.

On the other hand, if $M[t, S[t]] > M[t, k]$, then we know that $M[t, S[t]]$ and everything below it is dead. But by the inductive hypothesis, every entry above $M[t, S[t]]$ is already dead. Thus, the entire column $S[t]$ is dead, so we can safely pop it off the stack.

In both cases, all invariants are maintained.

Immediately after every comparison $M[t, S[t]] \geq M[t, k]$? in the Reduce algorithm, we either increment the column index k or declare a column to be dead; each of these events can happen at most once per column. It follows that Reduce performs at most $2n$ comparisons and thus runs in $\boldsymbol{O(n)}$ ***time*** overall.

Moreover, when Reduce ends, every column whose index is not on the stack is completely dead. Thus, to compute the leftmost minimum element in every row of M, it suffices to examine only the $t < m$ columns with indices in the output array $S[1 .. t]$.

Finally, the main SMAWK algorithm first Reduces the input array (if it has fewer rows than columns), then recursively computes the leftmost row minima in every other, and finally fills in the rest of the row minima in $O(m + n)$ additional time. The argument

of the recursive call is an array with exactly $m/2$ rows and at most $\min\{n, m\}$ columns, so the running time obeys the following recurrence:

$$T(m,n) \le \begin{cases} O(m) + T(m/2, n) & \text{if } m \ge n, \\ O(n) + T(m/2, m) & \text{if } m < n. \end{cases}$$

The algorithm needs $O(n)$ time to REDUCE the input array to at most m rows, after which every recursive call reduces the (worst-case) number of rows and columns by a factor of two. We conclude that SMAWK runs in $\boldsymbol{O(m+n)}$ ***time***.

Later variants of SMAWK can compute row minima in totally monotone arrays in $O(m+n)$ time even when the rows are revealed one by one, and we require the smallest element in each row before we are given the next. These later variants can be used to speed up many dynamic programming algorithms by a factor of n when the final memoization table is totally monotone.

D.7 Using SMAWK

Finally, let's consider an (admittedly artificial) example of a dynamic programming algorithm that can be improved by SMAWK. Recall the Shimbel-Bellman-Ford algorithm for computing shortest paths:

```
BELLMANFORD(V, E, w):
  dist(s) ← 0
  for every vertex v ≠ s
      dist(v) ← ∞
  repeat V − 1 times
      for every edge u→v
          if dist(v) > dist(u) + w(u→v)
              dist(v) ← dist(u) + w(u→v)
```

We can rewrite this algorithm slightly; instead of relaxing *every* edge, we first identify the *tensest* edge leading into each vertex, and then relax only those edges. (This modification is actually closer to the natural dynamic-programming algorithm to compute shortest paths.)

```
MODBELLMANFORD(V, E, w):
  dist(s) ← 0
  for every vertex v ≠ s
      dist(v) ← ∞
  repeat V − 1 times
      for every vertex v
          mind(v) ← min_u (dist(u) + w(u→v))
      for every vertex v
          dist(v) ← min{dist(v), mind(v)}
```

The two lines in red can be interpreted as finding the minimum element in every row of a two-dimensional array M indexed by vertices, where

$$M[v,u] := dist(u) + w(u \to v)$$

Now consider the following input graph. Fix n arbitrary points $p_1, p_2, \ldots, p_n$ on some line ℓ and n more arbitrary points $q_1, q_2, \ldots, q_n$ on another line parallel to ℓ, with each set indexed in order along their respective lines. Let G be the complete bipartite graph whose vertices are the points p_i or q_j, whose edges are the segments $p_i q_j$, and where the weight of each edge is its natural Euclidean length. The standard version of Bellman-Ford requires $O(VE) = O(n^3)$ time to compute shortest paths in this graph.

The $2n \times 2n$ array M is not itself Monge, but we can easily decompose it into $n \times n$ Monge arrays as follows. Index the rows and columns of M by the vertices $p_1, p_2, \ldots, p_n, q_1, q_2, \ldots, q_n$ in that order. Then M decomposes naturally into four $n \times n$ blocks

$$M = \begin{bmatrix} \infty & U \\ V & \infty \end{bmatrix},$$

where every entry in the upper left and lower right blocks is ∞, and the other two blocks satisfy

$$U[i,j] = dist(q_j) + |p_i q_j| \qquad \text{and} \qquad V[i,j] = dist(p_j) + |p_j q_i|$$

for all indices i and j. Let P, Q, and D be the $n \times n$ arrays where

$$P[i,j] = dist(p_j) \qquad Q[i,j] = dist(q_j) \qquad D[i,j] = |p_i q_j|$$

for all indices i and j. Arrays P and Q are Monge, because all entries in the same column are equal, and the matrix D is Monge by the triangle inequality. It follows that the blocks $U = Q + D$ and $V = P + D^T$ are both Monge. Thus, by calling SMAWK twice, once on U and once on V, we can find all the row minima in M in $O(n)$ time. The resulting modification of Bellman-Ford runs in $\boldsymbol{O(n^2)}$ ***time***.[3]

An important feature of this algorithm is that ***it never explicitly constructs the array M***. Instead, whenever the SMAWK algorithm needs a particular entry $M[u,v]$, we compute it on the fly in $O(1)$ time.

D.8 Saving Time: Four Russians

Some day. Maybe.

★★★

[3]Yes, we could also achieve this running time using Disjktra's algorithm with Fibonacci heaps.

Exercises

1. Describe an algorithm to compute the edit distance between two strings $A[1..m]$ and $B[1...n]$ in $O(m \log m + n \log n + K^2)$ time, where K is the number of match points. *[Hint: Use our earlier FINDMATCHES algorithm as a subroutine.]*

2. (a) Describe an algorithm to compute the *longest increasing subsequence* of a string $X[1..n]$ in $O(n \log n)$ time.

   (b) Using your solution to part (a) as a subroutine, describe an algorithm to compute the longest common subsequence of two strings $A[1..m]$ and $B[1...n]$ in $O(m \log m + n \log n + K \log K)$ time, where K is the number of match points.

3. Describe an algorithm to compute the edit distance between two strings $A[1..m]$ and $B[1...n]$ in $O(m \log m + n \log n + K \log K)$ time, where K is the number of match points. *[Hint: Combine your answers for problems 1 and 2(b).]*

4. Let T be an arbitrary rooted tree, where each vertex is labeled with a positive integer. A subset S of the nodes of T is *heap-ordered* if it satisfies two properties:

   - S contains a node that is an ancestor of every other node in S.
   - For any node v in S, the label of v is larger than the labels of any ancestor of v in S.

A heap-ordered subset of nodes in a tree.

   (a) Describe an algorithm to find the largest heap-ordered subset S of nodes in T that has the heap property in $O(n^2)$ time.

   (b) Modify your algorithm from part (a) so that it runs in $O(n \log n)$ time when T is a linked list. *[Hint: This special case is equivalent to a problem you've seen before.]*

   ♥(c) Describe an algorithm to find the largest subset S of nodes in T that has the heap property, in $O(n \log n)$ time. *[Hint: Find an algorithm to merge two sorted lists of lengths k and ℓ in $O(\log \binom{k+\ell}{k})$ time.]*

5. Suppose you are given a sorted array $X[1..n]$ of distinct numbers and a positive integer k. A set of k intervals ***covers*** X if every element of X lies inside one of the k intervals. Your aim is to find k intervals $[a_1, z_1], [a_2, z_2], \ldots, [a_k, z_k]$ that cover X where the function $\sum_{i=1}^{k}(z_i - a_i)^2$ is as small as possible. Intuitively, you are trying to cover the points with k intervals whose lengths are as close to equal as possible.

   (a) Describe an algorithm that finds k intervals with minimum total squared length that cover X. The running time of your algorithm should be a simple function of n and k.

   (b) Consider the two-dimensional matrix $M[1..n, 1..n]$ defined as follows:

$$M[i,j] = \begin{cases} (X[j] - X[i])^2 & \text{if } i \le j \\ \infty & \text{otherwise} \end{cases}$$

   Prove that M satisfies the ***Monge property***: $M[i,j] + M[i',j'] \le M[i,j'] + M[i',j]$ for all indices $i < i'$ and $j < j'$.

   (c) Describe an algorithm that finds k intervals with minimum total squared length that cover X ***in $O(nk)$ time***. *[Hint: Solve part (a) first, then use part (b).]*

6. Suppose we want to summarize a large set S of values—for example, grade-point averages for every student who has ever attended UIUC—using a variable-width histogram. To construct a histogram, we choose a sorted sequence of ***breakpoints*** $b_0 < b_1 < \cdots < b_k$, such that every element of S lies between b_0 and b_k. Then for each interval $[b_{i-1}, b_i]$, the histogram includes a rectangle whose height is the number of elements of S that lie inside that interval.

A variable-width histogram with seven bins.

Unlike a standard histogram, which requires the intervals to have equal width, we are free to choose the breakpoints arbitrarily. For statistical purposes, it is useful for the *areas* of the rectangles to be as close to equal as possible. To that end, define the ***cost*** of a histogram to be the sum of the *squares* of the rectangle areas; we seek a histogram with minimum cost.

More formally, suppose we fix a sequence of breakpoints $b_0 < b_1 < \cdots < b_k$. For each index i, let n_i denote the number of input values in the ith interval:

$$n_i := \#\left\{x \in S \;\middle|\; b_{i-1} \le x < b_i\right\}.$$

Then the ***cost*** of the resulting histogram is $\sum_{i=1}^{k} (n_i(b_i - b_{i-1}))^2$. We want to compute a histogram with minimum cost for the given set S, where every breakpoint b_i is equal to some value in S.[4] In particular, b_0 must be the minimum value in S, and b_k must be the maximum value in S. Informally, you are trying to compute a histogram with k rectangles whose areas are as close to equal as possible.

Describe and analyze an algorithm to compute a variable-width histogram with minimum cost for a given set of data values. Your input is a sorted array $S[1..n]$ of distinct real numbers and an integer k. Your algorithm should return a sorted array $B[0..k]$ of breakpoints that minimizes the cost of the resulting histogram, where every breakpoint $B[i]$ is equal to some input value $S[j]$, and in particular $B[0] = S[1]$ and $B[k] = S[n]$.

[4]Thanks to the non-linear cost function, removing this assumption makes the problem *considerably* more difficult!

© 2018 Jeff Erickson http://algorithms.wtf

The problem is that we attempt to solve the simplest questions cleverly,
thereby rendering them unusually complex.
One should seek the simple solution.

— Anton Pavlovich Chekhov (c. 1890)

I love deadlines. I like the whooshing sound they make as they fly by.

— Douglas Adams, *The Salmon of Doubt* (2002)

E

Matroids

[Read Chapter 4 and 7 first.]

Status: Needs significant revision or deletion

Merge into greedy notes as hardsection?

E.1 Definitions

Many problems that can be correctly solved by greedy algorithms can be described in terms of an abstract combinatorial object called a *matroid*. Matroids were first described in 1935 by the mathematician Hassler Whitney as a combinatorial generalization of linear independence of vectors—"matroid" means "something sort of like a matrix".

A matroid $(S, \mathcal{I})$ consists of a finite ***ground set*** S and a collection $\mathcal{I}$ of subsets of X that satisfies three axioms:

- **Non-emptiness:** The empty set is in $\mathcal{I}$. (Thus, $\mathcal{I}$ is not itself empty.)
- **Heredity:** If $\mathcal{I}$ contains a subset $X \subseteq S$, then $\mathcal{I}$ contains every subset of X.
- **Exchange:** If X and Y are two sets in $\mathcal{I}$ where $|X| > |Y|$, then $Y \cup \{x\} \in \mathcal{I}$ for some element $x \in X \setminus Y$.

© 2018 Jeff Erickson

The subsets in $\mathcal{I}$ are typically called ***independent sets***; for example, we would say that any subset of an independent set is independent. An independent set is called a ***basis*** of the matroid if it is not a proper subset of another independent set. The exchange property implies that every basis of a matroid has the same cardinality. The ***rank*** of a subset X of the ground set is the size of the largest independent subset of X. A subset of S that is not in $\mathcal{I}$ is called ***dependent*** (surprise, surprise). Finally, a dependent set is called a ***circuit*** if every proper subset is independent.

Most of this terminology is justified by Whitney's original example:

- ***Linear matroid:*** Let A be any $n \times m$ matrix. A subset $I \subseteq \{1, 2, \ldots, n\}$ is independent if and only if the corresponding subset of columns of A is linearly independent.

The heredity property follows directly from the definition of linear independence; the exchange property is implied by an easy dimensionality argument. A basis in any linear matroid is also a basis (in the linear-algebra sense) of the vector space spanned by the columns of A. Similarly, the rank of a set of indices is precisely the rank (in the linear-algebra sense) of the corresponding set of column vectors.

Here are several other examples of matroids; some of these we will see again later. I will leave the proofs that these are actually matroids as exercises for the reader.

- ***Uniform matroid $U_{k,n}$:*** A subset $X \subseteq \{1, 2, \ldots, n\}$ is independent if and only if $|X| \leq k$. Any subset of $\{1, 2, \ldots, n\}$ of size k is a basis; any subset of size $k+1$ is a circuit.
- ***Graphic/cycle matroid $\mathcal{M}(G)$:*** Let $G = (V, E)$ be an arbitrary undirected graph. A subset of E is independent if it defines an *acyclic* subgraph of G. A basis in the graphic matroid is a spanning tree of G; a circuit in this matroid is a cycle in G.
- ***Cographic/cocycle matroid $\mathcal{M}^*(G)$:*** Let $G = (V, E)$ be an arbitrary undirected graph. A subset $I \subseteq E$ is independent if the complementary subgraph $(V, E \setminus I)$ of G is *connected*. A basis in this matroid is the complement of a spanning tree; a circuit in this matroid is a *cocycle*—a minimal set of edges that disconnects the graph.
- ***Matching matroid:*** Let $G = (V, E)$ be an arbitrary undirected graph. A subset $I \subseteq V$ is independent if there is a matching in G that covers I.
- ***Disjoint path matroid:*** Let $G = (V, E)$ be an arbitrary *directed* graph, and let s be a fixed vertex of G. A subset $I \subseteq V$ is independent if and only if there are edge-disjoint paths from s to each vertex in I.

Now suppose each element of the ground set of a matroid $\mathcal{M}$ is given an arbitrary non-negative weight. The ***matroid optimization problem*** is to compute a basis with maximum total weight. For example, if $\mathcal{M}$ is the cycle matroid for a graph G, the matroid optimization problem asks us to find the maximum spanning tree of G. Similarly, if $\mathcal{M}$ is the cocycle matroid for G, the matroid optimization problem seeks (the complement of) the *minimum* spanning tree.

The following natural greedy strategy computes a basis for any weighted matroid $(S, \mathcal{I})$, where the ground set S is represented by an array $S[1..n]$, and the weights of ground elements are represented by another array $w[1..n]$.

```
GreedyBasis(S[1..n], I, w[1..n]):
  sort S in decreasing order of weight w
  G ← ∅
  for i ← 1 to n
      if G ∪ {S[i]} ∈ I
          add S[i] to G
  return G
```

Suppose we can test in $F(n)$ time whether a given subset $X \subset S$ is independent. Then this algorithm runs in $O(n \log n + n \cdot F(n))$ time.

Theorem E.1. *For any matroid $\mathcal{M} = (S, \mathcal{I})$ and any weight function w, GreedyBasis$(S, \mathcal{I}, w)$ returns a maximum-weight basis of $\mathcal{M}$.*

Proof: We use a standard exchange argument. Let $G = \{g_1, g_2, \ldots, g_k\}$ be the independent set returned by GreedyBasis$(S, \mathcal{I}, w)$. If any other element of S could be added to G to obtain a larger independent set, the greedy algorithm would have added it. Thus, G is a basis.

For purposes of deriving a contradiction, suppose there is an independent set $H = \{h_1, h_2, \ldots, h_\ell\}$ such that

$$\sum_{i=1}^{k} w(g_i) < \sum_{j=1}^{\ell} w(h_i).$$

Without loss of generality, assume that H is a basis. The exchange property now implies that $k = \ell$.

Now suppose the elements of G and H are indexed in order of decreasing weight. Let i be the smallest index such that $w(g_i) < w(h_i)$, and consider the independent sets

$$G_{i-1} = \{g_1, g_2, \ldots, g_{i-1}\} \quad \text{and} \quad H_i = \{h_1, h_2, \ldots, h_{i-1}, h_i\}.$$

By the exchange property, there is some element $h_j \in H_i$ such that $G_{i-1} \cup \{h_j\}$ is an independent set. We have $w(h_j) \geq w(h_i) > w(g_i)$. Thus, the greedy algorithm considers *and rejects* the heavier element h_j before it considers the lighter element g_i. But this is impossible—the greedy algorithm accepts elements in decreasing order of weight. □

We now immediately have a correct greedy optimization algorithm for *any* matroid. Returning to our examples:

- Linear matroid: Given a matrix A, compute a subset of vectors of maximum total weight that span the column space of A.

- Uniform matroid: Given a set of weighted objects, compute its k largest elements.
- Cycle matroid: Given a graph with weighted edges, compute its maximum spanning tree. In this setting, the greedy algorithm is better known as *Kruskal's algorithm*.
- Cocycle matroid: Given a graph with weighted edges, compute its minimum spanning tree.
- Matching matroid: Given a graph, determine whether it has a perfect matching.
- Disjoint path matroid: Given a directed graph with a special vertex s, find the largest set of edge-disjoint paths from s to other vertices.

The exchange condition for matroids turns out to be crucial for the success of this algorithm. A *subset system* is a finite collection $\mathcal{S}$ of finite sets that satisfies the heredity condition—If $X \in \mathcal{S}$ and $Y \subseteq X$, then $Y \in \mathcal{S}$—but not necessarily the exchange condition.

Theorem E.2. *For any subset system $\mathcal{S}$ that is **not** a matroid, there is a weight function w such that* GreedyBasis$(\mathcal{S}, w)$ *does **not** return a maximum-weight set in $\mathcal{S}$.*

Proof: Let X and Y be two sets in $\mathcal{S}$ that violate the exchange property—$|X| > |Y|$, but for any element $x \in X \setminus Y$, the set $Y \cup \{x\}$ is not in $\mathcal{S}$. Let $m = |Y|$. We define a weight function as follows:

- Every element of Y has weight $m + 2$.
- Every element of $X \setminus Y$ has weight $m + 1$.
- Every other element of the ground set has weight zero.

With these weights, the greedy algorithm will consider and accept every element of Y, then consider and reject every element of X, and finally consider all the other elements. The algorithm returns a set with total weight $m(m+2) = m^2 + 2m$. But the total weight of X is at least $(m+1)^2 = m^2 + 2m + 1$. Thus, the output of the greedy algorithm is not the maximum-weight set in $\mathcal{S}$. □

Recall the Applied Chaos scheduling problem considered in Chapter 4. There is a natural subset system associated with this problem: A set of classes is independent if and only if no two classes overlap. (This is just the graph-theory notion of "independent set"!) This subset system is *not* a matroid, because there can be maximal independent sets of different sizes, which violates the exchange property. If we consider a *weighted* version of the class scheduling problem, where each class is worth a different number of hours, Theorem **??** implies that the greedy algorithm will *not* always find the optimal schedule. (In fact, there's an easy counterexample with only two classes!) However, Theorem **??** does *not* contradict the correctness of the greedy algorithm for the original *unweighted* problem; that problem uses a particularly lucky choice of weights (all equal).

E.2 Scheduling with Deadlines

Suppose you have n tasks to complete in n days; each task requires your attention for a full day. Each task comes with a *deadline*, the last day by which the job should be completed, and a *penalty* that you must pay if you do not complete each task by its assigned deadline. What order should you perform your tasks in to minimize the total penalty you must pay?

More formally, you are given an array $D[1..n]$ of deadlines an array $P[1..n]$ of penalties. Each deadline $D[i]$ is an integer between 1 and n, and each penalty $P[i]$ is a non-negative real number. A *schedule* is a permutation of the integers $\{1, 2, \ldots, n\}$. The scheduling problem asks you to find a schedule π that minimizes the following cost:

$$cost(\pi) := \sum_{i=1}^{n} P[i] \cdot [\pi(i) > D[i]].$$

This doesn't look anything like a matroid optimization problem. For one thing, matroid optimization problems ask us to find an optimal *set*; this problem asks us to find an optimal *permutation*. Surprisingly, however, this scheduling problem is actually a matroid optimization in disguise! For any schedule π, call tasks i such that $\pi(i) > D[i]$ *late*, and all other tasks *on time*. The following trivial observation reveals the underlying matroid structure.

The cost of a schedule is determined by the subset of tasks that are on time.

Call a subset X of the tasks ***realistic*** if there is a schedule π in which every task in X is on time. We can precisely characterize the realistic subsets as follows. Let $X(t)$ denote the subset of tasks in X whose deadline is on or before t:

$$X(t) := \{i \in X \mid D[i] \leq t\}.$$

In particular, $X(0) = \varnothing$ and $X(n) = X$.

Lemma E.3. *Let $X \subseteq \{1, 2, \ldots, n\}$ be an arbitrary subset of the n tasks. X is realistic if and only if $|X(t)| \leq t$ for every integer t.*

Proof: Let π be a schedule in which every task in X is on time. Let i_t be the tth task in X to be completed. On the one hand, we have $\pi(i_t) \geq t$, since otherwise, we could not have completed $t-1$ other jobs in X before i_t. On the other hand, $\pi(i_t) \leq D[i]$, because i_t is on time. We conclude that $D[i_t] \geq t$, which immediately implies that $|X(t)| \leq t$.

Now suppose $|X(t)| \leq t$ for every integer t. If we perform the tasks in X in increasing order of deadline, then we complete all tasks in X with deadlines t or less by day t. In particular, for any $i \in X$, we perform task i on or before its deadline $D[i]$. Thus, X is realistic. □

We can now define a ***canonical schedule*** for any set X as follows: execute the tasks in X in increasing deadline order, and then execute the remaining tasks in any order. The previous proof implies that a set X is realistic if and only if every task in X is on time in the canonical schedule for X. Thus, our scheduling problem can be rephrased as follows:

Find a realistic subset X such that $\sum_{i \in X} P[i]$ is maximized.

So we're looking for optimal subsets after all!

Lemma E.4. *The collection of realistic sets of jobs forms a matroid.*

Proof: The empty set is vacuously realistic, and any subset of a realistic set is clearly realistic. Thus, to prove the lemma, it suffices to show that the exchange property holds. Let X and Y be realistic sets of jobs with $|X| > |Y|$.

Let t^* be the largest integer such that $|X(t^*)| \leq |Y(t^*)|$. This integer must exist, because $|X(0)| = 0 \leq 0 = |Y(0)|$ and $|X(n)| = |X| > |Y| = |Y(n)|$. By definition of t^*, there are more tasks with deadline $t^* + 1$ in X than in Y. Thus, we can choose a task j in $X \setminus Y$ with deadline $t^* + 1$; let $Z = Y \cup \{j\}$.

Let t be an arbitrary integer. If $t \leq t^*$, then $|Z(t)| = |Y(t)| \leq t$, because Y is realistic. On the other hand, if $t > t^*$, then $|Z(t)| = |Y(t)| + 1 \leq |X(t)| < t$ by definition of t^* and because X is realistic. The previous lemma now implies that Z is realistic. This completes the proof of the exchange property. □

This lemma implies that our scheduling problem is actually a matroid optimization problem, so the greedy algorithm finds the optimal schedule.

```
GreedySchedule(D[1..n], P[1..n]):
  Sort P in increasing order, and permute D to match
  j ← 0
  for i ← 1 to n
      X[j+1] ← i
      if X[1..j+1] is realistic
          j ← j+1
  return the canonical schedule for X[1..j]
```

To turn this outline into a real algorithm, we need a procedure to test whether a given subset of jobs is realistic. Lemma 9 immediately suggests the following strategy to answer this question in $O(n)$ time.

```
REALISTIC?(X[1..m], D[1..n]):
  ⟨⟨X is sorted by increasing deadline: i ≤ j ⟹ D[X[i]] ≤ D[X[j]]⟩⟩
  N ← 0
  j ← 0
  for t ← 1 to n
      if D[X[j]] = t
          N ← N + 1
          j ← j + 1
      ⟨⟨Now N = |X(t)|⟩⟩
      if N > t
          return FALSE
  return TRUE
```

If we use this subroutine, GREEDYSCHEDULE runs in $O(n^2)$ time. By using some appropriate data structures, the running time can be reduced to $O(n \log n)$; details are left as an exercise for the reader.

Exercises

1. Prove that for any graph G, the "graphic matroid" $\mathcal{M}(G)$ is in fact a matroid. (This problem is really asking you to prove that Kruskal's algorithm is correct!)

2. Prove that for any graph G, the "cographic matroid" $\mathcal{M}^*(G)$ is in fact a matroid.

♦3. Prove that for any graph G, the "matching matroid" of G is in fact a matroid. *[Hint: What is the symmetric difference of two matchings?]*

♦4. Prove that for any directed graph G and any vertex s of G, the resulting "disjoint path matroid" of G is in fact a matroid. *[Hint: This question is **much** easier if you're already familiar with maximum flows.]*

5. Let G be an undirected graph. A set of cycles $\{c_1, c_2, \ldots, c_k\}$ in G is called *redundant* if every edge in G appears in an even number of c_i's. A set of cycles is *independent* if it contains no redundant subset. A maximal independent set of cycles is called a *cycle basis* for G.

   (a) Let C be any cycle basis for G. Prove that for any cycle γ in G, there is a subset $A \subseteq C$ such that $A \cap \{\gamma\}$ is redundant. In other words, γ is the 'exclusive or' of the cycles in A.

   (b) Prove that the set of independent cycle sets form a matroid.

   ♥(c) Now suppose each edge of G has a weight. Define the weight of a cycle to be the total weight of its edges, and the weight of a *set* of cycles to be the total weight of all cycles in the set. (Thus, each edge is counted once for every cycle in

which it appears.) Describe and analyze an efficient algorithm to compute the minimum-weight cycle basis in G.

6. Describe a modification of GREEDYSCHEDULE that runs in $O(n \log n)$ time. *[Hint: Store X in an appropriate data structure that supports the operations "Is $X \cup \{i\}$ realistic?" and "Add i to X" in $O(\log n)$ time each.]*

© 2018 Jeff Erickson http://algorithms.wtf

Man muss immer generalisieren. [One must always generalize.]

— attributed to Carl Gustav Jacob Jacobi (c. 1830) by
Philip J. Davis and Reuben Hersh, *The Mathematical Experience* (1981)

Life is like riding a bicycle. To keep your balance you must keep moving.

— Albert Einstein, in a letter to his son Eduard (February 5, 1930)

Scarcely pausing for breath, Vroomfondel shouted, "We don't demand solid facts! What we demand is a total absence of solid facts. I demand that I may or may not be Vroomfondel!"

— Douglas Adams, *The Hitchhiker's Guide to the Galaxy* (1979)

F

Balances and Pseudoflows

[Read Chapters 10 and 11 first.]

Status: Beta. Needs more figures.

F.1 Unbalanced Flows

In this chapter, we consider a generalization of flows that allows "stuff" to be injected or extracted from the flow network at the vertices. For the moment, consider a flow network $G = (V, E)$ without any specific source and target vertices. Let $b : V \to \mathbb{R}$ be a ***balance*** function describing how much flow should be injected (if the value is positive) or extracted (if the value is negative) at each vertex. We interpret positive balances as ***demand*** or ***deficit*** and negative balances as (negated) ***supply*** or ***excess***.

We now redefine the word ***flow*** to mean a function $f : E \to \mathbb{R}$ that satisfies the modified balance condition

$$\sum_{u \in V} f(u \to v) - \sum_{w \in V} f(v \to w) = b(v)$$

at *every* node v. A flow f is ***feasible*** if it satisfies the usual capacity constraints $0 \le f(e) \le c(e)$ at every edge e. Our problem now is to compute, given a flow network

© 2018 Jeff Erickson http://algorithms.wtf

with edge capacities and vertex balances, either a *feasible* flow in or a proof that no such flow exists.

Note two significant differences from the standard maximum flow problem. First, there are no special source and target vertices that are exempt from the balance constraint. Second, we are not trying to optimize the value of the flow; in fact, the "value" of a flow is no longer even *defined*, because the network has no source and target vertices.

One easy necessary condition is that all the vertex balances must sum to zero; intuitively, every edge $u \to v$ adds the same amount to v's balance that it subtracts from u's balance. More formally, for every feasible flow f, we have

$$\begin{aligned}\sum_v b(v) &= \sum_v \left(\sum_{u \in V} f(u \to v) - \sum_{w \in V} f(v \to w) \right) \\ &= \sum_{u \to v \in E} f(u \to v) - \sum_{v \to w \in E} f(v \to w) = 0.\end{aligned}$$

F.2 Reduction to Maximum Flow

We can reduce the problem of finding a feasible flow in a network with non-zero balance constraints to the standard maximum-flow problem, by adding new vertices and edges to the input graph as follows.

Starting with with original graph G, we construct a new graph $G' = (V', E')$ by adding a new source vertex s with edges to every supply vertex, a new target vertex t with edges from every demand vertex. Specifically, for each vertex v in G, if $b(v) > 0$, we add an edge $v \to t$ with capacity $b(v)$, and if $b(v) < 0$, we add a new edge $s \to v$ with capacity $-b(v)$. Let $c' \colon E' \to \mathbb{R}$ be the resulting capacity function; by construction, $c'|_E = c$.

A flow network G with non-zero balance constraints, and the transformed network G'.

Now call an (s, t)-flow in G' ***saturating*** if every edge leaving s (or equivalently, every edge entering t) is saturated. Every saturating flow is a maximum flow; conversely, either all maximum flows in G' are saturating, or G' has no saturating flow.

Lemma F.1. *G has a feasible flow if and only if G' has a saturating (s, t)-flow.*

Proof: Let $f : E \to \mathbb{R}$ be any feasible flow in G, and consider the function $f' : E' \to \mathbb{R}$ defined as follows:

$$f'(e') = \begin{cases} f(e') & \text{if } e' \in E \\ c'(e') & \text{otherwise} \end{cases}$$

Every edge incident to s or t is saturated, and every edge in E satisfies the capacity constraint $0 \le f'(e) = f(e) \le c(e) = c'(e)$. For each vertex v except s and t, we immediately have

$$\begin{aligned} \sum_{u \in V'} f'(u \to v) - \sum_{w \in V'} f'(v \to w) &= \left(\sum_{u \in V} f(u \to v) - \sum_{w \in V} f(v \to w) \right) - b(v) \\ &= b(v) - b(v) \\ &= 0. \end{aligned}$$

We conclude that f' is a feasible (s,t)-flow in G'. Every edge out of s or into t is saturated by definition, so f' is a saturating flow in G'.

Similarly tedious algebra implies that for any saturating (s,t)-flow $f' : E' \to \mathbb{R}$ through G', the restriction $f = f'|_E$ is a feasible flow in G. □

A feasible flow in G and the corresponding saturating flow in G'.

We emphasize that there are flow networks with no feasible flow, even when the sum of the balances is zero. Suppose we partition the vertices of G into two arbitrary subsets S and T. As usual, let $\|S,T\|$ be the total capacity of the cut (S,T):

$$\|S,T\| = \sum_{u \in S} \sum_{v \in T} c(u \to v).$$

Let $b(S)$ and $b(T)$ denote the sum of the balances of vertices in S and T, respectively:

$$b(S) := \sum_{u \in S} b(u) \qquad \text{and} \qquad b(T) := \sum_{v \in T} b(v).$$

We call the cut (S,T) ***infeasible*** if $\|S,T\| < b(T)$; that is, if T has more demand than can be moved across the cut. The following theorem is a straightforward consequence of the maxflow/mincut theorem (hint, hint):

Theorem F.2. *Every flow network has either a feasible flow or an infeasible cut.*

F.3 Pseudoflows

Instead of reducing our new unbalanced flow problem to the classical maximum flow problem, it is instructive to project the behavior of Ford-Fulkerson on the transformed flow network G' back to the original flow network G. The resulting algorithm no longer maintains and incrementally improves a feasible *flow* in G, but a more general function called a ***pseudoflow***. Formally, a pseudoflow in G is *any* function $\psi : E \to \mathbb{R}$. We say that a pseudoflow ψ is ***feasible*** if $0 \le \psi(e) \le c(e)$ for every edge $e \in E$. A ***flow*** is any pseudoflow that also satisfies the balance constraints at every vertex.

For any pseudoflow ψ in G, we define the residual capacity of any edge $u{\to}v$ as usual:

$$c_\psi(u{\to}v) := \begin{cases} c(u{\to}v) - \psi(u{\to}v) & \text{if } u{\to}v \in E \\ \psi(v{\to}u) & \text{if } v{\to}u \in E \end{cases}$$

We also define the ***residual balance*** of any node to be its original balance minus its net incoming pseudoflow:

$$b_\psi(v) := b(v) - \sum_u \psi(u{\to}v) + \sum_w \psi(v{\to}w).$$

A pseudoflow ψ is a flow if and only if $b_\psi(v) = 0$ for every vertex v. Finally, we define the ***residual network*** G_ψ to be the graph of all edges with positive residual capacity, where the balance of each node v is $b_\psi(v)$. As usual, if ψ is zero everywhere, then G_ψ is the original network G, with its original capacities and balances.

Now we redefine an ***augmenting path*** to be any path in the residual graph G_ψ from any vertex with negative residual balance (supply or excess) to any vertex with positive residual balance (demand or deficit). Pushing flow along an augmenting path decreases the total residual supply

$$B_\psi := \sum_v \left| b_\psi(v) \right|$$

and therefore moves ψ closer to being a feasible flow. The largest amount of flow that we can push along an augmenting path from u to v, before it ceases to be an augmenting path, is the minimum of three quantities:

- The residual supply $-b_\psi(u)$ at the beginning of the path,
- The residual demand $b_\psi(v)$ at the end of the path, and
- The minimum residual capacity among the edges in the path.

On the other hand, if G_ψ contains a vertex v with non-zero residual balance, but does not contain an augmenting path starting or ending at v, then G has no feasible flow. Specifically, if $b_\psi(v) > 0$, then the set S of vertices reachable from v and its complement $T = V \setminus S$ define an infeasible cut; symmetrically, if $b_\psi(v) < 0$, then the set T of vertices that can reach v and its complement $S = V \setminus T$ define an infeasible cut.

Putting all these pieces together, we obtain a simple algorithm for computing either a feasible flow or an infeasible cut. Initialize $\psi(e) = 0$ at every edge e. Then as long

From left to right: A pseudoflow ψ in a flow network G; the residual graph G_ψ with one augmenting path highlighted; and the updated pseudoflow after pushing 4 units along the augmenting path.

as the residual graph G_ψ has a vertex with non-zero balance, update ψ by pushing as much flow as possible along an arbitrary augmenting path. When all residual balances are zero, the current pseudoflow ψ is actual a feasible flow.

```
FeasibleFlow(V, E, c, b):
  for every edge e ∈ E
      ψ(e) ← 0
  B ← Σ_v |b(v)|/2
  while B > 0
      construct G_ψ
      ⟨⟨Find augmenting path π⟩⟩
      s ← any vertex with b_ψ(s) < 0
      if s cannot reach a vertex t in G_ψ with b_ψ(t) > 0
          return Infeasible
      t ← any vertex reachable from s with b_ψ(t) > 0
      π ← any path in G_ψ from s to t
      ⟨⟨Push as much flow as possible along π⟩⟩
      R ← min{−b_ψ(s), b_ψ(t), min_{e∈π} c_ψ(e)}
      B ← B − R
      for every directed edge e ∈ π
          if e ∈ E
              ψ(e) ← ψ(e) + R
          else ⟨⟨rev(e) ∈ E⟩⟩
              ψ(e) ← ψ(e) − R
  return ψ
```

Naturally this algorithm comes with both the same power and the same limitations as Ford-Fulkerson. We can find a single augmenting path in $O(V+E)$ time via whatever-first search. If all capacities and balances are integers, the basic algorithm halts after at most B iterations, where $B = \sum_v |b(v)|/2$, but if we allow irrational capacities **⟨⟨or irrational balances?⟩⟩**, the algorithm could run forever without converging to a flow. Choosing the augmenting path with maximum residual capacity or with the fewest edges leads to

faster algorithms; in particular, if we always choose the shortest augmenting path, the algorithm runs in $O(VE^2)$ time.

F.4 Variations on a Theme

There are several variations on the standard maximum-flow problem, with additional or modified constraints, that can be solved quickly using the pseudoflow formulation. These variations are all solved using a two-stage process:

- First find a *feasible* flow f in the original input graph G.
- Then find a *maximum* flow f' in the residual graph G_f.
- Finally, return the flow $f + f'$.

In each variation, the residual graph G_f we use in the second stage will be a textbook-standard flow network. Notice that Ford-Fulkerson itself can be seen as an example of this two-stage algorithm, where the flow f found in the first stage is zero everywhere, or more subtly, where f is obtained by interrupting Ford-Fulkerson after *any* augmentation step.

Maximum Flows with Non-Zero Balances

Suppose we are given a flow network $G = (V, E)$ with edge capacities $c : E \to \mathbb{R}^+$, non-trivial vertex balances $b : V \to \mathbb{R}$, and two special vertices s and t, and we are asked to compute the maximum (s, t)-flow in this network. In this context, an (s, t)-flow is a function $f : E \to \mathbb{R}^+$ that satisfies the modified balance conditions

$$\sum_w f(v \to w) - \sum_u f(u \to v) = b(v)$$

at every vertex v ***except s and t***. As usual, our goal is to find an (s, t)-flow that maximizes the net flow out of s:

$$|f| = \sum_w f(s \to w) - \sum_u f(u \to s)$$

The algorithms in the previous sections *almost* solve the first stage directly, except for two issues: (1) the terminal vertices s and t are not subject to balance constraints, and (2) the sum of the vertex balances need not be zero. In fact, we can handle both of these issues at once by modifying the graph as follows. First, to avoid any ambiguity, we (re)define $b(s) = b(t) = 0$. Then we add one new vertex z with balance $b(z) = -\sum_v b(v)$, and two new infinite-capacity edges $t \to z$ and $z \to s$. Call the resulting modified flow network G'. Straightforward definition-chasing implies that any feasible flow in G' restricts to a feasible (s, t)-flow in G, and conversely, any feasible (s, t)-flow in G can be extended to a feasible flow in G'.

★★★

Figure!

Thus, we can find a feasible (s,t)-flow f in G in $O(VE^2)$ time by repeatedly pushing flow along shortest augmenting paths, or in $O(VE)$ time using Orlin's maximum-flow algorithm. In the resulting residual graph G_f, every vertex (except at s and t) has residual balance zero, so we can find a maximum flow in G_f using any standard algorithm.

Lower Bounds on Edges

In another standard variant, the input includes a ***lower bound*** $\boldsymbol{\ell(e)}$ on the flow value of each edge e, in addition to the capacity $c(e)$. In this context, a flow f is feasible if and only if $\ell(e) \le f(e) \le c(e)$ for every edge e. In the standard flow problem, we have $\ell(e) = 0$ for every edge e.

Although it is natural to require the lower bounds $\ell(e)$ to be non-negative, we can in fact allow negative lower bounds, and therefore negative flow values $f(e)$, if we interpret negative flow along an edge $u{\to}v$ as positive flow along its reversal $v{\to}u$. More formally, we define a pseudoflow as a function $\psi : E \to \mathbb{R}$ such that

$$\psi(v{\to}u) = -\psi(u{\to}v)$$

for every edge $u{\to}v$; more simply, a pseudoflow is an *antisymmetric* function over the edges. The antisymmetry is reflected in the upper and lower bounds on flow:

$$\ell(v{\to}u) = -c(u{\to}v) \qquad c(v{\to}u) = -\ell(u{\to}v)$$

Then for any pseudoflow ψ, each edge $u{\to}v$ has both a residual capacity and a residual lower bound, defined as follows to maintain antisymmetry:

$$\ell_\psi(u{\to}v) := \begin{cases} \ell(u{\to}v) - \psi(u{\to}v) & \text{if } u{\to}v \in E \\ \psi(u{\to}v) - c(v{\to}u) & \text{if } v{\to}u \in E \end{cases}$$

$$c_\psi(u{\to}v) := \begin{cases} c(u{\to}v) - \psi(u{\to}v) & \text{if } u{\to}v \in E \\ \psi(u{\to}v) - \ell(v{\to}u) & \text{if } v{\to}u \in E \end{cases}$$

Now the residual network G_ψ consists of all edges $u{\to}v$ with non-negative residual capacity $c_\psi(u{\to}v) \ge 0$. We can easily verify (hint, hint) that this antisymmetric formulation of (pseudo)flows and residual graphs is completely consistent with our usual formulation of flows as *non-negative* functions.

Given a flow network with both lower bounds and capacities on the edges, we can compute a maximum (s,t)-flow as follows. If all lower bounds are zero or negative, we can apply any standard maxflow algorithm, after replacing each edge $u{\to}v$ with a negative lower bound with a pair of opposing edges $u{\to}v$ and $v{\to}u$, each with positive capacity.[1] Otherwise, we first define an initial feasible pseudoflow ψ that meets every positive lower bound:

$$\psi(u{\to}v) = \max\left\{\ell(u{\to}v), 0\right\}$$

[1]Wait, didn't we forbid opposing edges two chapters ago? Okay, fine: Replace each edge $u{\to}v$ with a pair of opposing *paths* $u{\to}x{\to}v$ and $v{\to}y{\to}u$, where x and y are new vertices.

Vertices in the resulting residual graph G_ψ may have non-zero residual balance, but every edge has a lower bound that is either zero or negative. Thus, we can compute a maximum (s,t)-flow in G_ψ using the previous two-stage approach, in $O(VE)$ time.

★★★

Figure!

♥F.5 Push-Relabel

★★★

This section needs figures and an editing sweep. Cite Alexander Karzanov (preflows)?

The pseudoflow formulation is the foundation of another family of efficient maximum-flow algorithms that is *not* based on path-augmentation, called ***push-relabel*** or ***preflow-push*** algorithms, discovered by Andrew Goldberg and then refined in collaboration with Robert Tarjan before formal publication in 1986, while Goldberg was still a PhD student.

Every push-relabel algorithm maintains a special type of pseudoflow, called a ***preflow***, in which every node (except s) has non-negative residual balance:

$$b_\psi(v) := \sum_u \psi(u{\to}v) - \sum_w \psi(v{\to}w) \geq 0.$$

We call a vertex ***active*** if it is not the target vertex t and its residual balance is positive; we immediately observe that ψ is actually a *flow* if and only if no vertex is active. The algorithm also maintains a non-negative integer ***height*** $\boldsymbol{ht(v)}$ for each vertex v. We call any edge $u{\to}v$ with $ht(u) > ht(v)$ a ***downward*** edge.

The push-relabel algorithm begins by setting $ht(s) = |V|$ and $ht(v) = 0$ for every node $v \neq s$, and choosing an initial pseudoflow ψ that saturates all edges leaving s and leaves all other edges empty:

$$\psi(u{\to}v) = \begin{cases} c(u{\to}v) & \text{if } u = s \\ 0 & \text{otherwise} \end{cases}$$

Then for as long as there are active nodes, the algorithm repeatedly performs one of the following operations at an arbitrary active node u:

- ***Push:*** For some downward residual edge $u{\to}v$ out of u, increase $\psi(u{\to}v)$ by the minimum of the excess at u and the residual capacity of $u{\to}v$.
- ***Relabel:*** If u has no downward outgoing residual edges, increase the height of u by 1.

It is not at all obvious (at least, it wasn't obvious to me at first) that the push-relabel algorithm correctly computes a maximum flow, or even that it halts in finite time. To prove that the algorithm is correct, we need a series of relatively simple observations.

First, we say that the height function $ht : V \to \mathbb{N}$ and the pseudoflow $\psi : E \to \mathbb{R}$ are ***compatible*** if $ht(u) \leq ht(v) + 1$ for every edge $u{\to}v$ in the residual graph G_ψ.

Lemma F.3. *After each step of the push-relabel algorithm, the height function ht and the pseudoflow ψ are compatible.*

Proof: We prove the lemma by induction. Initially, every residual edge either enters the source vertex s or has both endpoints with height zero, so the initial heights and pseudoflow are compatible. For each later step of the algorithm, there are two cases to consider.

- Just before we push flow along the residual edge $u \to v$, then $u \to v$ must be a downward edge, so $ht(v) < ht(u)$. If the edge $u \to v$ was empty before the push, then this step adds the reversed edge $v \to u$ to the residual graph, and $ht(v) < ht(u) \leq ht(u) + 1$.
- On the other hand, just before we relabel an active vertex v, we must have $ht(v) \leq ht(w)$ for every outgoing residual edge $v \to w$ and (by the induction hypothesis) $ht(u) \leq ht(v) + 1$ for every incoming residual edge $u \to v$. Thus, after relabeling, we have $ht(v) \leq ht(w) + 1$ for every outgoing residual edge $v \to w$ and $ht(u) \leq ht(v) \leq ht(v) + 1$ for every incoming residual edge $u \to v$.

In both cases, compatibility of the new height function and the new pseudoflow follows from the inductive hypothesis. □

Lemma F.4. *After each step of the push-relabel algorithm, there is no residual path from s to t.*

Proof: Suppose to the contrary that there is a simple residual path $v_0 \to v_1 \to \cdots \to v_k$, where $v_0 = s$ and $v_k = t$. This path does not repeat vertices, so $k < |V|$. Because the height of s and t never change, we have $ht(v_0) = |V|$ and $ht(v_k) = 0$. Finally, compatibility implies $ht(v_i) \geq ht(v_{i-1}) - 1$, and thus by induction $ht(v_i) \geq ht(v_0) - i$, for each index i. In particular, we have $ht(v_k) \geq |V| - k > 0$, giving us a contradiction. □

Lemma F.5. ***If** the push-relabel algorithm terminates, it returns a maximum (s, t)-flow.*

Proof: The algorithm terminates only when there are no active vertices, which means that every vertex except s and t has zero residual balance. A pseudoflow with non-zero residual balance is *definition* of a flow! Any flow whose residual graph has no paths from s to t is a maximum (s, t)-flow. □

For a full proof of correctness, we still need to prove that the algorithm terminates, but the easiest way to prove termination is by proving an upper bound on the running time. In the analysis, we distinguish between two types of push operations. A push along edge $u \to v$ is ***saturating*** if we have $\psi(u \to v) = c(u \to v)$ after the push, and ***non-saturating*** otherwise.

Lemma F.6. *The push-relabel algorithm performs at most $O(V^2)$ relabel operations.*

Proof: At every stage of the algorithm, each active node v has a residual path back to s (because we can follow some unit of flow from s to v). Compatibility now implies that the height of each active node is less than $2|V|$. But we only change the height of a node when it is active, and then only upward, so the height of *every* node is less than $2|V|$. We conclude that each node is relabeled at most $2|V|$ times. □

Lemma F.7. *The push-relabel algorithm performs at most $O(VE)$ saturating pushes.*

Proof: Consider an arbitrary edge $u \to v$. We only push along $u \to v$ when $ht(u) > ht(v)$. If this push is saturating, it removes $u \to v$ from the residual graph. This edge reappears in the residual graph only when we push along the reversed edge $v \to u$, which is only possible when $ht(u) < ht(v)$. Thus, between any saturating push through $u \to v$ and the reappearance of $u \to v$ in the residual graph, vertex v must be relabeled at least twice. Similarly, vertex u must be relabeled at least twice before the next push along $u \to v$. By the previous lemma, u and v are each relabeled at most $2|V|$ times. We conclude that there are at most $|V|$ saturating pushes along $u \to v$. □

Lemma F.8. *The push-relabel algorithm performs at most $O(V^2E)$ non-saturating pushes.*

Proof: Define the ***potential*** $\boldsymbol{\Phi}$ of the current residual graph to be the sum of the heights of all active vertices. This potential is always non-negative, because heights are non-negative. Moreover, we have $\Phi = 0$ when the algorithm starts (because every active node has height zero) and $\Phi = 0$ again when the algorithm terminates (because there are no active vertices).

Every relabel operation increases Φ by 1. Every saturating push along $u \to v$ makes v active (if it wasn't already), and therefore increases Φ by at most $ht(v) \leq 2|V|$. Thus, the total potential increase from all relabels and saturating pushes is at most $O(V^2E)$.

On the other hand, every non-saturating push along $u \to v$ makes the vertex u inactive and makes v active (if it wasn't already) and therefore *decreases* the potential by at least $ht(u) - ht(v) \geq 1$.

Because the potential starts and ends at zero, the total potential *decrease* from all non-saturating pushes must equal the total potential *increase* from the other operations. The lemma follows immediately. □

With appropriate elementary data structures, we can perform each push in $O(1)$ time, and each relabel in time proportional to the degree of the node. It follows that the algorithm runs in $\boldsymbol{O(V^2E)}$ ***time***; in particular, the algorithm always terminates with the correct output!

Like the Ford-Fulkerson algorithm, the push-relabel approach can be made more efficient by carefully choosing *which* push or relabel operation to perform at each step. Two natural choices lead to faster algorithms:

- **FIFO:** The active vertices are kept in a standard queue. At each step, we remove the active vertex from the front of the queue, and then either push from or relabel that vertex until it becomes inactive. Any newly active vertices are inserted at the back of the queue. This rule reduces the number of non-saturating pushes to $O(V^3)$, and so the resulting algorithm runs in $\boldsymbol{O(V^3)}$ ***time***.
- **Highest label:** At each step, we either push from or relabel the vertex with maximum height (breaking ties arbitrarily). This rule reduces the number of non-saturating pushes to $O(V^2\sqrt{E})$, and so the resulting algorithm runs in $\boldsymbol{O(V^2\sqrt{E})}$ ***time***.

With more advanced data structures that support pushing flow along more than one edge at a time, the running time of the push-relabel algorithm can be improved to $\boldsymbol{O(VE\log(V^2/E))}$. (This was one of the theoretically-fastest algorithms known before Orlin's algorithm.) In practice, however, this optimization is usually slower than the more basic algorithm that handles one edge at a time.

Exercises

1. Recall from Chapter 11 that a ***path cover*** of a directed acyclic graph is a collection of directed paths, such that every vertex in G appears in at least one path. We previously saw how to compute *disjoint* path covers (where each vertex lies on *exactly* one path) by reduction to maximum bipartite matching. Your task in this problem is to compute path covers *without* the disjointness constraint.
   (a) Suppose you are given a dag G with a unique source s and a unique sink t. Describe an algorithm to find the smallest path cover of G in which every path starts at s and ends at t.
   (b) Describe an algorithm to find the smallest path cover of an arbitrary dag G, with no additional restrictions on the paths.

2. (a) Prove that any flow f in a network $G = (V, E)$ with non-zero balance constraints (and no source or target) can be expressed as a weighted sum of directed paths and directed cycles, such that
      - each path leads from an excess node to a deficit node;
      - a directed edge $u{\to}v$ appears in at least one path or cycle if and only if $f(u{\to}v) > 0$; and
      - the total number of paths and cycles is at most E.
   (b) Describe an algorithm to construct such a decomposition in $O(VE)$ time.

3. Let $G = (V, E)$ be an arbitrary flow network with source s and sink t. Recall that a ***preflow*** is a pseudoflow $\psi : E \to \mathbb{R}$ where every vertex except s has non-negative residual balance; that is, for each vertex v, we have
$$\sum_u \psi(u{\to}v) > \sum_w \psi(v{\to}w).$$

A preflow ψ is *feasible* if $0 \le \psi(u{\to}v) \le c(u{\to}v)$ for every edge $u{\to}v$. A ***maximum*** preflow if is a feasible preflow such that the net flow into t

$$\sum_u \psi(u{\to}t) - \sum_w \psi(t{\to}w)$$

is as large as possible. Describe an algorithm to transform an arbitrary maximum pseudoflow into a maximum flow in $O(VE)$ time. *[Hint: Flow decomposition!]*

4. (a) An ***edge cover*** of an undirected graph $G = (V, E)$ is a subset of edges $C \subseteq E$ such that every vertex is the endpoint of *at least* one edge in C. (Thus, edge covers are the "opposite" of matchings.) Describe and analyze an efficient algorithm to compute the *smallest* edge cover of a given bipartite graph.

   (b) Describe and analyze an algorithm for the following more general problem. The input consists of a bipartite graph $G = (V, E)$ and two functions $\ell, u \colon V \to \mathbb{N}$. Your algorithm should either output a subset of edges $C \subseteq E$ such that each vertex $v \in V$ is incident to at least $\ell(v)$ edges and at most $u(v)$ edges in C, or correctly report that no such subset exists.

5. (a) Suppose we are given a directed graph G, two vertices s and t, and two functions $\ell, c \colon V \to \mathbb{R}$ over the *vertices*. An (s, t)-flow $f : E \to \mathbb{R}$ in this network is feasible if and only if the total flow into each vertex v (except s and t) lies between $\ell(v)$ and $c(v)$:

$$\ell(v) \le \sum_{u \to v} f(u{\to}v) \le c(v).$$

   Describe an efficient algorithm to compute a maximum (s, t)-flow in this network.

   (b) Suppose we are given a directed graph G, two vertices s and t, and two functions $b^-, b^+ \colon V \to \mathbb{R}$ over the vertices. An (s, t)-flow $f : E \to \mathbb{R}$ in this network is feasible if and only if the total ***net*** flow into each vertex v (except s and t) lies between $b^-(v)$ and $b^+(v)$:

$$b^-(v) \le \sum_{u \to v} f(u{\to}v) - \sum_{v \to w} f(v{\to}w) \le b^+(v).$$

   Describe an efficient algorithm to compute a maximum (s, t)-flow in this network.

   (c) Describe an efficient algorithm to compute a maximum (s, t)-flow in a network with *all* of the features we've seen so far:

   - upper and lower bounds on the flow through each edge,
   - upper and lower bounds on the flow into each vertex, and
   - upper and lower bounds on the flow balance at each vertex.

© 2018 Jeff Erickson http://algorithms.wtf

La distance n'y fait rien; il n'y a que le premier pas qui coûte.
[The distance is nothing; it is only the first step that costs.]

— Marie Anne de Vichy-Chamrond, marquise du Deffand,
letter to Jean le Rond d'Alembert, July 7, 1763

Tam quaestiones altioris indaginis poscuntur.
[Then these questions require further investigation.]

— Carl Gustav Jacob Jacobi, unpublished notes (c. 1836)

Cecil Graham: *What is a cynic?*
Lord Darlington: *A man who knows the price of everything, and the value of nothing.*
Cecil Graham: *And a sentimentalist, my dear Darlington, is a man who sees an absurd value in everything and doesn't know the market price of any single thing.*

— Oscar Wilde, *Lady Windermere's Fan, A Play About a Good Woman* (1892)

G

Minimum-Cost Flows

[Read Chapters 8, 10, 11, and F first.]
Status: Text beta. Needs figures.

In this final chapter on flows, we consider a significant generalization of the maximum-flow problem that (as far as we know) cannot be solved by modifying the graph and applying a standard flow algorithm. The input to our problem consists of a flow network without special source and target vertices, where each edge e has a ***cost*** $\$(e)$, in addition to the usual edge capacities and vertex balances. Edge costs can be positive, negative, or zero. The total cost of any flow f is then defined as

$$\$(f) = \sum_{e \in E} \$(e) \cdot f(e).$$

The ***minimum-cost flow*** problem asks for a feasible flow with minimum cost, instead of a feasible flow with maximum value.

G.1 Minimum-Cost Circulations

The simplest special case of the minimum-cost flow problem requires the input flow network to satisfy two constraints:

© 2018 Jeff Erickson

- All vertex balances are zero. Thus, every flow f must satisfy the conservation constraint $\sum_u f(u{\to}v) = \sum_w f(v{\to}w)$ at every vertex v.
- Edges have non-negative capacities and zero lower bounds. Thus, every *feasible* flow f satisfies the capacity constraint $0 \le f(u{\to}v) \le c(u{\to}v)$ at every edge $u{\to}v$.

Because the vertex balances are all zero, any feasible flow in such a network is actually a *circulation*—a sum of directed cycles. This special case is normally called the ***minimum-cost circulation*** problem.

The standard maximum-flow problem can be reduced to the minimum-cost circulation problem as follows. Suppose we are given a flow network $G = (V, E)$ with source and target vertices s and t and non-negative edge capacities $c(e)$. Let $G' = (V, E')$ be the network constructed from G by assigning cost 0 to every edge, and then adding a single edge $t{\to}s$ with infinite capacity and cost -1. Every feasible (s,t)-flow f in G can be extended to a feasible circulation in G' by defining

$$f(t{\to}s) = |f| = \sum_w f(s{\to}w) - \sum_u f(u{\to}s);$$

moreover, the cost of the resulting circulation is precisely $-|f|$. Conversely, for any feasible circulation $f : E' \to \mathbb{R}$, the restriction $f|_E$ is a feasible (s,t)-flow in G with value $-f(t{\to}s)$. Thus, any maximum-value (s,t)-flow in G corresponds to a minimum-cost circulation in G' and vice versa.

★★★

Figure!

Cycle Canceling

We can solve any instance of the minimum-cost circulation problem using a natural generalization of the Ford-Fulkerson augmenting path algorithm called ***cycle canceling***. This algorithm is normally attributed to Morton Klein in 1967, but the key insights date back at least to A. N. Tolstoy's studies of railway transportation networks in the late 1920s.

Consider an arbitrary circulation f in G. As usual, the ***residual network*** G_f consists of all edges $u{\to}v$ with non-zero residual capacity $c_f(u{\to}v)$. For each residual edge $u{\to}v$ in G_f that is not an edge in the original graph G, we define its cost as follows:

$$\$(u{\to}v) := -\$(v{\to}u).$$

★★★

Figure showing flow f and residual graph G_f

Now let γ be a directed cycle in the residual graph G_f. We define the residual capacity $c_f(\gamma)$ of γ to be the minimum residual capacity of the edges in γ, and the cost $\$(\gamma)$ of γ to be the *sum* of the costs of the edges in γ. More concisely:

$$c_f(\gamma) := \min_{e\in\gamma} c_f(e) \qquad \$(\gamma) := \sum_{e\in\gamma} \$(e)$$

Exactly as in Ford-Fulkerson, we can ***augment*** the circulation f, by pushing R units of flow around γ, to obtain a new circulation f':

$$f'(u{\to}v) = \begin{cases} f(u{\to}v) + c_f(\gamma) & \text{if } u{\to}v \in \gamma \\ f(u{\to}v) - c_f(\gamma) & \text{if } v{\to}u \in \gamma \\ f(u{\to}v) & \text{otherwise} \end{cases}$$

Straightforward calculation now implies that

$$\$(f') = \$(f) + c_f(\gamma) \cdot \$(\gamma).$$

In particular, if $\$(\gamma) < 0$, the new circulation f' has lower cost than the original circulation f. We immediately conclude that ***a feasible circulation f is a minimum-cost circulation if and only if G_f contains no negative cycles.***

Figure showing negative cycle and updated flow f'

★★★

Finally, Klein's cycle canceling algorithm initializes f to the all-zero circulation, and then repeatedly augments f along an arbitrary negative residual cycle, until there are no more negative residual cycles. For the special instances generated by our reduction from maximum flow, every negative-cost residual cycle consists of a path from s to t and the sole negative edge $t{\to}s$, so cycle canceling is equivalent to Ford and Fulkerson's augmenting path algorithm.

Analysis

In each iteration of the algorithm, we can find a negative-cost cycle in $O(VE)$ time using a straightforward modification of the Shimbel-Bellman-Ford shortest path algorithm (hint, hint). If both the capacity and the cost of each edge is an integer, then each cycle-augmentation decreases the cost of the circulation by a positive integer, and therefore by at least 1. It follows that the cycle canceling algorithm runs in $\boldsymbol{O(VE \cdot |\$(f^*)|)}$ ***time***, where f^* is the final circulation. The crude bound $\$(f^*) \geq -ECK$, where $C = \max_e c(e)$ and $K = \max_e |\$(e)|$, also implies that the algorithm runs in at most $\boldsymbol{O(VE^2CK)}$ ***time***. As one might expect from Ford-Fulkerson, these time bounds are exponential in the complexity of the input even when capacities are integers, both time bounds are tight in the worst case, and the algorithm may not terminate or even approach a minimum-cost circulation in the limit if any capacities are irrational.[1]

Like Ford-Fulkerson, more careful choices of *which* cycle to cancel lead to more efficient algorithms. Unfortunately, some natural choices are NP-hard to compute,

[1]If the input network has integer capacities, the algorithm must eventually terminate even when costs are irrational, because there are only a finite number of feasible integer circulations. However, the number of iterations could still be exponential, even if we always cancel the cycle with most negative total cost!

including the cycle with the most negative total cost and the negative cycle with the fewest edges. In the late 1980s, Andrew Goldberg and Bob Tarjan developed a minimum-cost flow algorithm that repeatedly cancels the so-called ***minimum-mean cycle***, which is the cycle whose *average cost per edge* is smallest. By combining an algorithm of Karp to compute minimum-mean cycles in $O(VE)$ time, efficient dynamic tree data structures, and other sophisticated techniques that are (unfortunately) beyond the scope of this class, their algorithm achieves a running time of $\boldsymbol{O(VE^2 \log V)}$.

Reduction from General Minimum-Cost Flow

In the standard general formulation of the minimum-cost flow problem, we are given a directed flow network $G = (V, E)$ with the following additional data:

- a capacity function $c : E \to \mathbb{R}$ such that $c(e) \geq 0$ for all $e \in E$;
- a lower bound function $\ell : E \to \mathbb{R}$ such that $0 \leq \ell(e) \leq c(e)$ for all $e \in E$;
- a balance function $b : V \to \mathbb{R}$ such that $\sum_v b(v) = 0$; and
- a cost function $\$: E \to \mathbb{R}$.

As usual, a feasible flow in G is a function $f : E \to \mathbb{R}$ that satisfies the capacity constraints $\ell(e) \leq f(e) \leq c(e)$ at every edge e and the balance constraints $\sum_u f(u{\to}v) - \sum_w f(v{\to}w) = b(v)$ at every vertex v. The minimum-cost flow problem asks for a feasible flow f with minimum total cost $\$(f) = \sum_e \$(e) \cdot f(e)$.

We can solve the general minimum-cost flow problem in two stages. First, find *any* feasible flow f in G, by reducing to the maximum-flow problem as described in the previous chapter. If there is no feasible flow in G, we can immediately report failure. Otherwise, in the second stage, we compute a minimum-cost circulation f' in the residual graph G_f, using (for example) cycle canceling. Tedious definition-chasing implies that the function $f + f'$ is a minimum-cost flow in the original network G. In total, this algorithm runs in $\boldsymbol{O(VE^2 \log V)}$ ***time***.

G.2 Successive Shortest Paths

The cycle canceling algorithm implies a natural characterization of minimum-cost flows. Consider a general flow network $G = (V, E)$ with capacities, lower bounds, costs, and balances. A pseudoflow $\psi : E \to \mathbb{R}$ is a minimum-cost flow if and only if it satisfies three conditions:

- **Feasible:** $\ell(e) \leq \psi(e) \leq c(e)$ for every edge e.
- **Balanced:** $\sum_u \psi(u{\to}v) - \sum_w \psi(v{\to}w) = b(v)$ for every vertex v.
- **Locally optimal:** The residual graph G_ψ contains no negative-cost cycles.

The cycle-canceling algorithm incrementally improves a feasible and balanced pseudoflow (that is, a circulation) until it is also locally optimal. A complementary strategy called ***successive shortest paths*** incrementally improves a feasible and *locally optimal*

pseudoflow until it is also *balanced*. The successive shortest paths algorithm was initially proposed by Ford and Fulkerson in the mid-1950s, and then later rediscovered by William Jewell in 1958, Masao Iri in 1960, and Robert Busacker and Paul Gowen in 1960.

The successive shortest paths algorithm requires two simplifying assumptions about the initial flow network G:

- **All lower bounds are zero.** Otherwise, think of the lower-bound function ℓ as a pseudoflow (sending $\ell(e)$ units of flow across each edge e), and consider the residual graph G_ℓ.
- **All costs are non-negative.** Otherwise, consider the residual graph of the following pseudoflow:

$$\bar{c}(u{\to}v) = \begin{cases} c(u{\to}v) & \text{if } \$(u{\to}v) \le 0 \\ 0 & \text{otherwise} \end{cases}$$

Enforcing these assumptions may introduce non-zero balances at the vertices, even if all balances in the initial flow network are zero. These two assumptions imply that the all-zero pseudoflow is both feasible and locally optimal, but not necessarily balanced.

The successive shortest path algorithm begins by initializing ψ to the all-zero pseudoflow, and then repeatedly augments ψ by pushing flow along a *shortest* path in the residual graph G_ψ from any vertex with negative residual balance to any vertex with positive residual balance, where the length of a path in G_ψ is the sum of the residual costs of its edges. The algorithm is described in more detail below:

```
SuccessiveShortestPaths(V, E, c, b, $):
  for every edge e ∈ E
      ψ(e) ← 0
  B ← Σ_v |b(v)|/2
  while B > 0
      construct G_ψ
      s ← any vertex with b_ψ(s) < 0
      t ← any vertex with b_ψ(t) > 0
      σ ← shortest path in G_ψ from s to t
      Augment(s, t, σ)
  return ψ
```

```
Augment(s, t, σ):
  R ← min{−b_ψ(s), b_ψ(t), min_{e∈σ} c_ψ(e)}
  B ← B − R
  for every directed edge e ∈ σ
      if e ∈ E
          ψ(e) ← ψ(e) + R
      else ⟨⟨rev(e) ∈ E⟩⟩
          ψ(e) ← ψ(e) − R
```

(To simplify the pseudocode, I have assumed implicitly that the residual graph G_ψ is always strongly connected. If necessary, this assumption can be enforced by adding a new vertex with zero balance and with infinite-cost edges to and from every other vertex in G. If SuccessiveShortestPaths ever tries to push flow along one of these new edges, we can immediately report that the original flow problem is infeasible.)

SuccessiveShortestPaths is *obviously* an instantiation of the FeasibleFlow algorithm from the previous chapter; the only difference is that we require each augmenting path σ to be a shortest path, instead of an arbitrary path. So *obviously* the

algorithm either returns a feasible flow (that is, a feasible and balanced pseudoflow) or reports correctly that no such flow exists.

Unfortunately, like almost(?) every other sentence that uses the words "obviously", the previous argument misses a crucial subtlety: Shortest paths are only *well-defined* in graphs without negative cycles! To argue that SUCCESSIVESHORTESTPATHS is even a well-defined algorithm, much less a *correct* algorithm, we need to prove the following lemma:

Lemma G.1. *After every iteration of the main loop of SUCCESSIVESHORTESTPATHS, the feasible pseudoflow ψ is locally optimal; that is, the residual graph G_ψ contains no negative cycles.*

Proof: We prove the lemma by induction. The base case is immediate; the initial all-zero pseudoflow is locally optimal because the graph G has no negative edges.

Suppose that ψ is locally optimal at the beginning of an iteration of the main loop. By definition, there are no negative cycles in the residual graph G_ψ, so shortest paths are well-defined. Let s be an arbitrary supply vertex and let t be an arbitrary demand vertex in G_ψ. Let σ be a shortest path in G_ψ, and let ψ' be the resulting pseudoflow after augmenting along σ. We need to prove that ψ' is locally optimal.

For the sake of argument, suppose the residual graph $G_{\psi'}$ contains a negative cycle γ. Let $f = \sigma + \gamma$ denote the pseudoflow in G_ψ obtained by sending one unit of flow along σ and then one unit of flow along γ. Although γ may contain edges that are not in G_ψ, all such edges must be reversals of edges in σ; thus, f is positive only on edges in G_ψ. Routine calculations imply that f is an integral (s,t)-flow in G_ψ with value 1.[2] Thus, by the Flow Decomposition Lemma, f can be decomposed into a single path σ' from s to t and possibly one or more cycles $\gamma_1, \ldots, \gamma_k$. Our assumption that ψ is locally optimal implies that each cycle γ_i has non-negative cost, and therefore $\$(\sigma') \le \$(f) < \$(\sigma)$. But this is impossible, because σ is a *shortest* path from s to t. □

★★★

Figure with a complicated instance of σ and γ.

For the special instances of minimum-cost flow generated by our reduction from maximum flow, the only supply node is the source vertex s, the only deficit node is the target vertex t, and every edge in the network has cost 0 except the edge $s \to t$, which has cost 1. Thus, for these instances, the successive shortest paths is equivalent to Ford and Fulkerson's augmenting path algorithm (followed by a single augmentation of the edge $s \to t$).

[2] Here I'm ignoring the residual balances in G_ψ. Note that f may not be a *feasible* flow in G_ψ, but that doesn't matter.

Analysis

In each iteration of the algorithm, we can find a shortest in $O(VE)$ time using a the Shimbel-Bellman-Ford shortest path algorithm. (Actually, we can use Dijkstra's algorithm to compute the *first* shortest path, but augmenting along a path of positive-cost edges introduces residual edges with negative costs, so we can't use Dijkstra's algorithm in later iterations.)

Assuming the capacity of each edge is an integer, then each augmentation decreases the total absolute balance B by a positive integer, and therefore by at least 1. It follows that the cycle canceling algorithm halts after at most B iterations and thus runs in ***$O(VEB)$ time***. As one might expect from Ford-Fulkerson, this time bound is exponential in the complexity of the input when capacities are integers, the time bound are tight in the worst case, and the algorithm may not terminate or even approach a minimum-cost circulation in the limit if any capacities are irrational.

G.3 Node Potentials and Reduced Costs

The main bottleneck in Ford and Fulkerson's successive shortest paths algorithm is computing a shortest paths in every iteration. Because some edges in the residual graph may have negative costs, we are forced to use Bellman-Ford instead of Dijkstra's algorithm. Or so it seems.

The following clever trick to speed up successive shortest paths was proposed by Jack Edmonds and Richard Karp in 1969 (published in 1972) and independently by Nobuaki Tomizawa in 1970 (published in 1971). The same trick was later applied by Donald Johnson in the late 1970s to speed up all-pairs shortest-path algorithms; Johnson attributes the trick to Edmonds and Karp.

Suppose each vertex v in the flow network has been assigned an arbitrary real ***potential $\pi(v)$***. We define the ***reduced cost*** of each edge with respect to potential function π as follows:

$$\$^{\pi}(u \to v) \;:=\; \pi(u) + \$(u \to v) - \pi(v).$$

Then for any path α from u to v, we immediately have

$$\$^{\pi}(\alpha) \;=\; \sum_{e \in \alpha} \$^{\pi}(e) \;=\; \pi(u) + \sum_{e \in \alpha} \$(e) - \pi(v) \;=\; \pi(u) + \$(\alpha) - \pi(v).$$

The potentials of all the intermediate vertices on α cancel out; intuitively, for each intermediate vertex x, we get an "entrance gift" of $\pi(x)$ when the path enters x, which we immediately pay back as an "exit tax" when the path leaves x. This simple observation has two important consequences:

- Shortest paths with respect to the reduced cost function $\$^{\pi}$ are identical to shortest paths with respect to the original cost function $\$$.

- For any cycle γ, we immediately have $\$^{\pi}(\gamma) = \$(\gamma)$; thus, a flow network G has no cycles with negative cost if and only if G has no cycles with negative *reduced* cost.

Edmonds, Karp, and Tomizawa observed that if we choose the right potential function, we can ensure that all reduced edge costs are non-negative, allowing us to compute shortest paths using Dijkstra's algorithm instead of Bellman-Ford.

Lemma G.2. *A flow network G has no negative-cost cycles if and only if, for some potential function $\pi : V \to \mathbb{R}$, every edge in G has non-negative reduced cost.*

Proof: One direction is easy. Suppose G contains a cycle γ with negative cost. Then for every potential function π, we have $\$^{\pi}(\gamma) = \$(\gamma) < 0$, which implies that at least one edge in γ has negative reduced cost.

Conversely, suppose G has no negative-cost cycles. Then shortest path distances with respect to the cost function $\$$ are well-defined. Fix an arbitrary vertex s, and for every vertex v, let $\boldsymbol{dist(v)}$ denote the shortest-path distance from s to v. (If necessary, add zero-capacity, high-cost edges from s to every other vertex, so that these distances are all finite.) Then for every edge $u \to v$, we immediately have

$$dist(v) \leq dist(u) + \$(u \to v),$$

since otherwise there would be a path from s through u to v with cost less than $dist(v)$; this inequality can be rewritten as

$$dist(u) + \$(u \to v) - dist(v) \geq 0.$$

Thus, if use these shortest-path distances as potentials, every edge in G has non-negative reduced cost. □

Speeding up Successive Shortest Paths

Now we seem to have a chicken-and-egg problem: We need a good potential function to compute shortest paths quickly, but we need shortest paths to compute a good potential function!

We can break this apparent cycle as follows. Because the original flow network G has only non-negative edge costs, we can start with $\pi(v) = 0$ at every vertex v. Then after each iteration of the main loop, we update the vertex potentials to ensure that reduced costs in the new residual graph are still non-negative. Specifically, ***we use the shortest-path distances from each iteration as the vertex potentials in the next iteration***, as described in the pseudocode below.

```
FastSuccessiveShortestPaths(V, E, c, b, $):
  for each edge e ∈ E
      ψ(e) ← 0
  for each vertex v ∈ V
      π(e) ← 0
  B ← Σ_v |b(v)|/2
  while B > 0
      construct G_ψ
      s ← any vertex with b_ψ(s) < 0
      t ← any vertex with b_ψ(t) > 0
      dist ← Disjkstra(G_ψ, s, $^π)
      σ ← shortest path in G_ψ from s to t    ⟨⟨computed in the previous line⟩⟩
      Augment(s, t, σ)
      for each vertex v ∈ V
          π(v) ← dist(v)
  return ψ
```

Lemma G.3. *After every iteration of the main loop of* FastSuccessiveShortestPaths*, every edge $u{\to}v$ in the residual graph G_ψ has non-negative reduced cost: $\$^\pi(u{\to}v) \geq 0$.*

Proof: We prove the lemma by induction. The base case is immediate, because the original graph G has no negative-cost edges.

Suppose at the beginning of some iteration that for every edge $u{\to}v$ in G_ψ, we have $\$^\pi(u{\to}v) \geq 0$. Let s be an arbitrary supply vertex, and for any vertex v, let $dist(v)$ denote the shortest-path distance from s to v with respect to the reduced cost function $\$^\pi$. Let σ be a shortest path in G_ψ from s to some demand vertex t, and let ψ' be the resulting pseudoflow after augmenting along σ.

Now let $u{\to}v$ be an arbitrary edge in the new residual graph $G_{\psi'}$. We need to prove that $\$^{dist}(u{\to}v) \geq 0$. There are two cases to consider:

- Suppose $u{\to}v$ is an edge in the old residual graph G_ψ. Then following the proof of Lemma 2, we immediately have $\$^{dist}(u{\to}v) = dist(u) + \$^\pi(u{\to}v) - dist(v) \geq 0$.
- Suppose $u{\to}v$ is not an edge in the old residual graph G_ψ. Then $u{\to}v$ must be the reversal of an edge in σ. It follows that $dist(u) = dist(v) + \$^\pi(v{\to}u)$, and therefore

$$\begin{aligned}
\$^{dist}(u{\to}v) &:= dist(u) + \$^\pi(u{\to}v) - dist(v) \\
&= dist(u) + \pi(u) + \$(u{\to}v) - \pi(v) - dist(v) \\
&= dist(u) + \pi(u) - \$(v{\to}u) - \pi(v) - dist(v) \\
&= dist(u) - \$^\pi(v{\to}u) - dist(v) \\
&= 0.
\end{aligned}$$

In both cases, we conclude that $\$^{dist}(u{\to}v) \geq 0$, as required. □

This lemma implies that we can compute shortest paths at every iteration of FAST-SUCCESSIVESHORTESTPATHS using Disjktra's algorithm in $O(E \log V)$ time. It follows that the overall algorithm runs in $\boldsymbol{O(BE \log V)}$ ***time***.

The fastest minimum-cost flow algorithm currently known (at least among algorithms whose running times depend only on V and E), due to James Orlin in the early 1990s, reduces the problem to $O(E \log V)$ iterations of Dijkstra's shortest-path algorithm and therefore runs in $\boldsymbol{O(E^2 \log^2 V)}$ ***time***.

G.4 Transshipment and Transportation

The ***transshipment*** and ***transportation*** problems are two interesting special cases of minimum-cost flows, where the flow values are not directly limited by capacity constraints on the edges, but rather indirectly limited by the balance conditions at the vertices.

A transshipment network is a directed graph $G = (V, E)$ with vertex balances $b : V \to \mathbb{R}$ and edge costs $\$: E \to \mathbb{R}$, but *no* capacity function; each edge is allowed to carry *any* non-negative amount of flow. Thus, a feasible flow in such a network is a function $f : E \to \mathbb{R}$ such that $f(e) \geq 0$ for every edge e and $\sum_u f(u \to v) - \sum_w f(v \to w) = b(v)$ for every vertex v. As usual, the transshipment problem asks for a feasible flow with minimum total cost $\$(f) = \sum_e \$(e) \cdot f(e)$.

The transshipment problem was first cast in this discrete form by Alex Orden in 1956. However, transshipment has been a common practice in long-distance commerce for as long as there has been long-distance commerce.

Traditionally, the vertices in a transshipment network are divided into three categories: *supply* or *source* vertices have negative balance; *demand* or *target* vertices have positive balance, and ***transshipment*** vertices have zero balance. The *transportation* problem is a special case of the transshipment problem in which the input satisfies two additional conditions:

- There are no transshipment nodes; every vertex has a non-zero balance constraint.
- Every edge $u \to v$ leaves a supply node u and enters a demand node v. Thus, the flow network is a directed bipartite graph, where the vertex partition separates supply vertices from demand vertices.

Some formulations of the transportation problem require G to be a *complete* bipartite graph, but if necessary we can introduce any missing edges with infinite cost.

The transportation problem was first cast in this modern form by Frank Hitchcock in 1942 and independently by Tjalling Koopmans in 1947; however, earlier versions of the problem were studied by Leonid Kantorivich in the late 1930s, A. N. Tolstoy in the 1920s and 1930s, and even earlier (as a continuous mass-transport problem) by Gaspard Monge in the late 1700s.

Reducing Minimum-Cost Flow to Transportation

Like the minimum-cost circulation problem, is is possible to reduce arbitrary instances of minimum-cost flow to the transportation problem; the following reduction was described by Delbert Fulkerson in 1960, generalizing Orden's 1956 reduction from the transshipment problem. Let $G = (V, E)$ be an arbitrary flow network, where every edge e has an associated lower bound $\ell(e)$, capacity $c(e)$, and cost $\$(e)$, and every vertex v has an associated balance $b(v)$.

We can remove the capacity constraints from any single edge $u \to v$ as follows. First, replace the edge with a pair of edges $u \to z_{uv}$ and $v \to z_{uv}$, where z_{uv} is a new vertex. Next, set the various constraints for the new vertex and edges as follows:

- $\ell(u \to z_{uv}) = \ell(v \to z_{uv}) = 0$,
- $c(u \to z_{uv}) = c(v \to z_{uv}) = \infty$,
- $\$(u \to z_{uv}) = \$(u \to v)$ and $\$(v \to z_{uv}) = 0$, and
- $b(z_{uv}) = c(u \to v) - \ell(u \to v)$;

Finally, increase the balance $b(u)$ by $\ell(u \to v)$ and decrease the balance $b(v)$ by $c(u \to v)$. Call the resulting flow network G'.

Removing capacity constraints from a single edge.

Given any feasible flow f in the original network G, we can construct a corresponding flow f' in G' by setting

$$f'(e) = \begin{cases} f(u \to v) - \ell(u \to v) & \text{if } e = u \to z_{uv}, \\ c(u \to v) - f(u \to v) & \text{if } e = v \to z_{uv}, \\ f(e) & \text{otherwise.} \end{cases}$$

Routine calculations imply that f' is a feasible flow in G' with cost $\$(f) - \ell(u \to v) \cdot \$(u \to v)$. Conversely, given a feasible flow f' in G', we can construct a corresponding flow f in G by setting

$$f(e) = \begin{cases} f'(u \to z_{uv}) + \ell(u \to v) & \text{if } e = u \to v, \\ f'(e) & \text{otherwise.} \end{cases}$$

Again, routine calculations imply that f is a feasible flow in G with cost $\$(f) + \ell(u \to v) \cdot \$(u \to v)$. We conclude, for both flow transformations, that f is a minimum-cost flow in G if and only if f' is a minimum cost flow in G'.

Applying this transformation to every edge in G, we obtain a bipartite transportation network H with $V + E$ vertices and $2E$ uncapacitated edges in $O(E)$ time. Given a minimum-cost flow in H, we can easily recover a minimum-cost flow in the original network G in $O(E)$ time.

Special case: Assignment

We conclude by describing two further interesting special cases of the transshipment/transportation problem.

The ***assignment*** problem is a special case of the Hitchcock-Koopman's transportation problem in which every supply vertex has balance -1 and every demand vertex has balance $+1$. In other words, the assignment problem asks for a ***minimum-weight perfect matching*** in a bipartite graph with weighted edges (where "weight" is what we previously called "cost"). The assignment problem can also be reformulated as a ***maximum***-weight perfect matching problem by negating all edge weights.

The assignment problem was first posed *and solved* by the German mathematician Carl Jacobi in the early 1800s, motivated by the problem of solving a system of ordinary differential equations. Jacobi's algorithm was not published until after his death.[3] Jacobi's algorithm was rediscovered by Harold Kuhn in the mid 1950s. Kuhn called his algorithm the "Hungarian method", because its relied on combinatorial results on perfect matchings published by Hungarian mathematicians Dénes Kőnig and Jenő Egerváry in 1931. (Alas, Jacobi was not Hungarian.) In 1957, topologist James Munkres observed that the Jacobi-Egerváry-Kuhn algorithm runs in $O(V^4)$ time; with more care, the running time can be improved to $O(V^3)$.

Any instance of the transportation problem *with integer balances* can be transformed into an equivalent instance of the assignment problem, by replacing each vertex v with $|b(v)|$ equivalent copies.

Special Case: Single-Supply Transshipment

Perhaps the most important special case of the *transhipment* problem requires that the network has exactly one supply node s. This special case is better known as the ***single-source shortest path*** problem. Specifically, let T be a shortest-path tree of G rooted at the supply vertex s, and let $dist(v)$ denote the shortest-path distance from s to v. We define a canonical flow $f_T : E \to \mathbb{R}$ for this spanning tree as follows:

$$f_T(u{\to}v) = \begin{cases} \sum_{w \downarrow v} b(w) & \text{if } u{\to}v \in T \\ 0 & \text{otherwise} \end{cases}$$

[3]Carl Gustav Jacob Jacobi. De investigando ordine systematis aequationum differentialum vulgarium cujuscunque. *J. Reine Angew. Math.* 64(4):297–320, 1865. Posthumously published by Carl Borchardt. English translation by François Ollivier: Looking for the order of a system of arbitrary ordinary differential equations. *Applicable Algebra in Engineering, Communication and Computing*, 20(1):7–32, 2009.

(Here $w \downarrow v$ indicates that w is a descendant of v in T.) Equivalently, f_t) is the sum of $V-1$ path flows, each sending $b(v)$ units of flow along the unique path from s to some other vertex v; thus,

$$\$(f_T) = \sum_v dist(v) \cdot b(v).$$

We can prove this canonical flow is optimal in (at least) two different ways. First, f_T is precisely the minimum-cost flow constructed by the successive shortest-path algorithm (at least if the shortest path tree T is unique). Second, following the proof of Lemma 1, if the residual graph G_{f_T} contained any negative cycles, we could find a shorter path from s to some other vertex v.

Ford's generic relaxation algorithm to compute shortest path trees is morally equivalent to cycle canceling. Ford's algorithm maintains a tentative shortest path tree T. Any tense edge $u \to v$ indicates a negative cycle in the residual graph $G_T = G_{f_T}$, consisting of $u \to v$ and the unique path from v to u in T. Relaxing $u \to v$ replaces the unique edge $x \to v$ in T with $u \to v$.

In fact, for *every* transhipment network G without negative cycles, there is an minimum-cost flow is non-zero only on the edges of some spanning tree of G.

Exercises

Need more!

★★★

1. Describe and analyze an algorithm for the following problem, first posed and solved by the German mathematician Carl Jacobi in the early 1800s.

   > *Disponantur nn quantitates $h_k^{(i)}$ quaecunque in schema Quadrati, ita ut k habeantur n series horizontales et n series verticales, quarum quaeque est n terminorum. Ex illis quantitatibus eligantur n transversales, i.e. in seriebus horizontalibus simul atque verticalibus diversis positae, quod fieri potest* $1.2 \ldots n$ *modis; ex omnibus illis modis quaerendum est is, qui summam n numerorum electorum suppeditet maximam.*

   For the tiny minority of students who are not fluent in mid-19th century academic Latin, here is a modern English formulation of Jacobi's problem. Suppose we are given an $n \times n$ matrix M. Describe and analyze an algorithm that computes a permutation σ that maximizes the sum $\sum_{i=1}^{n} M_{i,\sigma(i)}$, or equivalently, permutes the columns of M so that the sum of the elements along the diagonal is as large as possible.

   Please write a complete, self-contained solution in English, not in mid-19th century academic Latin. Or Hungarian.

2. An ***(s, t)-series-parallel*** graph is an directed acyclic graph with two designated vertices s (the *source*) and t (the *target* or *sink*) and with one of the following structures:

- **Base case:** A single directed edge from s to t.
- **Series:** The union of an (s, u)-series-parallel graph and a (u, t)-series-parallel graph that share a common vertex u but no other vertices or edges.
- **Parallel:** The union of two smaller (s, t)-series-parallel graphs with the same source s and target t, but with no other vertices or edges in common.

Recall that any series-parallel graph can be represented by a binary *decomposition tree*, whose interior nodes correspond to series compositions and parallel compositions, and whose leaves correspond to individual edges. The decomposition tree can be constructed in $O(V + E)$ time.

(a) Describe an efficient algorithm to compute a *minimum-cost* maximum flow from s to t in an (s, t)-series-parallel graph whose edges have *unit* capacity and arbitrary costs.

♥(b) Describe an efficient algorithm to compute a *minimum-cost* maximum flow from s to t in an (s, t)-series-parallel graph whose edges have *arbitrary* capacities and costs.

3. Every year, Professor Dumbledore assigns the instructors at Hogwarts to various faculty committees. There are n faculty members and c committees. Each committee member has submitted a list of their *prices* for serving on each committee; each price could be positive, negative, zero, or even infinite. For example, Professor Snape might declare that he would serve on the Student Recruiting Committee for 1000 Galleons, that he would *pay* 10000 Galleons to serve on the Defense Against the Dark Arts Course Revision Committee, and that he would not serve on the Muggle Relations committee for any price.

   Conversely, Dumbledore knows how many instructors are needed for each committee, as well as a list of instructors who would be suitable members for each committee. (For example: "Dark Arts Revision: 5 members, anyone but Snape.") If Dumbledore assigns an instructor to a committee, he must pay that instructor's price from the Hogwarts treasury.

   Dumbledore needs to assign instructors to committees so that (1) each committee is full, (3) no instructor is assigned to more than three committees, (2) only suitable and willing instructors are assigned to each committee, and (4) the total cost of the assignment is as small as possible. Describe and analyze an efficient algorithm that either solves Dumbledore's problem, or correctly reports that there is no valid assignment whose total cost is finite.

4. Vince wants to borrow a certain amount of money from his friends as cheaply as possible, possibly after first arranging a sequence of intermediate loans. Each of Vince's friends have a different amount of money that they can lend (possibly zero). For any two people x and y, there is a maximum amount of money (possibly zero

or infinite) that x is willing to lend to y and a certain profit (possibly zero or even negative) that x expects from any loan to y.

For example, suppose Vince wants to borrow \$100 from his friends Ben and Naomi, who have the following constraints:

- Ben has \$500 available to lend.
- Ben is willing to lend up to \$150 to Vince at a profit of 20¢ per dollar.
- Ben is willing to lend up to \$50 to Naomi, at a loss of 10¢ per dollar.
- Naomi has \$50 available to lend.
- Naomi is willing to lend any amount of money to Vince, at a profit of 10¢ per dollar.
- Naomi is not willing to lend money to Ben.

If Vince borrows \$100 directly from Ben, he needs \$120 to pay off the loan. If Vince borrows \$50 from Ben and \$50 from Naomi, he needs \$115 to pay off the loan: \$60 for Ben and \$55 for Naomi. But if Vince asks Naomi to borrow \$50 from Ben and then borrows the entire \$100 from Naomi, then he needs only \$110 to pay off Naomi, who can then pay off Ben with just \$45. With the same constraints, the maximum amount of money that Vince can borrow is \$250.

Describe and analyze an algorithm that finds a sequence of loans that minimizes the amount Vince needs to pay everyone off, or correctly reports that Vince cannot borrow his desired amount. The input has the following components:

- An array $Money[1..n]$, where $Money[i]$ is the amount of money that friend i has.
- An array $MaxLoan[1..n, 0..n]$, where $MaxLoan[i, j]$ is the amount of money that friend i is willing to lend to friend j. "Friend 0" is Vince.
- An array $Profit[1..n, 0..n]$, where $Profit[i, j]$ is the profit per dollar that friend i expects from any load to friend j. Again, "friend 0" is Vince.
- The total amount T that Vince wants to borrow.

5. Suppose you are given a directed flow network G, in which every edge has an *integer* capacity and an *integer* cost, and each vertex has an *integer* balance, along with an *integer* minimum-cost flow f^* in G. Describe algorithm for the following problems:

   (a) IncCost($u \to v$): Increase the cost of $u \to v$ by 1 and update the minimum-cost flow.

   (b) DecCost($u \to v$): Decrease the cost of $u \to v$ by 1 and update the minimum-cost flow.

   Both algorithms should modify f^* so that it is still a minimum-cost flow, more quickly than recomputing a minimum-cost flow from scratch.

The greatest flood has the soonest ebb; the sorest tempest the most sudden calm; the hottest love the coldest end; and from the deepest desire oftentimes ensues the deadliest hate.

— Socrates

Th' extremes of glory and of shame, Like east and west, become the same.

— Samuel Butler, *Hudibras* Part II, Canto I (c. 1670)

Everything is dual; everything has poles; everything has its pair of opposites; like and unlike are the same; opposites are identical in nature, but different in degree; extremes meet; all truths are but half-truths; all paradoxes may be reconciled.

— *The Kybalion: A Study of The Hermetic Philosophy of Ancient Egypt and Greece* (1908)

Oh, that!

— John von Neumann to George Dantzig (May 1947)

H

Linear Programming

[Read Chapters F and G first.]

Status: Text beta. Needs figures.

H.1 Introduction

The maximum flow and minimum cut problems are examples of a general class of problems called ***linear programming***. Many other optimization problems fall into this class, including minimum spanning trees and shortest paths, as well as several common problems in scheduling, logistics, and economics. Linear programming was used implicitly by Fourier and Jacobi in the early 1800s, but it was first formalized and applied to problems in economics in the 1930s by Leonid Kantorovich. Kantorivich's work was hidden behind the Iron Curtain (where it was largely ignored) and therefore unknown in the West. Linear programming was rediscovered and applied to shipping problems in the late 1930s by Tjalling Koopmans. The first complete algorithm to solve linear programming problems, called the ***simplex method***, was published by George Dantzig in 1947. Koopmans first proposed the name "linear programming" in a discussion with Dantzig in 1948. Kantorovich and Koopmans shared the 1975 Nobel Prize in Economics

© 2018 Jeff Erickson http://algorithms.wtf

"for their contributions to the theory of optimum allocation of resources". Dantzig did not; his work was apparently too pure. Koopmans wrote to Kantorovich suggesting that they refuse the prize in protest of Dantzig's exclusion, but Kantorovich saw the prize as a vindication of his use of mathematics in economics, which his Soviet colleagues had written off as "a means for apologists of capitalism".

A linear programming problem—or more simply, a ***linear program***—asks for a vector $x \in \mathbb{R}^d$ that maximizes or minimizes a given linear function, among all vectors x that satisfy a given set of linear inequalities. The general form of a linear programming problem is the following:

$$\begin{aligned}
\text{maximize} \quad & \sum_{j=1}^{d} c_j x_j \\
\text{subject to} \quad & \sum_{j=1}^{d} a_{ij} x_j \le b_i \quad \text{for each } i = 1 \mathinner{..} p \\
& \sum_{j=1}^{d} a_{ij} x_j = b_i \quad \text{for each } i = p+1 \mathinner{..} p+q \\
& \sum_{j=1}^{d} a_{ij} x_j \ge b_i \quad \text{for each } i = p+q+1 \mathinner{..} n
\end{aligned}$$

Here, the input consists of a *constraint matrix* $A = (a_{ij}) \in \mathbb{R}^{n \times d}$, an *offset vector* $b \in \mathbb{R}^n$, and an *objective vector* $c \in \mathbb{R}^d$. Each coordinate of the vector x is called a *variable*. Each of the linear inequalities is called a *constraint*. The function $x \mapsto c \cdot x$ is called the *objective function*.

A linear program is said to be in ***canonical form***[1] if it has the following structure:

$$\begin{aligned}
\text{maximize} \quad & \sum_{j=1}^{d} c_j x_j \\
\text{subject to} \quad & \sum_{j=1}^{d} a_{ij} x_j \le b_i \quad \text{for each } i = 1 \mathinner{..} n \\
& x_j \ge 0 \quad \text{for each } j = 1 \mathinner{..} d
\end{aligned}$$

A canonical linear program is completely determined by its constraint matrix A, its offset vector b, and its objective vector c. In addition to the n matrix constraints $Ax \ge b$, every canonical linear program includes d *sign constraints* of the form $x_j \ge 0$.

[1]Confusingly, some authors call this *standard form*. Linear programs in this form are also sometimes called *packing* LPs.

We can express this canonical form more compactly as follows. For two vectors $x = (x_1, x_2, \ldots, x_d)$ and $y = (y_1, y_2, \ldots, y_d)$, the expression $x \geq y$ means that $x_i \geq y_i$ for every index i.

$$\boxed{\begin{array}{rr} \max & c \cdot x \\ \text{s.t.} & Ax \leq b \\ & x \geq 0 \end{array}}$$

Any linear programming problem can be converted into canonical form as follows:

- For each variable x_j, add two new variables x_i^+ and x_i^-, an equality constraint $x_j = x_j^+ - x_j^-$, and inequalities $x_j^+ \geq 0$ and $x_j^- \geq 0$.
- Replace any equality constraint $\sum_j a_{ij} x_j = b_i$ with two inequality constraints $\sum_j a_{ij} x_j \geq b_i$ and $\sum_j a_{ij} x_j \leq b_i$.
- Finally, replace any upper bound constraint $\sum_j a_{ij} x_j \geq b_i$ with the equivalent lower bound $\sum_j -a_{ij} x_j \leq -b_i$.

This conversion potentially doubles the number of variables and the number of constraints; fortunately, it is rarely necessary in practice.

Another useful format for linear programs is ***slack form***[2], in which every inequality is of the form $x_j \geq 0$:

$$\boxed{\begin{array}{rr} \max & c \cdot x \\ \text{s.t.} & Ax = b \\ & x \geq 0 \end{array}}$$

It's fairly easy to convert any linear programming problem into slack form; I'll leave the details as an easy exercise (hint, hint). Slack form is especially useful in executing the simplex algorithm, which we'll see in the next chapter.

H.2 The Geometry of Linear Programming

A point $x \in \mathbb{R}^d$ is ***feasible*** with respect to some linear programming problem if it satisfies all the linear constraints. The set of all feasible points is called the *feasible region* for that linear program. The feasible region has a particularly nice geometric structure that lends some useful intuition to the linear programming algorithms we'll see later.

Any linear equation in d variables defines a *hyperplane* in $\mathbb{R}^d$; think of a line when $d = 2$, or a plane when $d = 3$. This hyperplane divides $\mathbb{R}^d$ into two *halfspaces*; each halfspace is the set of points that satisfy some linear inequality. Thus, the set of feasible points is the intersection of several hyperplanes (one for each equality constraint) and halfspaces (one for each inequality constraint). The intersection of a finite number of hyperplanes and halfspaces is called a ***polyhedron***. It's not hard to verify that any

[2]Confusingly, some authors call this *standard form*.

halfspace, and therefore any polyhedron, is *convex*—if a polyhedron contains two points x and y, then it contains the entire line segment $\overline{xy}$.

Figure H.1. A two-dimensional polyhedron defined by 11 linear inequalities.

By rotating $\mathbb{R}^d$ (or choosing an appropriate coordinate frame) so that the objective function points downward, we can express *any* linear programming problem in the following geometric form:

$$\boxed{\text{Find the lowest point in a given polyhedron.}}$$

With this geometry in hand, we can easily picture two pathological cases where a given linear programming problem has no solution. The first possibility is that there are no feasible points; in this case the problem is called ***infeasible***.

$$\boxed{\begin{array}{ll} \text{maximize} & x - y \\ \text{subject to} & 2x + y \leq 1 \\ & x + y \geq 2 \\ & x, y \geq 0 \end{array}}$$

Figure H.2. An infeasible linear programming problem; arrows indicate the constraints.

The second possibility is that there are feasible points at which the objective function is arbitrarily large; in this case, we call the problem ***unbounded***. The same polyhedron could be unbounded for some objective functions but not others, or it could be unbounded for every objective function.

Figure H.3. A two-dimensional polyhedron (white) that is unbounded downward but bounded upward.

H.3 Examples

One Shortest Path

We can compute the length of the shortest path from s to t in a weighted directed graph by solving the following simple linear program.

$$\begin{array}{lll} \text{maximize} & dist(t) & \\ \text{subject to} & dist(s) = 0 & \\ & dist(v) - dist(u) \leq \ell(u{\to}v) & \text{for every edge } u{\to}v \end{array}$$

Here the input consists of a graph $G = (V, E)$ and a vector $\ell \in \mathbb{R}^E$ (or equivalently, a function $\ell : E \to \mathbb{R}$) representing the lengths of edges in G, and our goal is to compute a vector $dist \in \mathbb{R}^V$ (or equivalently, a function $dist : V \to \mathbb{R}$) such that $dist(t)$ is the shortest path distance from s to t. The constraints in the linear program encode Ford's generic requirement that every edge in the graph must be relaxed. These relaxation constraints imply that in any feasible solution, $dist(v)$ is *at most* the shortest path distance from s to v, for *every* vertex v. Thus, somewhat counterintuitively, we are correctly *maximizing* $dist(t)$ to compute the *shortest*-path distance from s to t!

- This linear program is feasible if and only if the graph has no negative cycles. On the one hand, if the graph has no negative cycles, then the true shortest-path distances give a feasible solution. On the other hand, if the graph has a negative cycle, then for any distance values $dist(v)$, at least one edge in that cycle will be tense.
- This linear program is bounded if and only if the graph contains a path from s to t. On the one hand, if there is a path from s to t, the length of that path is an upper bound on the optimal objective value. On the other hand, if there is no path from s to t, we can obtain a feasible solution by setting $dist(v) = 0$ for all vertices v that are reachable from s, and $dist(v) = \infty$ for all vertices v that are not reachable from s.
- The optimal solution to this linear program is not necessarily unique. For example, if the graph contains a vertex v that cannot reach t, we can transform any optimal solution into another optimal solution by arbitrarily decreasing $dist(v)$. More

generally, this LP has a unique optimal solution if and only if (1) every vertex lies on a path from s to t and (2) every path from s to t has the same length.

One Shortest Path, Take Two

Alternatively, and perhaps more intuitively, we can formulate the shortest-path problem as a minimization problem. However, instead of using variables that represent distances, this linear program uses variables $x(u{\to}v)$ that *intuitively* indicate which edges $u{\to}v$ lie on the shortest path from s to t.

$$\begin{aligned}
\text{minimize} \quad & \sum_{u\to v} \ell(u{\to}v)\cdot x(u{\to}v) \\
\text{subject to} \quad & \sum_{u} x(u{\to}t) - \sum_{w} x(t{\to}w) = 1 \\
& \sum_{u} x(u{\to}v) - \sum_{w} x(v{\to}w) = 0 \quad \text{for every vertex } v \neq s, t \\
& x(u{\to}v) \geq 0 \quad \text{for every edge } u{\to}v
\end{aligned}$$

Any path from s to t—in particular, the *shortest* from s to t path—yields a feasible point x for this linear program, where $x(u{\to}v) = 1$ for each edge $u{\to}v$ in the path and $x(u{\to}v) = 0$ for every other edge. More generally, any *walk* from s to t implies a feasible solution, where $x(u{\to}v)$ is the number of times $u{\to}v$ appears in the walk.

Sharp-eyed readers might recognize this linear program as a *minimum-cost flow* problem, where each edge $u{\to}v$ has cost $\ell(u{\to}v)$ and infinite capacity, the source vertex s has balance -1 (one unit of supply), the target vertex t has balance 1 (one unit of demand), and all other vertices have balance 0. (The missing balance constraint for s is implied by the other $V-1$ balance constraints.) More concisely, the feasible points for this linear program are flows from s to t with value 1, and the objective function $\ell \cdot x$ is the *cost* of this flow.

Because all balances and (nonexistent) capacities in this network are integers, we know that there is an integral minimum-cost flow from s to t. (The existence of an integral optimal solution a feature of this particular linear program! It is possible for a linear program with only integral input data A, b, and c to have only non-integral optimal solutions.) A simple flow decomposition argument now implies that the minimum-cost flow from s to t is actually a simple path from s to t.

- This linear program is feasible if and only if the underlying graph contains a path from s to t. If G contains a path from s to t, then sending one unit of flow along that path gives us a feasible solution. On the other hand, any (s,t)-flow is a weighted sum of (s,t)-paths and possibly cycles.
- This linear program is bounded if and only if the underlying graph has no negative cycles. If the G contains a negative cycle, we can arbitrarily decrease the cost of any solution by pushing more flow around that cycle. On the other hand, suppose G has

no negative cycles, and let x be any feasible solution. If x contains any cycles, we can decrease the cost of x by removing them. Then the maxflow-mincut theorem implies that $x(u{\to}v) \leq 1$ for every edge $u{\to}v$. We conclude that the cost of any feasible solution is at least the sum of all negative edge lengths in G.

- The optimal solution to this linear program is not necessarily unique, because shortest paths are not necessarily unique. Moreover, if there is more than one optimal solution, then there are infinitely many optimal solutions. Specifically, if x and x' are optimal solutions, then for any constant $0 < \alpha < 1$, the convex combination $\alpha x + (1-\alpha)x'$ is also an optimal solution.

Single-Source Shortest Paths

In the optimal solution for the first shortest-path linear program, the objective function $dist(t)$ is the actual shortest-path distance from s to t, but for any vertex v that is not on the shortest path from s to t, the corresponding value $dist(v)$ may underestimate the true distance from s to v. We can obtain the true distances from s to *every* other vertex by modifying the objective function as follows:

$$\begin{aligned}
\text{maximize} \quad & \sum_v dist(v) \\
\text{subject to} \quad & dist(s) = 0 \\
& dist(v) - dist(u) \leq \ell(u{\to}v) \quad \text{for every edge } u{\to}v
\end{aligned}$$

Again, we are counterintuitively faced with a *maximization* problem to find *shortest* paths.

We can similarly modify the second shortest-path linear program to compute a spanning tree of shortest paths.

$$\begin{aligned}
\text{minimize} \quad & \sum_{u\to v} \ell(u{\to}v) \cdot x(u{\to}v) \\
\text{subject to} \quad & \sum_u x(u{\to}v) - \sum_w x(v{\to}w) = 1 \quad \text{for every vertex } v \neq s \\
& x(u{\to}v) \geq 0 \quad \text{for every edge } u{\to}v
\end{aligned}$$

Let T be any shortest-path tree rooted at s, and for any vertex v, let D_v denote set of all descendants of v in T. (In particular, $D_s = V$, and $D_v = |v|$ if v is a leaf of T.) Then we can obtain an optimal solution to this second LP by setting

$$x(u{\to}v) = \begin{cases} |D_v| & \text{if } u{\to}v \in T \\ 0 & \text{otherwise} \end{cases}$$

Again, sharp-eyed readers may recognize this linear program as an uncapacitated minimum-cost flow problem.

Maximum Flows and Minimum Cuts

Recall that the input to the maximum (s,t)-flow problem consists of a directed graph $G=(V,E)$, two special vertices s and t, and a function assigning a non-negative capacity $c(e)$ to each edge e. Our task is to compute a flow function $f \in E^{\mathbb{R}}$ subject to the usual capacity and balance constraints. The maximum-flow problem can be encoded as a linear program as follows:

$$\begin{aligned}
\text{maximize} \quad & \sum_{w} f(s{\to}w) - \sum_{u} f(u{\to}s) \\
\text{subject to} \quad & \sum_{w} f(v{\to}w) - \sum_{u} f(u{\to}v) = 0 && \text{for every vertex } v \neq s,t \\
& f(u{\to}v) \leq c(u{\to}v) && \text{for every edge } u{\to}v \\
& f(u{\to}v) \geq 0 && \text{for every edge } u{\to}v
\end{aligned}$$

Similarly, the minimum cut problem can be formulated using "indicator" variables similarly to the shortest-path problem. We define a variable $S(v)$ for each vertex v, indicating whether $v \in S$ or $v \in T$, and a variable $x(u{\to}v)$ for each edge $u{\to}v$, indicating whether $u \in S$ and $v \in T$, where (S,T) is some (s,t)-cut.

$$\begin{aligned}
\text{minimize} \quad & \sum_{u\to v} c(u{\to}v) \cdot x(u{\to}v) \\
\text{subject to} \quad & x(u{\to}v) + S(v) - S(u) \geq 0 && \text{for every edge } u{\to}v \\
& x(u{\to}v) \geq 0 && \text{for every edge } u{\to}v \\
& S(s) = 1 \\
& S(t) = 0
\end{aligned}$$

Like the minimization linear programs for shortest paths, there can be optimal solutions that assign fractional values to the variables. Nevertheless, the minimum value for the objective function is the cost of the minimum cut, and there is an optimal solution for which every variable is either 0 or 1, representing an actual minimum cut.

H.4 Linear Programming Duality

Each of the preceding pairs of linear programs is an example of a general relationship called *duality*. For any linear programming problem, there is a corresponding *dual* linear program that can be obtained by a mechanical translation, essentially by swapping the constraints and the variables. The translation is simplest when the original LP is in canonical form:

Primal (Π)		**Dual (Ц)**
$\begin{aligned} \max \quad & c \cdot x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$	$\Longleftrightarrow$	$\begin{aligned} \min \quad & y \cdot b \\ \text{s.t.} \quad & yA \geq c \\ & y \geq 0 \end{aligned}$

We can also write the dual linear program in exactly the same canonical form as the primal, by swapping the coefficient vector c and the objective vector b, negating both vectors, and replacing the constraint matrix A with its negative transpose.[3]

Primal (Π)		**Dual (Π)**
$\begin{array}{rr} \max & c \cdot x \\ \text{s.t.} & Ax \le b \\ & x \ge 0 \end{array}$	$\Longleftrightarrow$	$\begin{array}{rr} \max & -b^\top \cdot y^\top \\ \text{s.t.} & -A^\top y^\top \le -c^\top \\ & y^\top \ge 0 \end{array}$

Written in this form, it should be immediately clear that duality is an *involution*: The dual of the dual linear program Π is identical to the primal linear program Π. The choice of which linear program to call the "primal" and which to call the "dual" is totally arbitrary.[4]

Dualizing Arbitrary LPs

Given a linear program Π that is not in canonical form, we can mechanically derive its dual linear program by first converting Π into canonical form and then applying the pattern shown above. If we push the symbols around long enough, we eventually find the following general pattern for constructing the dual of any ***maximization*** linear program. (The rules for dualizing minimization programs are the reverse of these.)

- For each constraint in the primal LP *except* sign constraints of the form $x_i \gtrless 0$, there is a corresponding variable in the dual LP.
  - If the primal constraint is an upper bound, the dual variable must be non-negative.
  - If the primal constraint is a lower bound, the dual variable must be non-positive.
  - If the primal constraint is an equation, the sign of dual variable is not constrained.
  - The right side of the primal constraint becomes the dual variable's coefficient in the objective function.
  - The dual linear program *minimizes* its objective value.
- Conversely, for each variable in the primal LP, there is a corresponding constraint in the dual LP.
  - If the primal variable is non-negative, the dual constraint is a lower bound.
  - If the primal variable is non-positive, the dual constraint is an upper bound.
  - If the sign of the primal variable is not constrained, the dual constraint is an equation.

[3]For the notational purists: In these formulations, x and b are column vectors, and y and c are row vectors. This is a somewhat nonstandard choice. Yes, that means the dot in $c \cdot x$ is redundant. Sue me.

[4]For historical reasons, maximization linear programs tend to be called "primal" and minimization linear programs tend to be called "dual". (Or is it the other way around?) This habit is nothing but a pointless and distracting religious tradition.

- Finally, if the primal variable is equal to zero, the dual constraint doesn't actually exist. (If something can't vary, it's not really a variable!)
- The primal variable's coefficient in the objective function becomes the right side of the dual constraint.

Primal	Dual	Primal	Dual
$\max c \cdot x$	$\min y \cdot b$	$\min c \cdot x$	$\max y \cdot b$
$\sum_j a_{ij} x_j \le b_i$	$y_i \ge 0$	$\sum_j a_{ij} x_j \le b_i$	$y_i \le 0$
$\sum_j a_{ij} x_j \ge b_i$	$y_i \le 0$	$\sum_j a_{ij} x_j \ge b_i$	$y_i \ge 0$
$\sum_j a_{ij} x_j = b_i$	$-$	$\sum_j a_{ij} x_j = b_i$	$-$
$x_j \ge 0$	$\sum_i y_i a_{ij} \ge c_j$	$x_j \le 0$	$\sum_i y_i a_{ij} \ge c_j$
$x_j \le 0$	$\sum_i y_i a_{ij} \le c_j$	$x_j \ge 0$	$\sum_i y_i a_{ij} \le c_j$
$-$	$\sum_i y_i a_{ij} = c_j$	$-$	$\sum_i y_i a_{ij} = c_j$
$x_j = 0$	$-$	$x_j = 0$	$-$

Figure H.4. Constructing the dual of an arbitrary linear program.

At first glance, these rules appear to be asymmetric: The inequality directions for primal constraints and dual variables are opposites, but the inequality directions for primal variables and dual constraints are identical. In fact, this asymmetry reflects the switch from a primal *maximization* LP to a dual *minimization* LP. The dual of the dual of any linear program is always (equivalent to) the original linear program.

Shortest Path Example

As an example, let's derive the dual of our very first linear program for shortest paths:

$$\begin{aligned} \text{maximize} \quad & dist(t) \\ \text{subject to} \quad & dist(s) = 0 \\ & dist(v) - dist(u) \le \ell(u \to v) \quad \text{for every edge } u \to v \end{aligned}$$

This linear program has a variable $dist(v)$ for every vertex v and a constraint for every edge. Thus, the underlying constraint matrix A has a row (constraint) for every edge and a column (variable) for every vertex. Specifically, we have

$$A(u \to v, w) = \begin{cases} -1 & \text{if } u = w, \\ +1 & \text{if } v = w, \\ 0 & \text{otherwise.} \end{cases}$$

(A is the *signed incidence matrix* of the underlying directed graph G.) The coefficients of the objective vector are $+1$ for t and 0 for every other vertex, and the coefficients of the offset vector are the edge lengths.

So, the dual linear program has a constraint for every vertex and a variable for every edge. Let's call the edge variables $x(u{\to}v)$, because why not? Because $dist(s) = 0$, there is no dual constraint corresponding to $dist(s)$ after all. Because the other primal variables $dist(v)$ are not sign-constrained, the dual constraints are all equations. The primal constraints are upper bounds, so the dual variables $x(u{\to}v)$ must be non-negative. Finally, the coefficients of the dual objective vector are the edge lengths, and the coefficients of the dual offset vector are +1 for t and 0 for every other vertex.

$$\begin{aligned}
\text{minimize} \quad & \sum_{u\to v} \ell(u{\to}v)\cdot x(u{\to}v) \\
\text{subject to} \quad & \sum_{u} x(u{\to}t) - \sum_{w} x(t{\to}w) = 1 \\
& \sum_{u} x(u{\to}v) - \sum_{w} x(v{\to}w) = 0 \quad \text{for every vertex } v \neq s, t \\
& x(u{\to}v) \geq 0 \quad \text{for every edge } u{\to}v
\end{aligned}$$

Hey, this looks familiar!

H.5 The Fundamental Theorem

The most important structural observation about linear programming is the following broad generalization of the maxflow-mincut theorem.

The Fundamental Theorem of Linear Programming. *A canonical linear program* Π *has an optimal solution* x^* *if and only if the dual linear program* $\amalg$ *has an optimal solution* y^* *such that* $c \cdot x^* = y^* A x^* = y^* \cdot b$.

The Fundamental Theorem (in a slightly different form) was first conjectured by John von Neumann in May 1947, during a fateful meeting with George Dantzig. Dantzig met with von Neumann to explain his general theory of linear programming and his simplex algorithm (which we'll see in the next chapter). According to Dantzig, after about half an hour into his presentation, von Neumann told him "Get to the point!" When Dantzig then explained his algorithm more concisely, von Neumann replied dismissively "Oh, *that!*" and proceeded to lecture Dantzig for over an hour on convexity, fixed points, and his theory of two-player games, while Dantzig sat (as he later put it) dumbfounded. During his lecture, von Neumann defined the negative-transpose dual of an arbitrary linear program—based on a similar duality in two-player zero-sum games that trivially results from swapping the players—and conjectured correctly that the Fundamental Theorem could be derived from his famous min-max theorem, which he proved in 1928:

Min-Max Theorem. *For any matrix* A*, we have* $\min_x \max_y xAy = \max_y \min_x xAy$.

Von Neumann's min-max theorem is itself a consequence of several equivalent "Theorems of the Alternative" proved by Paul Gordan, Hermann Minkowski, Gyula Farkas, Erich Stiemke, Theodore Motzkin, and others in the late 1800s and early 1900s. The earliest example of such a theorem (that I know of) was proved by Gordan in 1873.

Gordan's Theorem of the Alternative. *For any matrix A, either $Ax = 0$ and $x \geq 0$ for some non-zero vector x, or $yA > 0$ for some vector y, but not both.*

Soon after their 1947 meeting, von Neumann and Dantzig independently wrote up proofs of the Fundamental Theorem, which they each distributed privately for many years. Unfortunately, von Neumann's proof was flawed, although his errors were finally corrected when he finally published the proof in 1963, and Dantzig never published his proof. The first *published* proof was written by Albert Tucker, David Gale, and Harold Kuhn in 1951.

I will defer a proof of the full Fundamental Theorem until after we develop some more intuition, but the following weak form is trivial to prove.

Weak Duality Theorem. *If x is a feasible solution for a canonical linear program Π, and y is a feasible solution for its dual Π, then $c \cdot x \leq yAx \leq y \cdot b$.*

Proof: Because x is feasible for Π, we have $Ax \leq b$. Since y is non-negative, we can multiply both sides of the inequality to obtain $yAx \leq y \cdot b$. Conversely, y is feasible for Π and x is non-negative, so $yAx \geq c \cdot x$. □

The Weak Duality Theorem has two important immediate consequences:

- Let x and y be arbitrary *feasible* solutions to a canonical linear program and its dual, respectively. If $c \cdot x = y \cdot b$, then x and y are *optimal* primal and dual solutions, respectively, and $c \cdot x = yAx = y \cdot b$.
- If a linear program is unbounded, then its dual is infeasible; equivalently, if a linear program is feasible, then its dual is bounded. (It is possible for a linear program and its dual to both be infeasible, but only if both linear programs are degenerate.)

The full version of the Fundamental Theorem implies that that all these implications are actually equivalences.

H.6 Another Duality Example

Before I prove the stronger duality theorem, let me try to provide some intuition about how this duality thing actually works.[5] Consider the following canonical linear

[5]This example is taken from Robert Vanderbei's excellent textbook *Linear Programming: Foundations and Extensions* [Springer, 2001], but the idea appears earlier in Jens Clausen's 1997 paper "Teaching Duality in Linear Programming: The Multiplier Approach".

programming problem:

$$\begin{array}{ll} \text{maximize} & 4x_1 + x_2 + 3x_3 \\ \text{subject to} & x_1 + 4x_2 \le 2 \\ & 3x_1 - x_2 + x_3 \le 4 \\ & x_1, x_2, x_3 \ge 0 \end{array}$$

Let σ^* denote the optimum objective value for this LP. The feasible solution $x = (1, 0, 0)$ gives us a lower bound $\sigma^* \ge 4$. A different feasible solution $x = (0, 0, 3)$ gives us a better lower bound $\sigma^* \ge 9$. We could play this game all day, finding different feasible solutions and getting ever larger lower bounds. How do we know when we're done? Is there a way to prove an *upper* bound on σ^*?

In fact, there is. Let's multiply each of the constraints in our LP by a new non-negative scalar value y_i:

$$\begin{array}{ll} \text{maximize} & 4x_1 + x_2 + 3x_3 \\ \text{subject to} & y_1(x_1 + 4x_2) \le 2y_1 \\ & y_2(3x_1 - x_2 + x_3) \le 4y_2 \\ & x_1, x_2, x_3 \ge 0 \end{array}$$

Because each y_i is non-negative, we do not reverse any of the inequalities. Any feasible solution (x_1, x_2, x_3) must satisfy both of these inequalities, so it must also satisfy their sum:

$$(y_1 + 3y_2)x_1 + (4y_1 - y_2)x_2 + y_2x_3 \le 2y_1 + 4y_2.$$

Now suppose that the coefficient of each variable x_i in the expression on the left side of this inequality is larger than the corresponding coefficient of the objective function:

$$y_1 + 3y_2 \ge 4, \qquad 4y_1 - y_2 \ge 1, \qquad y_2 \ge 3.$$

This assumption implies an upper bound on the objective value of *any* feasible solution:

$$4x_1 + x_2 + 3x_3 \;\le\; (y_1 + 3y_2)x_1 + (4y_1 - y_2)x_2 + y_2x_3 \;\le\; 2y_1 + 4y_2. \qquad (*)$$

In particular, by plugging in the optimal solution (x_1^*, x_2^*, x_3^*) for the original LP, we obtain the following upper bound on σ^*:

$$\sigma^* \;=\; 4x_1^* + x_2^* + 3x_3^* \;\le\; 2y_1 + 4y_2.$$

Now it is natural to ask how tight this upper bound can be. That is, how small can we make the expression $2y_1 + 4y_2$ without violating any of the inequalities we used to

prove the upper bound? This is just another linear programming problem.

$$\begin{aligned} \text{minimize} \quad & 2y_1 + 4y_2 \\ \text{subject to} \quad & y_1 + 3y_2 \geq 4 \\ & 4y_1 - \ \ y_2 \geq 1 \\ & y_2 \geq 3 \\ & y_1, y_2 \geq 0 \end{aligned}$$

In fact, the resulting linear program is precisely the dual of our original linear program! Moreover, inequality $(*)$ is just an instantiation of the Weak Duality Theorem.

H.7 Strong Duality

The Fundamental Theorem can be rephrased in the following form:

Strong Duality Theorem. *If x^* is an optimal solution for a canonical linear program Π, then there is an optimal solution y^* for its dual $\amalg$, such that $c \cdot x^* = y^* A x^* = y^* \cdot b$.*

Proof (sketch): I'll prove the theorem only for ***non-degenerate*** linear programs, in which (a) the objective vector is not normal to any constraint hyperplane, and (b) at most d constraint hyperplanes pass through any point. The first assumption implies that the optimal solution to the LP—if it exists—is unique and is therefore a vertex of the feasible region. These non-degeneracy assumptions are relatively easy to enforce in practice and can be removed from the proof at the expense of some technical detail. I will also assume that $n \geq d$; the argument for under-constrained LPs is similar (if not simpler).

To develop some intuition, let's first consider the *very* special case where $x^* = (0, 0, \ldots, 0)$. Let e_i denote the ith standard basis vector, whose ith coordinate is 1 and all other coordinates are 0. Because $x_i^* = 0$ for all i, our non-degeneracy assumption implies the *strict* inequality $a_i \cdot x^* < b_i$ for all i. Thus, any sufficiently small ball around the origin does not intersect any other constraint hyperplane $a_i \cdot x = b_i$. Thus, for all i, and for any sufficiently small $\delta > 0$, the vector δe_i is feasible. Because x^* is the unique optimum, we must have $\delta c_i = c \cdot (\delta e_i) < c \cdot x^* = 0$. We conclude that $c_i < 0$ for all i.

Now let $y = (0, 0, \ldots, 0)$ as well. We immediately observe that $yA \geq c$ and $y \geq 0$; in other words, y is a *feasible* solution for the dual linear program $\amalg$. But $y \cdot b = 0 = c \cdot x^*$, so the weak duality theorem implies that y is an *optimal* solution to $\amalg$, and the proof is complete for this very special case!

Now let's consider the more general case. Let x^* be the optimal solution for the linear program Π; our non-degeneracy assumption implies that this solution is unique, and that it satisfies exactly d of the n linear constraints with equality. Without loss of

generality (by permuting the constraints and possibly changing coordinates), we can assume that these are the first d constraints. Thus, we have

$$\begin{aligned} a_i \cdot x^* &= b_i \qquad \text{for all } i \le d, \\ a_i \cdot x^* &< b_i \qquad \text{for all } i \ge d+1, \end{aligned}$$

where a_i denotes the ith row of A. Let $A_\bullet$ denote the $d \times d$ matrix containing the first d rows of A. Our non-degeneracy assumption implies that $A_\bullet$ has full rank, and thus has a well-defined inverse $V = A_\bullet^{-1}$.

Now define a vector $y \in \mathbb{R}^n$ by setting

$$\begin{aligned} y_j &:= c \cdot v^j \qquad \text{for all } j \le d, \\ y_j &:= 0 \qquad \text{for all } j \ge d+1, \end{aligned}$$

where v^j denotes the jth column of V. The definition of matrix inverse implies that $a_i \cdot v^i = 1$ for all i, and $a_i \cdot v^j = 0$ for all $i \neq j$.

To simplify notation, let $y_\bullet = (y_1, y_2, \ldots, y_d)$ and let $b_\bullet = (b_1, b_2, \ldots, b_d) = A_\bullet x^*$. Because $y_i = 0$ for all $i \ge d+1$, we immediately have

$$y \cdot b = y_\bullet \cdot b_\bullet = cVb_\bullet = cA_\bullet^{-1} b_\bullet = c \cdot x^*$$

and

$$yA = y_\bullet A_\bullet = cVA_\bullet = cA_\bullet^{-1} A_\bullet = c.$$

The point x^* lies on exactly d constraint hyperplanes; moreover, any sufficiently small ball around x^* intersects *only* those d constraint hyperplanes. Consider the point $\tilde{x} = x^* - \varepsilon v^j$, for some index $1 \le j \le d$ and some sufficiently small $\varepsilon > 0$. We have $a_i \cdot \tilde{x} = a_i \cdot x^* - \varepsilon(a_i \cdot v^j) = b_i$ for all $i \neq j$, and $a_j \cdot \tilde{x} = a_j \cdot x^* - \varepsilon(a_j \cdot v^j) = b_j - \varepsilon < b_j$. Thus, $\tilde{x}$ is a feasible point for Π. Because x^* is the *unique* optimum for Π, we must have $c \cdot \tilde{x} = c \cdot x^* - \varepsilon(c \cdot v^j) < c \cdot x^*$. We conclude that $y_j = c \cdot v^j > 0$ for all j.

We have shown that $yA \ge c$ and $y \ge 0$, so y is a *feasible* solution for the dual linear program Π. We have also shown that $y \cdot b = c \cdot x^*$, so the Weak Duality Theorem implies that y is an *optimal* solution for Π, and the proof is complete! □

We can also give a useful geometric interpretation to the dual vector $y_\bullet \in \mathbb{R}^d$. Each linear equation $a_i \cdot x = b_i$ defines a hyperplane in $\mathbb{R}^d$ with normal vector a_i. The normal vectors $a_1, \ldots, a_d$ are linearly independent (by non-degeneracy) and therefore describe a coordinate frame for the vector space $\mathbb{R}^d$. The definition of $y_\bullet$ implies that $c = y_\bullet A_\bullet = \sum_{i=1}^d y_i a_i$. In other words, ***the non-zero dual variables $y_1, \ldots, y_d$ are the coefficients of the objective vector c in the coordinate frame $a_1, \ldots, a_d$.***

In more physical terms, imagine that the objective vector c points downward, and that x^* is the lowest point in the feasible region, which is found by tossing a marble into the feasible region and letting it fall as far as possible without crossing any constraint

hyperplane. Even after the marble reaches the lowest feasible point, gravity is still pulling it downward, but the marble isn't moving, so the constraint hyperplanes must be pushing it upward. Each constraint hyperplane can only push in the direction of its normal vector. ***Each dual variable y_i is the amount of force applied by the ith constraint hyperplane.***

H.8 Complementary Slackness

The Strong Duality Theorem implies a relationship between the optimal solutions of a linear program and its dual that is stronger than just the equality of their objective values. In physical terms, a constraint hyperplane applies a non-zero force to the marble only if it touches the marble; that is, if a dual variable is positive, then the corresponding constraint must be satisfied with equality.

Recall that the theorem states that if x^* is the optimal solution to a canonical linear program Π, then the dual program $\amalg$ has an optimal solution y^* such that $c \cdot x^* = y^* A x^* = y^* \cdot b$. These equations have two immediate consequences for any optimal solution vectors x^* and y^*.

- For any row index i, either $y_i^* = 0$ or $a_i \cdot x^* = b$ (or both).
- For any column index j, either $x_j^* = 0$ or $y^* \cdot a^j = c$ (or both).

Here, a_i and a^j respectively denote the ith row and jth column of A. These consequences of strong duality are called the ***complementary slackness*** conditions.

Call a constraint *tight* if it is satisfied with equality and *loose* otherwise. The complementary slackness conditions can be phrased more colloquially as follows:

- If a primal variable is positive, the corresponding dual constraint is tight.
- If a dual constraint is loose, the corresponding primal variable is zero.
- If a dual variable is positive, the corresponding primal constraint is tight.
- If a primal constraint is loose, the corresponding dual variable is zero.

Complementary Slackness Theorem. *Let x be an arbitrary feasible solution to a canonical linear program Π, and let y be an arbitrary feasible solution to the dual linear program $\amalg$. The following conditions are equivalent:*

- *x and y are optimal solutions to their respective linear programs.*
- $c \cdot x = y \cdot b$
- $c \cdot x = yAx$
- $yAx = y \cdot b$
- *x and y satisfy all complementary slackness conditions.*

The complementary slackness conditions are not necessarily exclusive; it is possible for a constraint to be tight *and* for the corresponding dual variable to be zero. Nevertheless, every bounded and feasible linear program has optimal primal and dual solutions x^* and y^* that satisfy *strict* complementary slackness conditions:

- For any row index i, either $y_i^* > 0$ or $a_i \cdot x^* < b$ (but not both).
- For any column index j, either $x_j^* > 0$ or $y^* \cdot a^j > c$ (but not both).

However, if the linear program is non-degenerate, the primal and dual optimal solutions are both unique and therefore satisfy these stricter conditions.

Exercises

Need more!

★★★

1. A matrix $A = (a_{ij})$ is *skew-symmetric* if and only if $a_{ji} = -a_{ij}$ for all indices $i \neq j$; in particular, every skew-symmetric matrix is square. A canonical linear program $\max\{c \cdot x \mid Ax \leq b; x \geq 0\}$ is *self-dual* if the matrix A is skew-symmetric and the objective vector c is *equal* to the constraint vector b.
   (a) Prove that any self-dual linear program Π is syntactically equivalent to its dual program Π.
   (b) Show that any linear program Π with d variables and n constraints can be transformed into a self-dual linear program with $n + d$ variables and $n + d$ constraints. The optimal solution to the self-dual program should include both the optimal solution for Π (in d of the variables) and the optimal solution for the dual program Π (in the other n variables).

2. (a) Give a linear-programming formulation of the ***bipartite maximum matching*** problem. The input is a bipartite graph $G = (U \cup V; E)$, where $E \subseteq U \times V$; the output is the largest matching in G. Your linear program should have one variable for each edge.
   (b) Now dualize the linear program from part (a). What do the dual variables represent? What does the objective function represent? What problem is this!?

3. (a) Give a linear-programming formulation of the ***minimum-cost circulation*** problem. You are given a flow network whose edges have both capacities and costs, and your goal is to find a feasible circulation (flow with value 0) whose total cost is as small as possible.
   (b) Derive the dual of your linear program from part (a).

4. Suppose we are given a sequence of n linear inequalities of the form $a_i x + b_i y \leq c_i$. Collectively, these n inequalities describe a convex polygon P in the plane.
   (a) Describe a linear program whose solution describes the largest axis-aligned square that lies entirely inside P.
   (b) Describe a linear program whose solution describes the maximum-perimeter axis-aligned rectangle that lies entirely inside P.

(c) Describe a linear program whose solution describes the largest circle that lies entirely inside P.

(d) Describe a polynomial-time algorithm to compute two interior-disjoint axis-aligned squares with maximum total perimeter that lie entirely inside P. *[Hint: There are exactly two interesting cases to consider; for each case, formulate a corresponding linear program.]*

(e) Describe a polynomial-time algorithm to compute two interior-disjoint axis-aligned rectangles with maximum total perimeter that lie entirely inside P. *[Hint: Again, there are only two interesting cases to consider.]*

In all subproblems, "axis-aligned" means that the edges of the square(s) or rectangles(s) are horizontal and vertical.

5. Given points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ in the plane, the ***linear regression problem*** asks for real numbers a and b such that the line $y = ax + b$ fits the points as closely as possible, according to some criterion. The most common fit criterion is minimizing the $\boldsymbol{L_2}$ ***error***, defined as follows:[6]

$$\varepsilon_2(a, b) = \sum_{i=1}^{n} (y_i - ax_i - b)^2.$$

But there are many other ways of measuring a line's fit to a set of points, some of which can be optimized via linear programming.

(a) The $\boldsymbol{L_1}$ ***error*** (or *total absolute deviation*) of the line $y = ax + b$ is defined as follows:

$$\varepsilon_1(a, b) = \sum_{i=1}^{n} \left| y_i - ax_i - b \right|.$$

Describe a linear program whose solution (a, b) describes the line with minimum L_1 error.

[6] This measure is also known as *sum of squared residuals*, and the algorithm to compute the best fit is normally called *(ordinary/linear) least squares*.

(b) The ***L_∞ error*** (or *maximum absolute deviation*) of the line $y = ax + b$ is defined as follows:

$$\varepsilon_\infty(a, b) = \max_{i=1}^{n} \left| y_i - ax_i - b \right|.$$

Describe a linear program whose solution (a, b) describes the line with minimum L_∞ error.

6. Let $G = (V, E)$ be an arbitrary directed graph with weighted vertices; vertex weights may be positive, negative, or zero. A ***prefix*** of G is a subset $P \subseteq V$, such that there is no edge $u \to v$ where $u \notin P$ but $v \in P$. A ***suffix*** of G is the complement of a prefix. Finally, an ***interval*** of G is the intersection of a prefix of G and a suffix of G. The weight of a prefix, suffix, or interval is the sum of the weights of its vertices.

   (a) Describe a linear program whose solution describes the maximum-weight prefix of G. Your linear program should have one variable per vertex, indicating whether that vertex is or is not in the chosen prefix.

   (b) Describe an algorithm that computes the maximum-weight prefix of G, by reducing to a standard maximum-flow problem in a related graph. *[Hint: Consider the special case where G is a dag. Haven't we seen this problem before?]*

   (c) Describe an efficient algorithm that computes the maximum-weight prefix of G, when G is a rooted tree with all edges pointing away from the root. *[Hint: This is really easy if you* **don't** *think about linear programming.]*

   (d) Describe a linear program whose solution describes the maximum-weight interval of G.

   (e) Describe an efficient algorithm that computes the maximum-weight interval of G, when G is a rooted tree with all edges pointing away from the root. *[Hint: Again, this is easy if you* **don't** *think about linear programming.]*

   [Hint: Don't worry about the solutions to your linear programs being integral; they will be (essentially by part (b)). If all vertex weights are negative, the maximum-weight interval is empty; if all vertex weights are positive, the maximum-weight interval contains every vertex.]

7. Suppose you are given an arbitrary directed graph $G = (V, E)$ with arbitrary edge weights $\ell : E \to \mathbb{R}$, and two special vertices. Each edge in G is colored either red, white, or blue to indicate how you are permitted to modify its weight:

   - You may increase, but not decrease, the length of any red edge.
   - You may decrease, but not increase, the length of any blue edge.
   - You may not change the length of any black edge.

   Your task is to modify the edge weights—subject to the color constraints—so that *every* path from s to t has exactly the same length. Both the given weights and the

new weights of the edges can be positive, negative, or zero. To keep the following problems simple, assume every edge in G lies on at least one path from s to t, and that G has no isolated vertices.

(a) Describe a linear program that is feasible if and only if it is possible to make every path from s to t the same length. *[Hint: Let* $\text{dist}(v)$ *denote the length of every path from* s *to* v*.]*

(b) Construct the dual of the linear program from part (a). *[Hint: Choose a convenient objective function for your primal LP.]*

♥(c) Give a self-contained description of the combinatorial problem encoded by the dual linear program from part (b), and prove *directly* that it is equivalent to the original path-equalization problem. Do not use the words "linear", "program", or "dual". Yes, you've seen this problem before. *[Hint: The proof is the hard part.]*

(d) Describe and analyze an algorithm to determine *in* $O(EV)$ *time* whether it is possible to make every path from s to t the same length. Do not use the words "linear", "program", or "dual".

(e) Finally, suppose we want to equalize path lengths while modifying the edge weights as little as possible. Specifically, we want to compute new edge weights $\ell'(e)$ that give every path from s to t the same length, such that the sum $\sum_e |\ell'(e) - \ell(e)|$ is as small as possible. Describe and analyze an efficient algorithm to solve this optimization problem. *[Hint: Modify your solutions to parts (a)–(c).]*

♦8. An ***integer program*** is a linear program with the additional constraint that the variables must take only integer values.

(a) Prove that deciding whether an integer program has a feasible solution is NP-complete.

(b) Prove that finding the optimal feasible solution to an integer program is NP-hard.

[Hint: Almost any NP-hard decision problem can be formulated as an integer program. Pick your favorite.]

9. A *convex combination* of n points $x_1, x_2, \ldots, x_n \in \mathbb{R}^d$ is weighted average of those points:

$$p = \sum_{i=1}^{n} \lambda_i x_i$$

for some coefficients $\lambda_1, \lambda_2, \ldots, \lambda_n$ such that $\lambda_i \geq 0$ for all i and $\sum_{i=1}^{n} \lambda_i = 1$. The *convex hull* of a set of points X is the set of all convex combinations of points in X. For example, the convex hull of two points is the line segment between them, the convex hull of three points is the (filled) triangle with those three vertices, and so on.

Carathéodory's theorem states that for any set of points $X \subset \mathbb{R}^d$, every point in the convex hull of X lies in the convex hull of $d+1$ points of X. For example, when

$d = 2$, the theorem states every point inside a convex polygon P is also inside the triangle spanned by three vertices of P.

Prove Carathéodory's theorem. *[Hint: Express "p lies in the convex hull of X" as a system of linear inequalities over the variables λ_i.]*

10. *Helly's theorem* states that for any collection of convex bodies in $\mathbb{R}^d$, if every $d + 1$ of them intersect, then there is a point lying in the intersection of all of them. Prove Helly's theorem for the special case where the convex bodies are halfspaces. Equivalently, show that if a system of linear inequalities $Ax \leq b$ is infeasible, then we can select a subset of $d + 1$ inequalities such that the resulting subsystem infeasible. *[Hint: Construct a linear program from the system by imposing a* 0 *objective vector, and consider the dual LP.]*

© 2018 Jeff Erickson

Simplicibus itaque verbis gaudet Mathematica Veritas, cum etiam per se simplex sit Veritatis oratio. [And thus Mathematical Truth prefers simple words, because the language of Truth is itself simple.]

— Tycho Brahe (quoting Seneca (quoting Euripides))
Epistolarum astronomicarum liber primus (1596)

When a jar is broken, the space that was inside
Merges into the space outside.
In the same way, my mind has merged in God;
To me, there appears no duality.

— Sankara, *Viveka-Chudamani* (c. 700), translator unknown

I

Linear Programming Algorithms

[Read Chapters G and H first.]
Status: Half-finished.

In this chapter I will describe several variants of the *simplex algorithm* for solving linear programming problems, first proposed by George Dantzig in 1947. Although most variants of the simplex algorithm perform well in practice, no deterministic simplex variant is known to run in sub-exponential time in the worst case.[1] However, if the dimension of the problem is considered a constant, there are several variants of the simplex algorithm that run in *linear* time. I'll describe a particularly simple randomized algorithm due to Raimund Seidel.

My approach to describing these algorithms relies much more heavily on geometric intuition than the usual linear-algebraic formalism. This works better for me, but your mileage may vary. For a more traditional description of the simplex algorithm, see Robert Vanderbei's excellent textbook *Linear Programming: Foundations and Extensions*

[1]However, there are *randomized* variants of the simplex algorithm that run in subexponential *expected* time, most notably the RandomFacet algorithm analyzed by Gil Kalai in 1992, and independently by Jiří Matoušek, Micha Sharir, and Emo Welzl in 1996. No randomized variant is known to run in polynomial time. In particular, in 2010, Oliver Friedmann, Thomas Dueholm Hansen, and Uri Zwick proved that the worst-case expected running time of RandomFacet is superpolynomial.

© 2018 Jeff Erickson

[Springer, 2001], which can be freely downloaded (but not legally printed) from the author's website.

I.1 Bases, Feasibility, and Local Optimality

Consider the canonical linear program $\max\{c \cdot x \mid Ax \leq b, x \geq 0\}$, where A is an $n \times d$ constraint matrix, b is an n-dimensional coefficient vector, and c is a d-dimensional objective vector. We will interpret this linear program geometrically as looking for the lowest point in a convex polyhedron in $\mathbb{R}^d$, described as the intersection of $n + d$ halfspaces. As in the last lecture, we will consider only *non-degenerate* linear programs: Every subset of d constraint hyperplanes intersects in a single point; at most d constraint hyperplanes pass through any point; and objective vector is linearly independent from any $d - 1$ constraint vectors.

A ***basis*** is a subset of d constraints, which by our non-degeneracy assumption must be linearly independent. The ***location*** of a basis is the unique point x that satisfies all d constraints with equality; geometrically, x is the unique intersection point of the d hyperplanes. The ***value*** of a basis is $c \cdot x$, where x is the location of the basis. There are precisely $\binom{n+d}{d}$ bases. Geometrically, the set of constraint hyperplanes defines a decomposition of $\mathbb{R}^d$ into convex polyhedra; this cell decomposition is called the ***arrangement*** of the hyperplanes. Every subset of d hyperplanes (that is, every basis) defines a *vertex* of this arrangement (the location of the basis). I will use the words 'vertex' and 'basis' interchangeably.

A basis is ***feasible*** if its location x satisfies all the linear constraints, or geometrically, if the point x is a vertex of the polyhedron. If there are no feasible bases, the linear program is ***infeasible***.

A basis is ***locally optimal*** if its location x is the optimal solution to the linear program with the same objective function and *only* the constraints in the basis. Geometrically, a basis is locally optimal if its location x is the lowest point in the intersection of those d halfspaces. A careful reading of the proof of the Strong Duality Theorem reveals that local optimality is the dual equivalent of feasibility; a basis is locally optimal for a linear program Π if and only if the same basis is feasible for the dual linear program $\amalg$. For this reason, locally optimal bases are sometimes also called ***dual feasible***. If there are no locally optimal bases, the linear program is ***unbounded***.[2]

Two bases are ***neighbors*** if they have $d - 1$ constraints in common. Equivalently, in geometric terms, two vertices are neighbors if they lie on a *line* determined by some $d - 1$ constraint hyperplanes. Every basis is a neighbor of exactly dn other bases; to change a basis into one of its neighbors, there are d choices for which constraint to remove and n

[2]For non-degenerate linear programs, the feasible region is unbounded in the objective direction if and only if no basis is locally optimal. However, there are degenerate linear programs with no locally optimal basis that are infeasible.

choices for which constraint to add. The graph of vertices and edges on the boundary of the feasible polyhedron is a subgraph of the basis graph.

The Weak Duality Theorem implies that the value of every feasible basis is less than or equal to the value of every locally optimal basis; equivalently, every feasible vertex is higher than every locally optimal vertex. The Strong Duality Theorem implies that (under our non-degeneracy assumption), if a linear program has an optimal solution, it is the *unique* vertex that is both feasible and locally optimal. Moreover, the optimal solution is both the lowest feasible vertex and the highest locally optimal vertex.

I.2 The Simplex Algorithm

Primal: Falling Marbles

From a geometric standpoint, Dantzig's simplex algorithm is very simple. The input is a set H of halfspaces; we want the lowest vertex in the intersection of these halfspaces.

```
PrimalSimplex(H):
  if ∩H = ∅
      return Infeasible
  x ← any feasible vertex
  while x is not locally optimal
      ⟨⟨pivot downward, maintaining feasibility⟩⟩
      if every feasible neighbor of x is higher than x
          return Unbounded
      else
          x ← any feasible neighbor of x that is lower than x
  return x
```

Let's ignore the first three lines for the moment. The algorithm maintains a feasible vertex x. At each so-called ***pivot*** operation, the algorithm moves to a *lower* vertex, so the algorithm never visits the same vertex more than once. Thus, the algorithm must halt after at most $\binom{n+d}{d}$ pivots. When the algorithm halts, either the feasible vertex x is locally optimal, and therefore the optimum vertex, or the feasible vertex x is not locally optimal but has no lower feasible neighbor, in which case the feasible region must be unbounded.

Notice that we have not specified *which* neighbor to choose at each pivot. Many different pivoting rules have been proposed, but for almost every known pivot rule, there is an input polyhedron that requires an exponential number of pivots under that rule. *No* pivoting rule is known that guarantees a polynomial number of pivots in the worst case, or even in expectation.[3]

[3] In 1957, Hirsch conjectured that for *any* linear programming instance with d variables and $n+d$ constraints, starting at any feasible basis, there is a sequence of ***at most n*** pivots that leads to the optimal basis. This long-standing conjecture was finally disproved in 2010 by Fransisco Santos, who described an counterexample with 43 variables and 86 constraints, where the worst-case number of required pivots is 44.

Dual: Rising Bubbles

We can also geometrically interpret the execution of the simplex algorithm on the dual linear program II. Again, the input is a set H of halfspaces, and we want the lowest vertex in the intersection of these halfspaces. By the Strong Duality Theorem, this is the same as the *highest locally-optimal* vertex in the hyperplane arrangement.

```
DualSimplex(H):
  if there is no locally optimal vertex
      return Unbounded
  x ← any locally optimal vertex
  while x is not feasbile
      ⟨⟨pivot upward, maintaining local optimality⟩⟩
      if every locally optimal neighbor of x is lower than x
          return Infeasible
      else
          x ← any locally-optimal neighbor of x that is higher than x
  return x
```

Let's ignore the first three lines for the moment. The algorithm maintains a locally optimal vertex x. At each pivot operation, it moves to a *higher* vertex, so the algorithm never visits the same vertex more than once. Thus, the algorithm must halt after at most $\binom{n+d}{d}$ pivots. When the algorithm halts, either the locally optimal vertex x is feasible, and therefore the optimum vertex, or the locally optimal vertex x is not feasible but has no higher locally optimal neighbor, in which case the problem must be infeasible.

Figure I.1. The primal simplex (falling marble) and dual simplex (rising bubble) algorithms in action.

From the standpoint of linear algebra, there is absolutely no difference between running PrimalSimplex on any linear program Π and running DualSimplex on the

dual linear program II. The actual *code* is identical. The only difference between the two algorithms is how we interpret the linear algebra geometrically.

I.3 Computing the Initial Basis

To complete our description of the simplex algorithm, we need to describe how to find the initial vertex x in the third line of PrimalSimplex or DualSimplex. There are several methods to find feasible or locally optimal bases, but perhaps the most natural method uses the simplex algorithm itself. Our approach relies on two simple observations.

First, the feasibility of a vertex does not depend on the choice of objective vector; a vertex is either feasible for every objective function or for none. Equivalently (by duality), the local optimality of a vertex does not depend on the exact location of the d hyperplanes, but only on their normal directions and the objective function; a vertex is either locally optimal for every translation of the hyperplanes or for none. In terms of the original matrix formulation, feasibility depends on A and b but not c, and local optimality depends on A and c but not b.

Second, *every* basis is locally optimal for *some* objective vector. Specifically, it suffices to choose any vector that has a positive inner product with each of the normal vectors of the d chosen hyperplanes. Equivalently, *every* basis is feasible for some offset vector. Specifically, it suffices to translate the d chosen hyperplanes so that they pass through the origin, and then translate all other halfspaces so that they strictly contain the origin.

Thus, to find an initial feasible vertex for the primal simplex algorithm, we can choose an *arbitrary* vertex x, *rotate* the objective function so that x becomes locally optimal, and then find the optimal vertex for the *rotated* objective function by running the (dual) simplex algorithm. This vertex must be feasible, even after we restore the original objective function!

Equivalently, to find an initial locally optimal vertex for the dual simplex algorithm, we can choose an *arbitrary* vertex x, *translate* the constraint hyperplanes so that x becomes feasible, and then find the optimal vertex for the *translated* constraints by running the (primal) simplex algorithm. This vertex must be locally optimal, even after we restore the hyperplanes to their original locations!

Pseudocode for both algorithms is given in Figures I.4 and I.5. As usual, the input to both algorithms is a set H of halfspaces, and the algorithms either return the lowest vertex in the intersection of those halfspaces, report that the linear program is infeasible, or report that the linear program is unbounded.

Figure I.2. Primal simplex with dual initialization: (a) Choose any basis. (b) Rotate the objective to make the basis locally optimal, and pivot "up" to a feasible basis. (c) Pivot down to the optimum basis for the original objective.

Figure I.3. Dual simplex with primal optimization: (a) Choose any basis. (b) Translate the constraints to make the basis feasible, and pivot down to a locally optimal basis. (c) Pivot up to the optimum basis for the original constraints.

```
DualPrimalSimplex(H):
  x ← any vertex
  H̃ ← any rotation of H that makes x locally optimal
  while x is not feasible
      if every locally optimal neighbor of x is lower (wrt H̃) than x
          return Infeasible
      else
          x ← any locally optimal neighbor of x that is higher (wrt H̃) than x
  while x is not locally optimal
      if every feasible neighbor of x is higher than x
          return Unbounded
      else
          x ← any feasible neighbor of x that is lower than x
  return x
```

Figure I.4. The primal simplex algorithm with dual initialization.

```
PrimalDualSimplex(H):
  x ← any vertex
  H̃ ← any translation of H that makes x feasible

  while x is not locally optimal
      if every feasible neighbor of x is higher (wrt H̃) than x
          return Unbounded
      else
          x ← any feasible neighbor of x that is lower (wrt H̃) than x

  while x is not feasible
      if every locally optimal neighbor of x is lower than x
          return Infeasible
      else
          x ← any locally-optimal neighbor of x that is higher than x
  return x
```

Figure I.5. The dual simplex algorithm with primal initializzation.

I.4 Network Simplex

This section needs revision.

★★★

Our first natural examples of linear programming problems were shortest paths, maximum flows, and minimum cuts in edge-weighted graphs. It is instructive to reinterpret the behavior of the abstract simplex algorithm in terms of the original input graphs; this reinterpretation allows for a much more efficient implementation of the simplex algorithm, which is normally called ***network simplex***.

As a concrete working example, I will consider a special case of the minimum-cost flow problem called the ***transshipment problem***. The input consists of a directed graph $G = (V, E)$, a ***balance*** function $b : V \to \mathbb{R}$, and a ***cost*** function $\$: E \to \mathbb{R}$, but no capacities or lower bounds on the edges. Our goal is to compute a ***flow*** function $f : E \to \mathbb{R}$ that is non-negative everywhere, satisfies the balance constraint

$$\sum_{u \to v} f(u \to v) - \sum_{v \to w} f(v \to w) = b(v)$$

at every vertex v, and minimizes the total cost $\sum_e f(e) \cdot \$(e)$.

We can easily express this problem as a linear program with a variable for each edge and constraints for each vertex and edge.

$$\begin{array}{lll} \text{maximize} & \displaystyle\sum_{u \to v} \$(u \to v) \cdot f(u \to v) & \\ \text{subject to} & \displaystyle\sum_{u \to v} f(u \to v) - \sum_{v \to w} f(v \to w) = b(v) & \text{for every vertex } v \neq s \\ & f(u \to v) \geq 0 & \text{for every edge } u \to v \end{array}$$

Here I've omitted the balance constraint for some fixed vertex s, because it is redundant; if f is balanced at every vertex except s, then f must be balanced at s as well. By interpreting the balance, cost, and flow functions as vectors, we can write this linear program more succinctly as follows:

$$\boxed{\begin{array}{ll} \max & \$ \cdot f \\ \text{s.t.} & Af = b \\ & f \geq 0 \end{array}}$$

Here A is the ***vertex-edge incidence matrix*** of G; this matrix has one row for each edge and one column for each vertex, and whose entries are defined as follows:

$$A(x \to y, v) = \begin{cases} 1 & \text{if } v = y \\ -1 & \text{if } v = x \\ 0 & \text{otherwise} \end{cases}$$

Let $\overline{G} = (V, \overline{E})$ be the undirected version of G, defined by setting $\overline{E} = \{uv \mid u \to v \in E\}$. In the following arguments, I will refer to "undirected cycles" and "spanning trees" in G; these phrases are shorthand for the subset of directed edges in G corresponding to undirected cycles and spanning trees in $\overline{G}$.

To simplify the remaining presentation, I will make two non-degeneracy assumptions:

- The cost vector $\$$ is non-degenerate: No residual cycle has cost 0.
- The balance vector is non-degenerate: No non-empty proper subset of vertices has total balance 0.

Because the transshipment LP has E variables, a basis consists of E linearly independent constraints. We call a basis ***balanced*** if it contains all $V-1$ balance constraints; any flow consistent with a balanced basis is balanced at *every* vertex of G. Every balanced basis contains exactly $E - V + 1$ edge constraints, and therefore *omits* exactly $V - 1$ edge constraints. We call an edge *fixed* if its constraint is included in the basis and *free* otherwise. Any flow consistent with a balanced basis is zero on every fixed edge and non-negative on every free edge.

Lemma I.1. *For every balanced basis, the free edges define a spanning tree of G; conversely, for every spanning tree T of G, there is a balanced basis for which T is the set of free edges.*[4]

Proof: First, fix an arbitrary balanced basis, let f be any flow consistent with that basis, and let T be the set of $V-1$ free edges for that basis. (The flow f need not be feasible.)

[4]More generally, every basis (balanced or not) is associated with a spanning *forest* F; the basis contains edge constraints for every edge *not* in F and all but one vertex constraint in each component of F.

For the sake of argument, suppose T contains an undirected cycle. Then by pushing flow around that cycle, we can obtain another (not necessarily feasible) flow f' that is still consistent with our fixed basis. So the basis constraints do not determine a unique flow, which means the constraints are not linearly independent, contradicting the definition of basis. We conclude that T is acyclic, and therefore defines a spanning tree of G.

On the other hand, suppose T is an arbitrary spanning tree of G. We define a function $\mathit{flow}_T : E \to \mathbb{R}$ as follows:

- For each edge $u{\to}v \in T$, we define $\mathit{flow}_T(u{\to}v)$ to be sum of balances in the component of $T \setminus u{\to}v$ that contains v. Our non-degeneracy assumption implies that $\mathit{flow}_T(u{\to}v) \neq 0$.
- For each edge $u{\to}v \notin T$, we define $\mathit{flow}_T(u{\to}v) = 0$.

Routine calculations imply flow_T is balanced at every vertex; moreover, flow_T is the *unique* flow in G that is non-zero only on edges of T. We conclude that the $V-1$ balance constraints and the $E-V+1$ edge constraints for edges not in T are linearly independent; in other words, T is the set of free edges of a balanced basis. □

For any spanning tree T and any edges $u{\to}v \notin T$, let $\boldsymbol{\mathit{cycle}_T(u{\to}v)}$ denote the directed cycle consisting of $u{\to}v$ and the unique residual path in T from v to u. Our non-degeneracy assumption implies that the total cost $\$(\mathit{cycle}_T(u{\to}v))$ of this cycle is not equal to zero. We define the ***slack*** of each edge in G as follows:

$$\mathit{slack}_T(u{\to}v) := \begin{cases} 0 & \text{if } u{\to}v \in T \\ \$(\mathit{cycle}_T(u{\to}v)) & \text{if } u{\to}v \notin T \end{cases}$$

The function $\mathit{flow}_T : E \to \mathbb{R}$ is the location of the balanced basis associated with T; the function $\mathit{slack}_T : E \to \mathbb{R}$ is essentially the location of the corresponding *dual* basis. With these two functions in hand, we can characterize balanced bases as follows:

- The basis associated with any spanning tree T is *feasible* (and thus the dual basis is locally optimal) if and only if $\mathit{flow}_T(e) \geq 0$ (and therefore $\mathit{flow}_T(e) > 0$) for every edge $e \in T$.
- The basis associated with any spanning tree T is *locally optimal* (and thus the dual basis is feasible) if and only if $\mathit{slack}_T(e) \geq 0$ (and therefore $\mathit{slack}_T(e) > 0$) for every edge $e \notin T$.

Notice that the complementary slackness conditions are automatically satisfied: For any edge e, and for any spanning tree T, we have $\mathit{flow}_T(e) \cdot \mathit{slack}_T(e) = 0$. In particular, if T is the optimal basis, then either $\mathit{flow}_T(e) > 0$ and $\mathit{slack}_T(e) = 0$, or $\mathit{flow}_T(e) = 0$ and $\mathit{slack}_T(e) > 0$.

A pivot in the simplex algorithm modifies the current basis by removing one constraint and adding another. For the transshipment LP, a pivot modifies a spanning tree T by adding an edge $e_{\text{in}} \notin T$ and removing an edge $e_{\text{out}} \in T$ to obtain a new spanning tree T'.

- The leaving edge e_{out} must lie in the unique residual cycle in $T + e_{\text{in}}$. The pivot modifies the flow function by pushing flow around the unique residual cycle in $T + e_{\text{in}}$, so that some edge e_{out} becomes empty. In particular, the pivot decreases the overall cost of the flow by $\mathit{flow}_T(e_{\text{out}}) \cdot \mathit{slack}_T(e_{\text{in}})$.
- Equivalently, the entering edge e_{in} must have one endpoint in each component of $T - e_{\text{out}}$. Let S be the set of vertices in the component of $T - e_{\text{out}}$ containing the tail of e_{out}. The pivot subtracts $\mathit{slack}_T(e_{\text{in}})$ from the slack of every edge from S to $V \setminus S$, and adds $\mathit{slack}_T(e_{\text{in}})$ to the slack of every edge from $V \setminus S$ to S.

The primal simplex algorithm starts with an arbitrary feasible basis and then repeatedly pivots to a new feasible basis with smaller cost. For the transshipment LP, we can find an initial feasible flow using the FEASIBLEFLOW algorithm from Chapter F. Each primal simplex pivot finds an edge e_{in} with negative slack and pushes flow around $\mathit{cycle}_T(e_{\text{in}})$ until some edge e_{out} is saturated. In other words, *the primal network simplex algorithm is an implementation of cycle cancellation*.

The dual simplex algorithm starts with an arbitrary locally optimal basis and then repeatedly pivots to a new locally optimal basis with larger cost. For the transshipment LP, the shortest-path tree rooted at any vertex provides a locally optimal basis. Each pivot operation finds an edge e_{out} with negative flow, removes it from the current spanning tree, and then adds the edge e_{in} whose slack is as small as possible.

★★★

I'm not happy with this presentation. I really need to reformulate the dual LP in terms of slacks, instead of the standard "distances", so that I can talk about pushing slack across cuts, just like pushing flow around cycles. This might be helped by a general discussion of cycle/circulation and cut/cocycle spaces of G: (1) orthogonal complementary subspaces of the edge/pseudoflow space of G, (2) generated by fundamental cycles and fundamental cuts of any spanning tree of G. Also, this needs examples/figures.

I.5 Linear Expected Time for Fixed Dimensions

★★★

This section needs careful revision.

In most geometric applications of linear programming, the number of variables is a small constant, but the number of constraints may still be very large.

The input to the following algorithm is a set H of n halfspaces and a set B of b hyperplanes. (B stands for *basis*.) The algorithm returns the lowest point in the intersection of the halfspaces in H and the hyperplanes B. At the top level of recursion, B is empty. I will implicitly assume that the linear program is both feasible and bounded. (If necessary, we can guarantee boundedness by adding a single halfspace to H, and we can guarantee feasibility by adding a dimension.) A point x ***violates*** a constraint h if it is not contained in the corresponding halfspace.

```
SeidelLP(H, B):
  if |B| = d
      return ⋂B
  if |H ∪ B| = d
      return ⋂(H ∪ B)
  h ← random element of H
  x ← SeidelLP(H \ h, B)        (*)
  if x violates h
      return SeidelLP(H \ h, B ∪ ∂h)
  else
      return x
```

The point x recursively computed in line $(*)$ is the optimal solution if and only if the random halfspace h is *not* one of the d halfspaces that define the optimal solution. In other words, the probability of calling SeidelLP$(H, B \cup h)$ is exactly $(d-b)/n$. Thus, we have the following recurrence for the expected number of recursive calls for this algorithm:

$$T(n,b) = \begin{cases} 1 & \text{if } b = d \text{ or } n + b = d \\ T(n-1,b) + \dfrac{d-b}{n} \cdot T(n-1,b+1) & \text{otherwise} \end{cases}$$

The recurrence is somewhat simpler if we write $\delta = d - b$:

$$T(n,\delta) = \begin{cases} 1 & \text{if } \delta = 0 \text{ or } n = \delta \\ T(n-1,\delta) + \dfrac{\delta}{n} \cdot T(n-1,\delta-1) & \text{otherwise} \end{cases}$$

It's easy to prove by induction that $T(n,\delta) = O(\delta!\, n)$:

$$\begin{aligned} T(n,\delta) &= T(n-1,\delta) + \frac{\delta}{n} \cdot T(n-1,\delta-1) \\ &\le \delta!\,(n-1) + \frac{\delta}{n}(\delta-1)! \cdot (n-1) && \text{[induction hypothesis]} \\ &= \delta!\,(n-1) + \delta!\,\frac{n-1}{n} \\ &\le \delta!\, n \end{aligned}$$

At the top level of recursion, we perform one violation test in $O(d)$ time. In each of the base cases, we spend $O(d^3)$ time computing the intersection point of d hyperplanes, and in the first base case, we spend $O(dn)$ additional time testing for violations. More careful analysis implies that the algorithm runs in $\boldsymbol{O(d! \cdot n)}$ ***expected time***.

Exercises

1. Fix a non-degenerate linear program in canonical form with d variables and $n+d$ constraints.
   (a) Prove that every *feasible* basis has exactly d *feasible* neighbors.
   (b) Prove that every *locally optimal* basis has exactly n *locally optimal* neighbors.

2. (a) Give an example of a non-empty polyhedron $Ax \leq b$ that is unbounded for *every* objective vector c.
   (b) Give an example of an infeasible linear program whose dual is also infeasible.

   In both cases, your linear program will be degenerate.

3. Describe and analyze an algorithm that solves the following problem in $O(n)$ time: Given n red points and n blue points in the plane, either find a line that separates every red point from every blue point, or prove that no such line exists.

4. In this exercise, we develop another standard method for computing an initial feasible basis for the primal simplex algorithm. Suppose we are given a canonical linear program Π with d variables and $n+d$ constraints as input:

$$\boxed{\begin{array}{ll} \max & c \cdot x \\ \text{s.t.} & Ax \leq b \\ & x \geq 0 \end{array}}$$

   To compute an initial feasible basis for Π, we solve a modified linear program Π' defined by introducing a new variable λ and two new constraints $0 \leq \lambda \leq 1$, and modifying the objective function:

$$\boxed{\begin{array}{ll} \max & \lambda \\ \text{s.t.} & Ax - b\lambda \leq 0 \\ & \lambda \leq 1 \\ & x, \lambda \geq 0 \end{array}}$$

   (a) Prove that $x_1 = x_2 = \cdots = x_d = \lambda = 0$ is a feasible basis for Π'.
   (b) Prove that Π is feasible if and only if the optimal value for Π' is 1.
   (c) What is the dual of Π'?

5. Suppose you have a subroutine that can solve linear programs in polynomial time, but only if they are both feasible and bounded. Describe an algorithm that solves *arbitrary* linear programs in polynomial time. Your algorithm should return an

optimal solution if one exists; if no optimum exists, your algorithm should report that the input instance is UNBOUNDED or INFEASIBLE, whichever is appropriate. *[Hint: Add one variable and one constraint.]*

6. Suppose your are given a rooted tree T, where every edge e has two associated values: a non-negative *length* $\ell(e)$, and a *cost* $\$(e)$ (which may be positive, negative, or zero). Your goal is to add a non-negative *stretch* $s(e) \geq 0$ to the length of every edge e in T, subject to the following conditions:
   - Every root-to-leaf path π in T has the same total stretched length $\sum_{e\in\pi}(\ell(e)+s(e))$
   - The total *weighted* stretch $\sum_e s(e) \cdot \$(\ell)$ is as small as possible.

   (a) Give a concise linear programming formulation of this problem.

   (b) Prove that in any optimal solution to this problem, we have $s(e) = 0$ for every edge on some longest root-to-leaf path in T. In other words, prove that the optimally stretched tree has the same depth as the input tree. *[Hint: What is a basis in your linear program? When is a basis feasible?]*

   (c) Describe and analyze an algorithm that solves this problem in $O(n)$ time. Your algorithm should either compute the minimum total weighted stretch, or report correctly that the total weighted stretch can be made arbitrarily negative.

7. Recall that the single-source shortest path problem can be formulated as a linear programming problem, with one variable d_v for each vertex $v \neq s$ in the input graph, as follows:

$$\begin{array}{lrl} \text{maximize} & \displaystyle\sum_v d_v & \\ \text{subject to} & d_v \leq \ell_{s\to v} & \text{for every edge } s \to v \\ & d_v - d_u \leq \ell_{u\to v} & \text{for every edge } u \to v \text{ with } u \neq s \\ & d_v \geq 0 & \text{for every vertex } v \neq s \end{array}$$

   This problem asks you to describe the behavior of the simplex algorithm on this linear program in terms of distances. Assume that the edge weights $\ell_{u\to v}$ are all non-negative and that there is a unique shortest path between any two vertices in the graph.

   (a) What is a basis for this linear program? What is a feasible basis? What is a locally optimal basis?

(b) Show that in the optimal basis, every variable d_v is equal to the shortest-path distance from s to v.

(c) Describe the primal simplex algorithm for the shortest-path linear program directly in terms of vertex distances. In particular, what does it mean to pivot from a feasible basis to a neighboring feasible basis, and how can we execute such a pivot quickly?

(d) Describe the dual simplex algorithm for the shortest-path linear program directly in terms of vertex distances. In particular, what does it mean to pivot from a locally optimal basis to a neighboring locally optimal basis, and how can we execute such a pivot quickly?

(e) Is Dijkstra's algorithm an instance of network simplex? Is Shimbel-Bellman-Ford? Justify your answers.

(f) Using the results in problem 9, prove that if the edge lengths $\ell_{u\to v}$ are all integral, then the optimal distances d_v are also integral.

8. The maximum (s,t)-flow problem can be formulated as a linear programming problem, with one variable $f_{u\to v}$ for each edge $u \to v$ in the input graph:

$$\begin{aligned}
\text{maximize} \quad & \sum_w f_{s\to w} - \sum_u f_{u\to s} \\
\text{subject to} \quad & \sum_w f_{v\to w} - \sum_u f_{u\to v} = 0 && \text{for every vertex } v \neq s, t \\
& f_{u\to v} \leq c_{u\to v} && \text{for every edge } u \to v \\
& f_{u\to v} \geq 0 && \text{for every edge } u \to v
\end{aligned}$$

This problem asks you to describe the behavior of the simplex algorithm on this linear program in terms of flows.

(a) What is a basis for this linear program? What is a feasible basis? What is a locally optimal basis?

(b) Show that the optimal basis represents a maximum flow.

(c) Describe the primal simplex algorithm for the flow linear program directly in terms of flows. In particular, what does it mean to pivot from a feasible basis to a neighboring feasible basis, and how can we execute such a pivot quickly?

(d) Describe the dual simplex algorithm for the flow linear program directly in terms of flows. In particular, what does it mean to pivot from a locally optimal basis to a neighboring locally optimal basis, and how can we execute such a pivot quickly?

(e) Is the Ford-Fulkerson augmenting-path algorithm an instance of network simplex? Justify your answer. *[Hint: There is a one-line argument.]*

(f) Using the results in problem 9, prove that if the capacities $c_{u\to v}$ are all integral, then the maximum flow values $f_{u\to v}$ are also integral.

9. A ***minor*** of a matrix A is the submatrix defined by any subset of the rows and any subset of the columns. A matrix A is ***totally unimodular*** if, for every square minor M, the determinant of M is -1, 0, or 1.

   (a) Let A be an arbitrary totally unimodular matrix.

      i. Prove that the transposed matrix $A^\top$ is also totally unimodular.

      ii. Prove that negating any row or column of A leaves the matrix totally unimodular.

      iii. Prove that the block matrix $[A \mid I]$ is totally unimodular.

   (b) Prove that for any totally unimodular matrix A and any integer vector b, the canonical linear program $\max\{c \cdot x \mid Ax \leq b, x \geq 0\}$ has an *integer* optimal solution. *[Hint: Cramer's rule.]*

   (c) The ***unsigned incidence matrix*** of an undirected graph $G = (V, E)$ is an $|V| \times |E|$ matrix A, with rows indexed by vertices and columns indexed by edges, where for each row v and column uw, we have

$$A[v, uw] = \begin{cases} 1 & \text{if } v = u \text{ or } v = w \\ 0 & \text{otherwise} \end{cases}$$

   Prove that the unsigned incidence matrix of every *bipartite* graph G is totally unimodular. *[Hint: Each square minor corresponds to a subgraph H of G with k vertices and k edges, for some integer k. Argue that at least one of the following statements must be true: (1) H is disconnected; (2) H has a vertex with degree 1; (3) H is an even cycle.]*

   (d) Prove for every *non*-bipartite graph G that the unsigned incidence matrix of G is *not* totally unimodular. *[Hint: Consider any odd cycle.]*

   (e) The ***signed incidence matrix*** of a directed graph $G = (V, E)$ is also an $|V| \times |E|$ matrix A, with rows indexed by vertices and columns indexed by edges, where for each row v and column $u{\to}w$, we have

$$A[u, v{\to}w] = \begin{cases} 1 & \text{if } v = w \\ -1 & \text{if } v = u \\ 0 & \text{otherwise} \end{cases}$$

   Prove that the signed incidence matrix of every directed graph G is totally unimodular.

© 2018 Jeff Erickson

Le mieux est l'ennemi du bien. [The best is the enemy of the good.]

— Voltaire, *La Bégueule* (1772)

Who shall forbid a wise skepticism, seeing that there is no practical question on which any thing more than an approximate solution can be had?

— Ralph Waldo Emerson, *Representative Men* (1850)

We dance round in a ring and suppose,
But the secret sits in the middle and knows.

— Robert Frost, "The Secret Sits" (1942)

Now, distrust of corporations threatens our still-tentative economic recovery; it turns out greed is bad, after all.

— Paul Krugman, "Greed is Bad", *The New York Times* (June 4, 2002)

J

Approximation Algorithms

[Read Chapter 12 first.]

Status: Needs revision.

J.1 Load Balancing

On the future viral-hit online reality game show *Grunt Work*, broadcast every Wednesday at 3am on YouTube Green, the contestants are given a series of utterly pointless tasks to perform. Each task has a predetermined time limit; for example, "Sharpen this pencil for 17 seconds," or "Pour pig's blood on your head and sing The Star-Spangled Banner for two minutes," or "Listen to this 75-minute algorithms lecture." The directors of the show want you to assign each task to one of the contestants, so that the last task is completed as early as possible. When your predecessor correctly informed the directors that their problem is NP-hard, he was immediately fired. "Time is money!" they screamed at him. "We don't need perfection. Wake up, dude, this is the *internet*!"

Less facetiously, suppose we have a set of n jobs, which we want to assign to m machines. We are given an array $T[1..n]$ of non-negative numbers, where $T[j]$ is the running time of job j. We can describe an *assignment* by an array $A[1..n]$, where

© 2018 Jeff Erickson

$A[j] = i$ means that job j is assigned to machine i. The *makespan* of an assignment is the maximum time that any machine is busy:

$$makespan(A) = \max_i \sum_{A[j]=i} T[j]$$

The *load balancing* problem is to compute the assignment with the smallest possible makespan.

It's not hard to prove that the load balancing problem is NP-hard by reduction from PARTITION: The array $T[1..n]$ can be evenly partitioned if and only if there is an assignment to two machines with makespan exactly $\sum_i T[i]/2$. A slightly more complicated reduction from 3PARTITION implies that the load balancing problem is *strongly* NP-hard. If we really need the optimal solution, there is a dynamic programming algorithm that runs in time $O(nM^m)$, where M is the minimum makespan, but that's just horrible.

There is a fairly natural and efficient greedy heuristic for load balancing: consider the jobs one at a time, and assign each job to the machine i with the earliest finishing time $Total[i]$.

```
GreedyLoadBalance(T[1..n], m):
  for i ← 1 to m
      Total[i] ← 0

  for j ← 1 to n
      mini ← arg min_i Total[i]
      A[j] ← mini
      Total[mini] ← Total[mini] + T[j]

  return A[1..m]
```

Theorem J.1. *The makespan of the assignment computed by* GREEDYLOADBALANCE *is at most twice the makespan of the optimal assignment.*

Proof: Fix an arbitrary input, and let OPT denote the makespan of its optimal assignment. The approximation bound follows from two trivial observations. First, the makespan of any assignment (and therefore of the optimal assignment) is at least the duration of the longest job. Second, the makespan of any assignment is at least the total duration of all the jobs divided by the number of machines.

$$OPT \geq \max_j T[j] \qquad \text{and} \qquad OPT \geq \frac{1}{m}\sum_{j=1}^{n} T[j]$$

Now consider the assignment computed by GREEDYLOADBALANCE. Suppose machine i has the largest total running time, and let j be the last job assigned to machine i. Our first trivial observation implies that $T[j] \leq OPT$. To finish the proof, we must show that

$Total[i] - T[j] \leq OPT$. Job j was assigned to machine i because it had the smallest finishing time, so $Total[i] - T[j] \leq Total[k]$ for all k. (Some values $Total[k]$ may have increased since job j was assigned, but that only helps us.) In particular, $Total[i] - T[j]$ is less than or equal to the *average* finishing time over all machines. Thus,

$$Total[i] - T[j] \leq \frac{1}{m}\sum_{i=1}^{m} Total[i] = \frac{1}{m}\sum_{j=1}^{n} T[j] \leq OPT$$

by our second trivial observation. We conclude that the makespan $Total[i]$ is at most $2 \cdot OPT$. □

Proof that GreedyLoadBalance is a 2-approximation algorithm

GreedyLoadBalance is an *online* algorithm: It assigns jobs to machines in the order that the jobs appear in the input array. Online approximation algorithms are useful in settings where inputs arrive in a stream of unknown length—for example, real jobs arriving at a real scheduling algorithm. In this online setting, it may be *impossible* to compute an optimum solution, even in cases where the offline problem (where all inputs are known in advance) can be solved in polynomial time. The study of online algorithms could easily fill an entire one-semester course (alas, not this one).

In our original offline setting, we can improve the approximation factor by sorting the jobs before piping them through the greedy algorithm.

```
SortedGreedyLoadBalance(T[1..n], m):
  sort T in decreasing order
  return GreedyLoadBalance(T, m)
```

Theorem J.2. *The makespan of the assignment computed by SortedGreedyLoadBalance is at most* $3/2$ *times the makespan of the optimal assignment.*

Proof: Let i be the busiest machine in the schedule computed by SortedGreedyLoadBalance. If only one job is assigned to machine i, then the greedy schedule is actually optimal, and the theorem is trivially true. Otherwise, let j be the last job assigned to machine i. Since each of the first m jobs is assigned to a unique machine, we must have $j \geq m+1$. As in the previous proof, we know that $Total[i] - T[j] \leq OPT$.

In *any* schedule, at least two of the first $m+1$ jobs, say jobs k and ℓ, must be assigned to the same machine. Thus, $T[k]+T[\ell] \leq OPT$. Since $\max\{k,\ell\} \leq m+1 \leq j$, and the jobs are sorted in decreasing order by duration, we have

$$T[j] \leq T[m+1] \leq T[\max\{k,\ell\}] = \min\{T[k], T[\ell]\} \leq OPT/2.$$

We conclude that the makespan $Total[i]$ is at most $3 \cdot OPT/2$, as claimed. □

In fact, neither of these analyses is tight. The exact approximation ratio of GreedyLoadBalance is actually $2-1/m$, and the exact approximation ratio of SortedGreedyLoadBalance is actually $4/3-1/(3m)$.

J.2 Generalities

Consider an arbitrary optimization problem. Let $OPT(X)$ denote the value of the optimal solution for a given input X, and let $A(X)$ denote the value of the solution computed by algorithm A given the same input X. We say that A is an ***$\alpha(n)$-approximation algorithm*** if and only if

$$\frac{OPT(X)}{A(X)} \leq \alpha(n) \quad \text{and} \quad \frac{A(X)}{OPT(X)} \leq \alpha(n)$$

for all inputs X of size n. The function $\alpha(n)$ is called the ***approximation factor*** for algorithm A. For any given algorithm, only one of these two inequalities will be important. For maximization problems, where we want to compute a solution whose cost is as small as possible, the first inequality is trivial. For maximization problems, where we want a solution whose value is as large as possible, the second inequality is trivial. A 1-approximation algorithm always returns the exact optimal solution.

Especially for problems where exact optimization is NP-hard, we have little hope of completely characterizing the optimal solution. The secret to proving that an algorithm satisfies some approximation ratio is to find a useful function of the input that provides both lower bounds on the cost of the optimal solution and upper bounds on the cost of the approximate solution. For example, if $OPT(X) \geq f(X)/2$ and $A(X) \leq 5f(X)$ for *any* function f, then A is a 10-approximation algorithm. Finding the right intermediate function can be a delicate balancing act.

J.3 Greedy Vertex Cover

Recall that the *vertex color* problem asks, given a graph G, for the smallest set of vertices of G that cover every edge. This is one of the first NP-hard problems introduced in the first week of class. There is a natural and efficient greedy heuristic[1] for computing a small vertex cover: mark the vertex with the largest degree, remove all the edges incident to that vertex, and recurse.

[1] A *heuristic* is an algorithm that doesn't work.

```
GreedyVertexCover(G):
  C ← ∅
  while G has at least one edge
      v ← vertex in G with maximum degree
      G ← G \ v
      C ← C ∪ v
  return C
```

Obviously this algorithm doesn't compute the optimal vertex cover—that would imply P=NP!—but it does compute a reasonably close approximation.

Theorem J.3. *GreedyVertexCover is an $O(\log n)$-approximation algorithm.*

Proof: For all i, let G_i denote the graph G after i iterations of the main loop, and let d_i denote the maximum degree of any node in G_{i-1}. We can define these variables more directly by adding a few extra lines to our algorithm:

```
GreedyVertexCover(G):
  C ← ∅
  G_0 ← G
  i ← 0
  while G_i has at least one edge
      i ← i + 1
      v_i ← vertex in G_{i-1} with maximum degree
      d_i ← deg_{G_{i-1}}(v_i)
      G_i ← G_{i-1} \ v_i
      C ← C ∪ v_i
  return C
```

Let $|G_{i-1}|$ denote the number of edges in the graph G_{i-1}. Let C^* denote the optimal vertex cover of G, which consists of OPT vertices. Since C^* is also a vertex cover for G_{i-1}, we have

$$\sum_{v \in C^*} \deg_{G_{i-1}}(v) \geq |G_{i-1}|.$$

In other words, the *average* degree in G_i of any node in C^* is at least $|G_{i-1}|/OPT$. It follows that G_{i-1} has at least one node with degree at least $|G_{i-1}|/OPT$. Since d_i is the maximum degree of any node in G_{i-1}, we have

$$d_i \geq \frac{|G_{i-1}|}{OPT}$$

Moreover, for any $j \geq i-1$, the subgraph G_j has no more edges than G_{i-1}, so $d_i \geq |G_j|/OPT$. This observation implies that

$$\sum_{i=1}^{OPT} d_i \;\geq\; \sum_{i=1}^{OPT} \frac{|G_{i-1}|}{OPT} \;\geq\; \sum_{i=1}^{OPT} \frac{|G_{OPT}|}{OPT} \;=\; |G_{OPT}| \;=\; |G| - \sum_{i=1}^{OPT} d_i.$$

In other words, the first OPT iterations of GREEDYVERTEXCOVER remove at least half the edges of G. Thus, after at most $OPT \lg|G| \leq 2\,OPT \lg n$ iterations, all the edges of G have been removed, and the algorithm terminates. We conclude that GREEDYVERTEXCOVER computes a vertex cover of size $O(OPT \log n)$. □

So far we've only proved an *upper bound* on the approximation factor of GREEDYVERTEXCOVER; perhaps a more careful analysis would imply that the approximation factor is only $O(\log \log n)$, or even $O(1)$. Alas, no such improvement is possible. For any integer n, a simple recursive construction gives us an n-vertex graph for which the greedy algorithm returns a vertex cover of size $\Omega(OPT \cdot \log n)$. Details are left as an exercise for the reader.

J.4 Set Cover and Hitting Set

The greedy algorithm for vertex cover can be applied almost immediately to two more general problems: ***minimum set cover*** and ***minimum hitting set***. The input for both of these problems is a *set system* $(X, \mathcal{F})$, where X is a finite *ground set*, and $\mathcal{F}$ is a family of subsets of X. A *set cover* of a set system $(X, \mathcal{F})$ is a subfamily of sets in $\mathcal{F}$ whose union is the entire ground set X. A *hitting set* for $(X, \mathcal{F})$ is a subset of the ground set X that intersects every set in $\mathcal{F}$.

An undirected graph can be cast as a set system in two different ways. In one formulation, the ground set X contains the vertices, and each edge defines a set of two vertices in $\mathcal{F}$. In this formulation, a vertex cover is a hitting set. In the other formulation, the *edges* are the ground set, the *vertices* define the family of subsets, and a vertex cover is a set cover.

Here are the natural greedy algorithms for finding a small set cover and finding a small hitting set. Essentially the same analysis as for our greedy algorithm for vertex cover implies that GREEDYSETCOVER finds a set cover whose size is at most $O(\log|\mathcal{F}|)$ times the size of smallest set cover, and that GREEDYHITTINGSET finds a hitting set whose size is at most $O(\log|X|)$ times the size of the smallest hitting set. These are the best approximation ratios possible (up to lower order terms) unless P=NP.

```
GreedySetCover(X, F):
  C ← ∅
  while X ≠ ∅
      S ← arg max_{S∈F} |S ∩ X|
      X ← X \ S
      C ← C ∪ {S}
  return C
```

```
GreedyHittingSet(X, F):
  H ← ∅
  while F ≠ ∅
      x ← arg max_{x∈X} |{S ∈ F | x ∈ S}|
      F ← F \ {S ∈ F | x ∈ S}
      H ← H ∪ {x}
  return H
```

The similarity between these two algorithms is no coincidence. For any set system $(X, \mathcal{F})$, there is a *dual* set system $(\mathcal{F}, X^*)$ defined as follows. For any element $x \in X$ in the ground set, let x^* denote the subfamily of sets in $\mathcal{F}$ that contain x:

$$x^* = \{S \in \mathcal{F} \mid x \in S\}.$$

Finally, let X^* denote the collection of all subsets of the form x^*:

$$X^* = \{x^* \mid x \in S\}.$$

As an example, suppose X is the set of letters of alphabet and $\mathcal{F}$ is the set of last names of student taking CS 473 this semester. Then X^* has 26 elements, each containing the subset of CS 473 students whose last name contains a particular letter of the alphabet. For example, m^* is the set of students whose last names contain the letter m.

There is a natural bijection between the ground set X and the dual set family X^*. It is a tedious but instructive exercise to prove that the dual of the dual of any set system is isomorphic to the original set system—$(X^*, \mathcal{F}^*)$ is essentially the same as $(X, \mathcal{F})$. It is also easy to prove that a set cover for any set system $(X, \mathcal{F})$ is also a hitting set for the dual set system $(\mathcal{F}, X^*)$, and therefore a hitting set for any set system $(X, \mathcal{F})$ is isomorphic to (or more simply, "is") a set cover for the dual set system $(\mathcal{F}, X^*)$.

J.5 Vertex Cover Redux

The greedy approach doesn't always lead to the best approximation algorithms. Consider the following obviously stupid heuristic for vertex cover:

```
DumbVertexCover(G):
  C ← ∅
  while G has at least one edge
      uv ← any edge in G
      G ← G \ {u, v}
      C ← C ∪ {u, v}
  return C
```

The minimum vertex cover—in fact, *every* vertex cover—contains at least one of the two vertices u and v chosen inside the while loop. It follows immediately that DumbVertexCover is a 2-approximation algorithm! Surprisingly, this "obviously stupid" algorithm is essentially the best polynomial-time approximation algorithm known.

The same stupid idea can be extended to approximate the minimum hitting set for any set system $(X, \mathcal{F})$; the resulting approximation factor is equal to the size of the largest set in $\mathcal{F}$.

J.6 Lightest Vertex Cover: LP Rounding

Now consider a generalization of the minimum vertex cover problem in which every vertex v of the input graph has a non-negative weight $w(v)$, and the goal is to compute a vertex cover C with minimum total weight $w(C) = \sum_{v \in C} w(v)$. Both the greedy and the stupid approximation algorithms can perform arbitrarily badly in this setting. Let G be a path of length 2, where the endpoints have weight 1 and the interior vertex has weight

10^{100}. The lightest vertex cover has weight 2, but the greedy vertex cover has weight 10^{100}, and the stupid vertex cover has weight $10^{100}+1$.

But we can recover a 2-approximation algorithm by casting the problem as an instance of ***integer linear programming***. An integer linear program is just a linear program with the added constraint that all variables are integers. Almost any of the NP-hard problems considered in the previous lecture note can be cast as integer linear programs, which implies that integer linear programming is NP-hard. In particular, the following integer linear program encodes the lightest vertex cover problem. Each feasible solution to this ILP corresponds to some vertex cover; each variable $x(v)$ indicates whether that vertex cover contains the corresponding vertex v.

$$\boxed{\begin{array}{lrl} \text{minimize} & \displaystyle\sum_v w(v)\cdot x(v) & \\ \text{subject to} & x(u)+x(v)\geq 1 & \text{for every edge } uv \\ & x(v)\in\{0,1\} & \text{for every vertex } v \end{array}}$$

Let OPT denote the weight of the lightest vertex cover; OPT is also the optimal objective value for this integer linear program.

Now we're going to do something weird. Consider the linear program obtained from this ILP by removing the integrality constraint; this linear program is usually called the ***linear relaxation*** of the integer linear program.

$$\boxed{\begin{array}{lrl} \text{minimize} & \displaystyle\sum_v w(v)\cdot x(v) & \\ \text{subject to} & x(u)+x(v)\geq 1 & \text{for every edge } uv \\ & 0\leq x(v)\leq 1 & \text{for every vertex } v \end{array}}$$

Like all linear programs, this one can be solved in polynomial time; let x^* denote its optimal solution, and let $\widetilde{\boldsymbol{OPT}}=\sum_v w(v)\cdot x^*(v)$ denote the denote the optimal objective value. Unfortunately, the optimal solution may not be integral; the "indicator" variables $x^*(v)$ may be rational numbers between 0 and 1. Thus, x^* is often called the optimal ***fractional*** solution. Nevertheless, this solution is useful for two reasons.

First, because every feasible solution to the original ILP is also feasible for its linear relaxation, the optimal fractional solution is a lower bound on the optimal integral solution: $\boldsymbol{OPT}\geq\widetilde{\boldsymbol{OPT}}$. As we've already seen, finding lower bounds on the optimal solution is the key to deriving good approximation algorithms.

Second, we can derive a good approximation to the lightest vertex cover by ***rounding*** the optimal fractional solution. Specifically, for each vertex v, let

$$x'(v)=\begin{cases}1 & \text{if } x^*(v)\geq 1/2,\\ 0 & \text{otherwise.}\end{cases}$$

For every edge uv, the constraint $x^*(u)+x^*(v) \geq 1$ implies that $\max\{x^*(u), x^*(v)\} \geq 1/2$, and therefore either $x'(u) = 1$ or $x'(v) = 1$. Thus, x' is the indicator function of a vertex cover. On the other hand, we have $x'(v) \leq 2x^*(v)$ for every vertex v, which implies that

$$\sum_v w(v) \cdot x'(v) \;\leq\; 2\sum_v w(v) \cdot x^*(v) \;=\; 2 \cdot \widetilde{OPT} \;\leq\; 2 \cdot OPT.$$

So we've just described a simple polynomial-time 2-approximation algorithm for lightest vertex cover: Compute the optimal fractional solution, and round it as described above.

J.7 Randomized LP Rounding

We can also round the optimal fractional solution *randomly* by interpreting each fractional value $x^*(v)$ as a probability. For each vertex v, independently set

$$x'(v) = \begin{cases} 1 & \text{with probability } x^*(v), \\ 0 & \text{otherwise.} \end{cases}$$

Our usual argument from linearity of expectation immediately implies that

$$\mathrm{E}\left[\sum_v w(v) \cdot x'(v)\right] \;=\; \sum_v w(v) \cdot x^*(v) \;=\; \widetilde{OPT} \;\leq\; OPT.$$

Unfortunately, x' is not necessarily the indicator function of a vertex cover. The probability that any particular edge uv is uncovered is

$$\Pr[x'(u) = 0 \wedge x'(v) = 0] \;=\; (1 - x^*(u))(1 - x^*(v)).$$

Thanks to the constraint $x^*(u) = x^*(v) \geq 1$, the expression $(1 - x^*(u))(1 - x^*(v))$ is minimized when $x^*(v) = x^*(u) = 1/2$. It follows that each edge uv is covered with probability at least $3/4$; equivalently, we expect x' to cover $3/4$ of the edges of the graph.

To construct an actual vertex cover, we can run the randomized rounding algorithm several times in succession, and take the union of the resulting vertex sets.

```
ApproxLightestVertexCover(G):
  C ← ∅
  x* ← optimal fractional solution
  for i ← 1 to 2 lg n
      for all vertices v
          with probability x*(v)
              C ← C ∪ {x}
  return C
```

Because the random choices in each iteration of the main loop are independent, we can simplify the algorithm by changing the rounding criterion as follows, where $N = 2\lg n$:

$$x'(v) = \begin{cases} 1 & \text{with probability } 1-(1-x^*(v))^N, \\ 0 & \text{otherwise,} \end{cases}$$

Let C be the set of vertices returned by this algorithm. Each edge uv is left uncovered by C with probability $(1-x^*(u))^N(1-x^*(v))^N \leq 1/4^N = 1/n^4$. It follows that C is a vertex cover with probability at least $1-1/n^2$. On the other hand, linearity of expectation immediately implies that the expected size of C is exactly $N \cdot \widetilde{OPT} \leq 2\lg n \cdot OPT$.

If we want to *guarantee* a vertex cover, we can modify the main loop in APPROXLIGHTEST-VERTEXCOVER to continue running until every edge is covered. The loop will terminate after $O(\log n)$ iterations with high probability, and the expected size of the resulting vertex cover is still at most $O(\log n) \cdot OPT$. Thus, at least in expectation, we obtain the same $O(\log n)$ approximation ratio (at least in expectation) as our original greedy algorithm.

This randomized rounding strategy can be generalized to the lightest *set* cover problem, where each subset in the input family has an associated weight, and the goal is to find the lightest collection of subsets that covers the ground set. The resulting rounding algorithm computes a set cover whose expected weight is at most $O(\log|\mathcal{F}|)$ times the weight of the lightest set cover, matching the performance of the greedy algorithm for unweighted set cover. The dual of this algorithm is a randomized $O(\log|X|)$-approximation algorithm for weighted hitting set (where every item in the ground set has a weight).

★★★ Derandomization: Greedy set cover with prices with proof via LP duality

J.8 Traveling Salesman: The Bad News

The ***traveling salesman problem***[2] problem asks for the shortest Hamiltonian cycle in a weighted undirected graph. To keep the problem simple, we can assume without loss of generality that the underlying graph is always the complete graph K_n for some integer n; thus, the input to the traveling salesman problem is just a list of the $\binom{n}{2}$ edge lengths.

Not surprisingly, given its similarity to the Hamiltonian cycle problem, it's quite easy to prove that the traveling salesman problem is NP-hard. Let G be an arbitrary undirected graph with n vertices. We can construct a length function for K_n as follows:

$$\ell(e) = \begin{cases} 1 & \text{if } e \text{ is an edge in } G, \\ 2 & \text{otherwise.} \end{cases}$$

[2]This is sometimes bowdlerized into the traveling sales*person* problem. That's just silly. Who ever heard of a traveling salesperson sleeping with the farmer's child?

Now it should be obvious that if G has a Hamiltonian cycle, then there is a Hamiltonian cycle in K_n whose length is exactly n; otherwise every Hamiltonian cycle in K_n has length at least $n+1$. We can clearly compute the lengths in polynomial time, so we have a polynomial time reduction from Hamiltonian cycle to traveling salesman. Thus, the traveling salesman problem is NP-hard, even if all the edge lengths are 1 and 2.

There's nothing special about the values 1 and 2 in this reduction; we can replace them with any values we like. By choosing values that are sufficiently far apart, we can show that even approximating the shortest traveling salesman tour is NP-hard. For example, suppose we set the length of the 'absent' edges to $n+1$ instead of 2. Then the shortest traveling salesman tour in the resulting weighted graph either has length exactly n (if G has a Hamiltonian cycle) or has length at least $2n$ (if G does not have a Hamiltonian cycle). Thus, if we could approximate the shortest traveling salesman tour within a factor of 2 in polynomial time, we would have a polynomial-time algorithm for the Hamiltonian cycle problem.

Pushing this idea to its limits us the following negative result.

Theorem J.4. *For any function $f(n)$ that can be computed in time polynomial in n, there is no polynomial-time $f(n)$-approximation algorithm for the traveling salesman problem on general weighted graphs, unless P=NP.*

J.9 Traveling Salesman: The Good News

Even though the general traveling salesman problem can't be approximated, a common special case can be approximated fairly easily. The special case requires the edge lengths to satisfy the so-called *triangle inequality*:

$$\ell(u,w) \leq \ell(u,v) + \ell(v,w) \quad \text{for any vertices } u,v,w.$$

This inequality is satisfied for *geometric graphs*, where the vertices are points in the plane (or some higher-dimensional space), edges are straight line segments, and lengths are measured in the usual Euclidean metric. Notice that the length functions we used above to show that the general TSP is hard to approximate do not (always) satisfy the triangle inequality.

With the triangle inequality in place, we can quickly compute a 2-approximation for the traveling salesman tour as follows. First, we compute the minimum spanning tree T of the weighted input graph; this can be done in $O(n^2 \log n)$ time (where n is the number of vertices of the graph) using any of several classical algorithms. Second, we perform a depth-first traversal of T, numbering the vertices in the order that we first encounter them. Because T is a spanning tree, every vertex is numbered. Finally, we return the cycle obtained by visiting the vertices according to this numbering.

Theorem J.5. *A depth-first ordering of the minimum spanning tree gives a 2-approximation of the shortest traveling salesman tour.*

A minimum spanning tree T, a depth-first traversal of T, and the resulting approximate traveling salesman tour.

Proof: Let *OPT* denote the cost of the optimal TSP tour, let *MST* denote the total length of the minimum spanning tree, and let A be the length of the tour computed by our approximation algorithm. Consider the 'tour' obtained by walking through the minimum spanning tree in depth-first order. Since this tour traverses every edge in the tree exactly twice, its length is $2 \cdot MST$. The final tour can be obtained from this one by removing duplicate vertices, moving directly from each node to the next *unvisited* node.; the triangle inequality implies that taking these shortcuts cannot make the tour longer. Thus, $A \leq 2 \cdot MST$. On the other hand, if we remove any edge from the optimal tour, we obtain a spanning tree (in fact a spanning *path*) of the graph; thus, $MST \geq OPT$. We conclude that $A \leq 2 \cdot OPT$; our algorithm computes a 2-approximation of the optimal tour. □

We can improve this approximation factor using the following algorithm discovered by Nicos Christofides in 1976. As in the previous algorithm, we start by constructing the minimum spanning tree T. Then let O be the set of vertices with *odd* degree in T; it is an easy exercise (hint, hint) to show that the number of vertices in O is even.

In the next stage of the algorithm, we compute a *minimum-cost perfect matching* M of these odd-degree vertices. A perfect matching is a collection of edges, where each edge has both endpoints in O and each vertex in O is adjacent to exactly one edge; we want the perfect matching of minimum total length. A minimum-cost perfect matching can be computed in polynomial time using maximum-flow techniques, which are described in a different lecture note.

Now consider the multigraph $T \cup M$; any edge in both T and M appears twice in this multigraph. This graph is connected, and every vertex has even degree. Thus, it contains an *Eulerian circuit*: a closed walk that uses every edge exactly once. We can compute such a walk in $O(n)$ time with a simple modification of depth-first search. To obtain the final approximate TSP tour, we number the vertices in the order they first appear in some Eulerian circuit of $T \cup M$, and return the cycle obtained by visiting the vertices according to that numbering.

Theorem J.6. *Given a weighted graph that obeys the triangle inequality, the Christofides heuristic computes a (3/2)-approximation of the shortest traveling salesman tour.*

A minimum spanning tree T, a minimum-cost perfect matching M of the odd vertices in T, an Eulerian circuit of $T \cup M$, and the resulting approximate traveling salesman tour.

Proof: Let A denote the length of the tour computed by the Christofides heuristic; let OPT denote the length of the optimal tour; let MST denote the total length of the minimum spanning tree; let MOM denote the total length of the minimum odd-vertex matching.

The graph $T \cup M$, and therefore any Euler tour of $T \cup M$, has total length $MST + MOM$. By the triangle inequality, taking a shortcut past a previously visited vertex can only shorten the tour. Thus, $A \leq MST + MOM$.

By the triangle inequality, the optimal tour of the odd-degree vertices of T cannot be longer than OPT. Any cycle passing through of the odd vertices can be partitioned into two perfect matchings, by alternately coloring the edges of the cycle red and green. One of these two matchings has length at most $OPT/2$. On the other hand, both matchings have length at least MOM. Thus, $MOM \leq OPT/2$.

Finally, recall our earlier observation that $MST \leq OPT$.

Putting these three inequalities together, we conclude that $A \leq 3 \cdot OPT/2$, as claimed. □

Four decades after its discovery, Christofedes' algorithm is the best approximation algorithm known for metric TSP, although better approximation algorithms are known for interesting special cases. In particular, there have been several recent breakthroughs for the special case where the underlying metric is determined by shortest-path distances in an *unweighted* graph G; this special case is often called the ***graphic*** traveling salesman problem. The best approximation ratio known for this special case is $7/5$, from a 2012 algorithm of Sebő and Vygen.

J.10 k-center Clustering

The *k-center clustering* problem is defined as follows. We are given a set $P = \{p_1, p_2, \ldots, p_n\}$ of n points in the plane[3] and an integer k. Our goal to find a collection of k circles that collectively enclose all the input points, such that the radius of the largest circle is as

[3]The k-center problem can be defined over *any* metric space, and the approximation analysis in this section holds in any metric space as well. The analysis in the *next* section, however, does require that the points come from the Euclidean plane.

small as possible. More formally, we want to compute a set $C = \{c_1, c_2, \ldots, c_k\}$ of k *center points*, such that the following cost function is minimized:

$$cost(C) := \max_i \min_j |p_i\, c_j|.$$

Here, $|p_i\, c_j|$ denotes the Euclidean distance between input point p_i and center point c_j. Intuitively, each input point is assigned to its closest center point; the points assigned to a given center c_j comprise a *cluster*. The distance from c_j to the farthest point in its cluster is the *radius* of that cluster; the cluster is contained in a circle of this radius centered at c_j. The k-center clustering cost $cost(C)$ is precisely the maximum cluster radius.

This problem turns out to be NP-hard, even to approximate within a factor of roughly 1.8. However, there is a natural greedy strategy, first analyzed in 1985 by Teofilo Gonzalez[4], that is guaranteed to produce a clustering whose cost is at most twice optimal. Choose the k center points one at a time, starting with an arbitrary input point as the first center. In each iteration, choose the input point that is farthest from any earlier center point to be the next center point.

The first six iterations of Gonzalez's k-center clustering algorithm.

In the pseudocode below, d_i denotes the current distance from point p_i to its nearest center, and r_j denotes the maximum of all d_i (or in other words, the cluster radius) after the first j centers have been chosen. The algorithm includes an extra iteration to compute the final clustering radius r_k (and the next center c_{k+1}).

[4]Teofilo F. Gonzalez. Clustering to minimize the maximum inter-cluster distance. *Theoretical Computer Science* 38:293-306, 1985.

```
GonzalezKCenter(P, k):
  for i ← 1 to n
      d_i ← ∞
  c_1 ← p_1
  for j ← 1 to k
      r_j ← 0
      for i ← 1 to n
          d_i ← min{d_i, |p_i c_j|}
          if r_j < d_i
              r_j ← d_i;  c_{j+1} ← p_i
  return {c_1, c_2, ..., c_k}
```

GonzalezKCenter clearly runs in $O(nk)$ time. Using more advanced data structures, Tomas Feder and Daniel Greene[5] described an algorithm to compute exactly the same clustering in only $O(n \log k)$ time.

Theorem J.7. *GonzalezKCenter computes a 2-approximation to the optimal k-center clustering.*

Proof: Let OPT denote the optimal k-center clustering radius for P. For any index i, let c_i and r_i denote the ith center point and ith clustering radius computed by GonzalezKCenter.

By construction, each center point c_j has distance at least r_{j-1} from any center point c_i with $i < j$. Moreover, for any $i < j$, we have $r_i \geq r_j$. Thus, $|c_i c_j| \geq r_k$ for all indices i and j.

On the other hand, at least one cluster in the optimal clustering contains at least two of the points $c_1, c_2, \ldots, c_{k+1}$. Thus, by the triangle inequality, we must have $|c_i c_j| \leq 2 \cdot OPT$ for some indices i and j. We conclude that $r_k \leq 2 \cdot OPT$, as claimed. □

♥J.11 Approximation Schemes

With just a little more work, we can compute an arbitrarily close approximation of the optimal k-clustering, using a so-called ***approximation scheme***. An approximation scheme accepts both an instance of the problem and a value $\varepsilon > 0$ as input, and it computes a $(1 + \varepsilon)$-approximation of the optimal output for that instance. As I mentioned earlier, computing even a 1.8-approximation is NP-hard, so we cannot expect our approximation scheme to run in polynomial time; nevertheless, at least for small

[5]Tomas Feder* and Daniel H. Greene. Optimal algorithms for approximate clustering. *Proc. 20th STOC*, 1988. Unlike Gonzalez's algorithm, Feder and Greene's faster algorithm does not work over arbitrary metric spaces; it requires that the input points come from some $\mathbb{R}^d$ and that distances are measured in some L_p metric. The time analysis also assumes that the distance between any two points can be computed in $O(1)$ time.

values of k, the approximation scheme will be considerably more efficient than any exact algorithm.

Our approximation scheme works in three phases:

1. Compute a 2-approximate clustering of the input set P using GONZALEZKCENTER. Let r be the cost of this clustering.
2. Create a regular grid of squares of width $\delta = \varepsilon r/2\sqrt{2}$. Let Q be a subset of P containing one point from each non-empty cell of this grid.
3. Compute an *optimal* set of k centers for Q. Return these k centers as the approximate k-center clustering for P.

The first phase requires $O(nk)$ time. By our earlier analysis, we have $r^* \leq r \leq 2r^*$, where r^* is the optimal k-center clustering cost for P.

The second phase can be implemented in $O(n)$ time using a hash table, or in $O(n \log n)$ time by standard sorting, by associating approximate coordinates $(\lfloor x/\delta \rfloor, \lfloor y/\delta \rfloor)$ to each point $(x, y) \in P$ and removing duplicates. The key observation is that the resulting point set Q is significantly smaller than P. We know P can be covered by k balls of radius r^*, each of which touches $O(r^*/\delta^2) = O(1/\varepsilon^2)$ grid cells. It follows that $|Q| = O(k/\varepsilon^2)$.

Let $T(n, k)$ be the running time of an *exact* k-center clustering algorithm, given n points as input. If this were a computational geometry class, we might see a "brute force" algorithm that runs in time $T(n, k) = O(n^{k+2})$; the fastest algorithm currently known[6] runs in time $T(n, k) = n^{O(\sqrt{k})}$. If we use this algorithm, our third phase requires $(k/\varepsilon^2)^{O(\sqrt{k})}$ time.

It remains to show that the optimal clustering for Q implies a $(1 + \varepsilon)$-approximation of the optimal clustering for P. Suppose the optimal clustering of Q consists of k balls $B_1, B_2, \ldots, B_k$, each of radius $\tilde{r}$. Clearly $\tilde{r} \leq r^*$, since any set of k balls that cover P also cover any subset of P. Each point in $P \setminus Q$ shares a grid cell with some point in Q, and therefore is within distance $\delta\sqrt{2}$ of some point in Q. Thus, if we increase the radius of each ball B_i by $\delta\sqrt{2}$, the expanded balls must contain every point in P. We conclude that the optimal centers for Q gives us a k-center clustering for P of cost at most $r^* + \delta\sqrt{2} \leq r^* + \varepsilon r/2 \leq r^* + \varepsilon r^* = (1 + \varepsilon)r^*$.

The total running time of the approximation scheme is $O(nk + (k/\varepsilon^2)^{O(\sqrt{k})})$. This is still exponential in the input size if k is large (say $\sqrt{n}$ or $n/100$), but if k and ε are fixed constants, the running time is linear in the number of input points.

♥J.12 An FPTAS for Subset Sum

An approximation scheme whose running time, for any fixed ε, is polynomial in n is called a ***polynomial-time approximation scheme*** or ***PTAS*** (usually pronounced *"pee*

[6] R. Z. Hwang, R. C. T. Lee, and R. C. Chan. The slab dividing approach to solve the Euclidean p-center problem. *Algorithmica* 9(1):1–22, 1993.

taz"). If in addition the running time depends only polynomially on ε, the algorithm is called a ***fully*** polynomial-time approximation scheme or *FPTAS* (usually pronounced "*eff pee taz*"). For example, an approximation scheme with running time $O(n^2/\varepsilon^2)$ is an FPTAS; an approximation scheme with running time $O(n^{1/\varepsilon^6})$ is a PTAS but not an FPTAS; and our approximation scheme for k-center clustering is not a PTAS.

The last problem we'll consider is the SubsetSum problem: Given a set X containing n positive integers and a target integer t, determine whether X has a subset whose elements sum to t. The lecture notes on NP-completeness include a proof that SubsetSum is NP-hard. As stated, this problem doesn't allow any sort of approximation—the answer is either True or False.[7] So we will consider a related optimization problem instead: Given set X and integer t, find the subset of X whose sum is as large as possible but no larger than t.

We have already seen a dynamic programming algorithm to solve the decision version SubsetSum in time $O(nt)$; a similar algorithm solves the optimization version in the same time bound. Here is a different algorithm, whose running time does not depend on t:

```
SubsetSum(X[1..n], t):
  S_0 ← {0}
  for i ← 1 to n
      S_i ← S_{i−1} ∪ (S_{i−1} + X[i])
      remove all elements of S_i bigger than t
  return max S_n
```

Here $S_{i-1} + X[i]$ denotes the set $\{s + X[i] \mid s \in S_{i-1}\}$. If we store each S_i in a sorted array, the ith iteration of the for-loop requires time $O(|S_{i-1}|)$. Each set S_i contains all possible subset sums for the first i elements of X; thus, S_i has at most 2^i elements. On the other hand, since every element of S_i is an integer between 0 and t, we also have $|S_i| \le t + 1$. It follows that the total running time of this algorithm is $\sum_{i=1}^{n} O(|S_{i-1}|) = O(\min\{2^n, nt\})$.

Of course, this is only an estimate of worst-case behavior. If several subsets of X have the same sum, the sets S_i will have fewer elements, and the algorithm will be faster. The key idea for finding an approximate solution quickly is to 'merge' nearby elements of S_i—if two subset sums are nearly equal, ignore one of them. On the one hand, merging similar subset sums will introduce some error into the output, but hopefully not too much. On the other hand, by reducing the size of the set of sums we need to maintain, we will make the algorithm faster, hopefully significantly so.

Here is our approximation algorithm. We make only two changes to the exact algorithm: an initial sorting phase and an extra Filtering step inside the main loop.

[7] Do, or do not. There is no 'try'. (Are old one thousand when years you, alphabetical also in order talk will you.)

```
FILTER(Z[1..k], δ):
  SORT(Z)
  j ← 1
  Y[j] ← Z[i]
  for i ← 2 to k
      if Z[i] > (1 + δ) · Y[j]
          j ← j + 1
          Y[j] ← Z[i]
  return Y[1..j]
```

```
APPROXSUBSETSUM(X[1..n], k, ε):
  SORT(X)
  R_0 ← {0}
  for i ← 1 to n
      R_i ← R_{i−1} ∪ (R_{i−1} + X[i])
      R_i ← FILTER(R_i, ε/2n)
      remove all elements of R_i bigger than t
  return max R_n
```

Theorem J.8. *ApproxSubsetSum returns a $(1+\varepsilon)$-approximation of the optimal subset sum, given any ε such that $0 < \varepsilon \leq 1$.*

Proof: The theorem follows from the following claim, which we prove by induction:

For any element $s \in S_i$, there is an element $r \in R_i$ such that $r \leq s \leq r \cdot (1 + \varepsilon n/2)^i$.

The claim is trivial for $i = 0$. Let s be an arbitrary element of S_i, for some $i > 0$. There are two cases to consider: either $x \in S_{i-1}$, or $x \in S_{i-1} + x_i$.

(1) Suppose $s \in S_{i-1}$. By the inductive hypothesis, there is an element $r' \in R_{i-1}$ such that $r' \leq s \leq r' \cdot (1 + \varepsilon n/2)^{i-1}$. If $r' \in R_i$, the claim obviously holds. On the other hand, if $r' \notin R_i$, there must be an element $r \in R_i$ such that $r < r' \leq r(1 + \varepsilon n/2)$, which implies that

$$r < r' \leq s \leq r' \cdot (1 + \varepsilon n/2)^{i-1} \leq r \cdot (1 + \varepsilon n/2)^i,$$

so the claim holds.

(2) Suppose $s \in S_{i-1} + x_i$. By the inductive hypothesis, there is an element $r' \in R_{i-1}$ such that $r' \leq s - x_i \leq r' \cdot (1 + \varepsilon n/2)^{i-1}$. If $r' + x_i \in R_i$, the claim obviously holds. On the other hand, if $r' + x_i \notin R_i$, there must be an element $r \in R_i$ such that $r < r' + x_i \leq r(1 + \varepsilon n/2)$, which implies that

$$\begin{aligned} r < r' + x_i \leq s &\leq r' \cdot (1 + \varepsilon n/2)^{i-1} + x_i \\ &\leq (r - x_i) \cdot (1 + \varepsilon n/2)^i + x_i \\ &\leq r \cdot (1 + \varepsilon n/2)^i - x_i \cdot ((1 + \varepsilon n/2)^i - 1) \\ &\leq r \cdot (1 + \varepsilon n/2)^i. \end{aligned}$$

so the claim holds.

Now let $s^* = \max S_n$ and $r^* = \max R_n$. Clearly $r^* \leq s^*$, since $R_n \subseteq S_n$. Our claim implies that there is some $r \in R_n$ such that $s^* \leq r \cdot (1 + \varepsilon/2n)^n$. But r cannot be bigger than r^*, so $s^* \leq r^* \cdot (1 + \varepsilon/2n)^n$. The inequalities $e^x \geq 1 + x$ for all x, and $e^x \leq 2x + 1$ for all $0 \leq x \leq 1$, imply that $(1 + \varepsilon/2n)^n \leq e^{\varepsilon/2} \leq 1 + \varepsilon$. □

Theorem J.9. *ApproxSubsetSum runs in $O((n^3 \log n)/\varepsilon)$ time.*

Proof: Assuming we keep each set R_i in a sorted array, we can merge the two sorted arrays R_{i-1} and $R_{i-1} + x_i$ in $O(|R_{i-1}|)$ time. Filterin R_i and removing elements larger than t also requires only $O(|R_{i-1}|)$ time. Thus, the overall running time of our algorithm is $O(\sum_i |R_i|)$; to express this in terms of n and ε, we need to prove an upper bound on the size of each set R_i.

Let $\delta = \varepsilon/2n$. Because we consider the elements of X in increasing order, every element of R_i is between 0 and $i \cdot x_i$. In particular, every element of $R_{i-1} + x_i$ is between x_i and $i \cdot x_i$. After Filtering, at most one element $r \in R_i$ lies in the range $(1+\delta)^k \le r < (1+\delta)^{k+1}$, for any k. Thus, at most $\lceil \log_{1+\delta} i \rceil$ elements of $R_{i-1} + x_i$ survive the call to Filter. It follows that

$$\begin{aligned}
|R_i| &= |R_{i-1}| + \left\lceil \frac{\log i}{\log(1+\delta)} \right\rceil \\
&\le |R_{i-1}| + \left\lceil \frac{\log n}{\log(1+\delta)} \right\rceil && [i \le n] \\
&\le |R_{i-1}| + \left\lceil \frac{2 \ln n}{\delta} \right\rceil && [e^x \le 1 + 2x \text{ for all } 0 \le x \le 1] \\
&\le |R_{i-1}| + \left\lceil \frac{n \ln n}{\varepsilon} \right\rceil && [\delta = \varepsilon/2n]
\end{aligned}$$

Unrolling this recurrence into a summation gives us the upper bound $|R_i| \le i \cdot \lceil (n \ln n)/\varepsilon \rceil = O((n^2 \log n)/\varepsilon)$.

We conclude that the overall running time of ApproxSubsetSum is $O((n^3 \log n)/\varepsilon)$, as claimed. □

Exercises

1. (a) Prove that for any set of jobs, the makespan of the greedy assignment is at most $(2-1/m)$ times the makespan of the optimal assignment, where m is the number of machines.
   (b) Describe a set of jobs such that the makespan of the greedy assignment is exactly $(2-1/m)$ times the makespan of the optimal assignment, where m is the number of machines.
   (c) Describe an efficient algorithm to solve the minimum makespan scheduling problem *exactly* if every processing time $T[i]$ is a power of two.

2. (a) Find the smallest graph (minimum number of edges) for which GreedyVertexCover does not return the smallest vertex cover.
   (b) For any integer n, describe an n-vertex graph for which GreedyVertexCover returns a vertex cover of size $OPT \cdot \Omega(\log n)$.

3. (a) Find the smallest graph (minimum number of edges) for which DUMBVERTEXCOVER does not return the smallest vertex cover.
   (b) Describe an infinite family of graphs for which DUMBVERTEXCOVER returns a vertex cover of size $2 \cdot OPT$.

4. Let R be a set of rectangles in the plane, with horizontal and vertical edges. A *stabbing set* for R is a set S of points such that every rectangle in R contains at least one point in S. The *rectangle stabbing problem* asks for the smallest stabbing set of a given set of rectangles.
   (a) Prove that the rectangle stabbing problem is NP-hard.
   (b) Describe and analyze an efficient approximation algorithm for the rectangle stabbing problem. What is the approximation ratio of your algorithm?

5. Consider the following heuristic for constructing a vertex cover of a connected graph G: return the set of non-leaf nodes in any depth-first spanning tree of G.
   (a) Prove that this heuristic returns a vertex cover of G.
   (b) Prove that this heuristic returns a 2-approximation to the minimum vertex cover of G.
   (c) Describe an infinite family of graphs for which this heuristic returns a vertex cover of size $2 \cdot OPT$.

6. Suppose we have n pieces of candy with weights $W[1..n]$ (in ounces) that we want to load into boxes. Our goal is to load the candy into as many boxes as possible, so that each box contains at least L ounces of candy. Describe an efficient 2-approximation algorithm for this problem. Prove that the approximation ratio of your algorithm is 2. *[Hint: First consider the case where every piece of candy weighs less than L ounces.]*

7. Suppose we want to route a set of N calls on a telecommunications network that consists of a cycle of n nodes, indexed in cyclic order from 0 to $n-1$. Each call can be routed either clockwise or counterclockwise around the cycle from its source node to its destination node. Our goal is to route the calls so as to minimize the overall load on the network. The load L_i on any edge $(i, (i+1) \mod n)$ is the number of calls routed through that edge, and the overall load is $\max_i L_i$. Describe and analyze an efficient 2-approximation algorithm for this problem.

8. The *linear arrangement* problem asks, given an n-vertex directed graph as input, for an ordering $v_1, v_2, \ldots, v_n$ of the vertices that maximizes the number of forward edges: directed edges $v_i \to v_j$ such that $i < j$. Describe and analyze an efficient 2-approximation algorithm for this problem. (Solving this problem exactly is NP-hard.)

9. A *three-dimensional matching* in an undirected graph G is a collection of vertex-disjoint triangles (cycles of length 3) in G. A three-dimensional matching is *maximal* if it is not a proper subgraph of a larger three-dimensional matching in the same graph.

   (a) Let M and M' be two arbitrary maximal three-dimensional matchings in the same underlying graph G. Prove that $|M| \leq 3 \cdot |M'|$.

   (b) Finding the *largest* three-dimensional matching in a given graph is NP-hard. Describe and analyze a fast 3-approximation algorithm for this problem.

   (c) Finding the *smallest maximal* three-dimensional matching in a given graph is NP-hard. Describe and analyze a fast 3-approximation algorithm for this problem.

10. Consider the following optimization version of the PARTITION problem. Given a set X of positive integers, our task is to partition X into disjoint subsets A and B such that $\max\left\{\sum A, \sum B\right\}$ is as small as possible. Solving this problem exactly is clearly NP-hard.

    (a) Prove that the following algorithm yields a (3/2)-approximation.

```
GreedyPartition(X[1..n]):
  a ← 0
  b ← 0
  for i ← 1 to n
      if a < b
          a ← a + X[i]
      else
          b ← b + X[i]
  return max{a, b}
```

    (b) Prove that the approximation ratio 3/2 cannot be improved, even if the input array X is sorted in *non-decreasing* order. In other words, give an example of a sorted array X where the output of GreedyPartition is 50% larger than the cost of the optimal partition.

    (c) Prove that if the array X is sorted in *non-increasing* order, then GreedyPartition achieves an approximation ratio of at most 4/3.

    ♥(d) Prove that if the array X is sorted in *non-increasing* order, then GreedyPartition achieves an approximation ratio of exactly 7/6.

11. The *chromatic number* $\chi(G)$ of a graph G is the minimum number of colors required to color the vertices of the graph, so that every edge has endpoints with different colors. Computing the chromatic number exactly is NP-hard.

    Prove that the following problem is also NP-hard: Given an n-vertex graph G, return any integer between $\chi(G)$ and $\chi(G)+31337$. *[Note: This does not contradict the possibility of a constant **factor** approximation algorithm.]*

12. Let $G = (V, E)$ be an undirected graph, each of whose vertices is colored either red, green, or blue. An edge in G is *boring* if its endpoints have the same color, and *interesting* if its endpoints have different colors. The *most interesting 3-coloring* is the 3-coloring with the maximum number of interesting edges, or equivalently, with the fewest boring edges. Computing the most interesting 3-coloring is NP-hard, because the standard 3-coloring problem is a special case.

    (a) Let $zzz(G)$ denote the number of boring edges in the most interesting 3-coloring of a graph G. Prove that it is NP-hard to approximate $zzz(G)$ within a factor of $10^{10^{100}}$.

    (b) Let $wow(G)$ denote the number of interesting edges in the most interesting 3-coloring of G. Suppose we assign each vertex in G a *random* color from the set $\{\text{red}, \text{green}, \text{blue}\}$. Prove that the expected number of interesting edges is at least $\frac{2}{3}wow(G)$.

    (c) Prove that with high probability, the expected number of interesting edges is at least $\frac{1}{2}wow(G)$. *[Hint: Use Chebyshev's inequality. But wait... How do we know that we* ***can*** *use Chebyshev's inequality?]*

13. The KNAPSACK problem can be defined as follows. We are given a finite set X, each of whose elements has a non-negative *size* and a non-negative *value*, along with an integer *capacity* c. Our task is to determine the maximum total value among all subsets of X whose total size is at most c. This problem is NP-hard. Specifically, the optimization version of SUBSETSUM is a special case, where each element's value is equal to its size.

    Determine the approximation ratio of the following polynomial-time approximation algorithm. Prove your answer is correct.

```
ApproxKnapsack(X, c):
  return max{GreedyKnapsack(X, c), PickBestOne(X, c)}
```

```
GreedyKnapsack(X, c):
  Sort X in decreasing order by the ratio value/size
  S ← 0;  V ← 0
  for i ← 1 to n
      if S + size(x_i) > c
          return V
      S ← S + size(x_i)
      V ← V + value(x_i)
  return V
```

```
PickBestOne(X, c):
  Sort X in increasing order by size
  V ← 0
  for i ← 1 to n
      if size(x_i) > c
          return V
      if value(x_i) > V
          V ← value(x_i)
  return V
```

14. In the *bin packing* problem, we are given a set of n items, each with weight between 0 and 1, and we are asked to load the items into as few bins as possible, such that the total weight in each bin is at most 1. It's not hard to show that this problem is

NP-hard; this question asks you to analyze a few common approximation algorithms. In each case, the input is an array $W[1..n]$ of weights, and the output is the number of bins used.

```
NextFit(W[1..n]):
  b ← 0
  Total[0] ← ∞

  for i ← 1 to n
      if Total[b] + W[i] > 1
          b ← b + 1
          Total[b] ← W[i]
      else
          Total[b] ← Total[b] + W[i]

  return b
```

```
FirstFit(W[1..n]):
  b ← 0

  for i ← 1 to n
      j ← 1; found ← False

      while j ≤ b and found = False
          if Total[j] + W[i] ≤ 1
              Total[j] ← Total[j] + W[i]
              found ← True
          j ← j + 1

      if found = False
          b ← b + 1
          Total[b] = W[i]

  return b
```

(a) Prove that NextFit uses at most twice the optimal number of bins.

(b) Prove that FirstFit uses at most twice the optimal number of bins.

♥(c) Prove that if the weight array W is initially sorted in decreasing order, then FirstFit uses at most $(4 \cdot OPT + 1)/3$ bins, where OPT is the optimal number of bins. The following facts may be useful (but you need to prove them if your proof uses them):

- In the packing computed by FirstFit, every item with weight more than $1/3$ is placed in one of the first OPT bins.
- FirstFit places at most $OPT - 1$ items outside the first OPT bins.

15. Given a graph G with edge weights and an integer k, suppose we wish to partition the vertices of G into k subsets $S_1, S_2, \ldots, S_k$ so that the sum of the weights of the edges that cross the partition (that is, have endpoints in different subsets) is as large as possible.

(a) Describe an efficient $(1 - 1/k)$-approximation algorithm for this problem.

(b) Now suppose we wish to minimize the sum of the weights of edges that do *not* cross the partition. What approximation ratio does your algorithm from part (a) achieve for the new problem? Justify your answer.

16. The lecture notes describe a $(3/2)$-approximation algorithm for the metric traveling salesman problem, which asks for the minimum-cost Hamiltonian cycle in a graph whose edge weights satisfy the triangle inequality. Here, we consider computing minimum-cost Hamiltonian *paths*. Our input consists of a graph G whose edges have

weights that satisfy the triangle inequality. Depending upon the problem, we are also given zero, one, or two endpoints.

(a) If our input includes zero endpoints, describe a (3/2)-approximation to the problem of computing a minimum cost Hamiltonian path.

(b) If our input includes one endpoint u, describe a (3/2)-approximation to the problem of computing a minimum cost Hamiltonian path that starts at u.

♥(c) If our input includes two endpoints u and v, describe a (5/3)-approximation to the problem of computing a minimum cost Hamiltonian path that starts at u and ends at v.

17. Suppose we are given a collection of n jobs to execute on a machine containing a row of p identical processors. The ***parallel scheduling problem*** asks us to schedule these jobs on these processors, given two arrays $T[1..n]$ and $P[1..n]$ as input, subject to the following constraints:

- When the ith job is executed, it occupies a contiguous interval of $P[i]$ processors for exactly $T[i]$ seconds.
- No processor works on more than one job at a time.

A valid schedule specifies a non-negative starting time and an interval of processors for each job that meets these constraints. Our goal is to compute a valid schedule with the smallest possible *makespan*, which is the earliest time when all jobs are complete.

(a) Prove that the parallel scheduling problem is NP-hard.

(b) Describe a polynomial-time algorithm that computes a 3-approximation of the minimum makespan of a given set of jobs. That is, if the minimum makespan is M, your algorithm should compute a schedule with make-span at most $3M$. You may assume that p is a power of 2. *[Hint: p is a power of 2.]*

(c) Describe an algorithm that computes a 3-approximation of the minimum makespan of a given set of jobs in $O(n \log n)$ time. Again, you may assume that n is a power of 2.

18. Consider the greedy algorithm for the metric traveling salesman problem: start at an arbitrary vertex u, and at each step, travel to the closest unvisited vertex.

(a) Prove that the approximation ratio for this algorithm is $O(\log n)$, where n is the number of vertices. *[Hint: Argue that the kth least expensive edge in the tour output by the greedy algorithm has weight at most* $\mathrm{OPT}/(n-k+1)$*; try $k=1$ and $k=2$ first.]*

♥(b) Prove that the approximation ratio for this algorithm is $\Omega(\log n)$. That is, describe an infinite family of weighted graphs such that the greedy algorithm returns a

Hamiltonian cycle whose weight is $\Omega(\log n)$ times the weight of the optimal TSP tour.

Models of Computation

Jeff Erickson

December 28, 2018

http://jeffe.cs.illinois.edu/teaching/algorithms/ • http://algorithms.wtf/

© Copyright 2014–2017 Jeff Erickson. Last update December 28, 2018.

This work may be freely copied and distributed in any medium.
It may not be sold for more than the actual cost of reproduction, storage, or transmittal.

This work is available under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.
For license details, see http://creativecommons.org/licenses/by-nc-sa/4.0/.

For the most recent edition, see http://jeffe.cs.illinois.edu/teaching/algorithms/.

I'm writing a book.
I've got the page numbers done,
so now I just have to fill in the rest.

— Stephen Wright

About These Notes

These are lecture notes that I wrote for the course "Algorithms and Models of Computation" at the University of Illinois, Urbana-Champaign for the first time in Fall 2014, and revised in Fall 2016. This course is a broad introduction to theoretical computer science, aimed at third-year computer science and computer engineering majors, that covers both fundamental topics in algorithms, for which I already have copious notes, and fundamental topics on formal languages and automata, for which I wrote the notes you are reading now.

The most recent revision of these notes (or nearly so) is available online at http://jeffe.cs.illinois.edu/teaching/algorithms/ (or at the shorter URLs http://algorithms.fyi/ and http://algorithms.wtf/) along with my algorithms notes and a near-complete archive of past homeworks and exams from all my theoretical computer science classes. I plan to revise and reorganize these whenever I teach this material, so you may find more recent versions on the web page of whatever course I am currently teaching.

About the Exercises

Each note ends with several exercises, many of which I used in homeworks, discussion sections, or exams. *Stars indicate more challenging problems (which I have *not* used in homeworks, discussion sections, or exams). Many of these exercises were contributed by my amazing teaching assistants:

> Alex Steiger, Chao Xu, Charles Carlson, Connor Clark, Gail Steitz, Grant Czajkowski, Hsien-Chih Chang, Junqing Deng, Konstantinos Koiliaris, Nick Bachmair, Spencer Gordon, Tana Wattanawaroon, and Yipu Wang

Please do not ask me for solutions to the exercises. If you are a student, seeing the solution will rob you of the experience of solving the problem yourself, which is the only way to learn the material. If you are an instructor, you shouldn't ask your students to solve problems that you can't solve yourself. (I don't always follow my own advice, so some of the problems are buggy.)

Caveat Lector!

These notes are best viewed as an unfinished first draft. You should assume the notes contain several major errors, in addition to the usual unending supply of typos, fencepost errors, off-by-one errors, and brain farts. Before Fall 2014, I had not taught this material in more than two decades. Moreover, the course itself is still very new—Lenny Pitt and I developed the course and offered the first pilot in Spring 2014 (with Lenny presenting the formal language material)—so even the choice of which material to emphasize, sketch, or exclude is still very much in flux.

I would sincerely appreciate feedback of any kind, especially bug reports.

Thanks, and enjoy!

— *Jeff*

Contents

1 Strings . . . 1
2 Regular Languages . . . 19
3 Finite-State Machines . . . 31
4 Nondeterministic Automata . . . 55
5 Context-Free Languages . . . 77
6 Turing Machines . . . 97
7 Undecidability . . . 117
8 Universal Models . . . 137
9 Nondeterministic Turing Machines . . . 145

Vario, inquit [Epicurus], ordine ac positione conveniunt atomi sicut literae, quae cum sint paucae, varie tamen collocatae innumerabilia verba conficiunt.
[Atoms, like letters, says Epicurus, come together in various orders and positions; there are few of them, but different combinations produce countless words.]

— Gottfried Leibniz, *Dissertatio de Arte Combinatoria* (1666)

THOMAS GODFREY, a self-taught mathematician, great in his way, and afterward inventor of what is now called Hadley's Quadrant. But he knew little out of his way, and was not a pleasing companion; as, like most great mathematicians I have met with, he expected universal precision in everything said, or was forever denying or distinguishing upon trifles, to the disturbance of all conversation. He soon left us.

— Benjamin Franklin, *Memoirs, Part 1* (1771)
describing one of the founding members of the Junto

If indeed, as Hilbert asserted, mathematics is a meaningless game played with meaningless marks on paper, the only mathematical experience to which we can refer is the making of marks on paper.

— Eric Temple Bell, *The Queen of the Sciences* (1931)

1 Strings

Throughout this course, we will discuss dozens of algorithms and computational models that manipulate sequences: one-dimensional arrays, linked lists, blocks of text, walks in graphs, sequences of executed instructions, and so on. Ultimately the input and output of any algorithm must be representable as a finite string of symbols—the raw contents of some contiguous portion of the computer's memory. Reasoning about computation requires reasoning about strings.

This note lists several formal definitions and formal induction proofs related to strings. These definitions and proofs are *intentionally* much more detailed than normally used in practice—most people's intuition about strings is fairly accurate—but the extra precision is necessary for any sort of formal proof. It may be helpful to think of this material as part of the "assembly language" of theoretical computer science. We normally think about computation at a *much* higher level of abstraction, but ultimately every argument must "compile" down to these (and similar) definitions.

But the actual definitions and theorems are *not* the point. The point of playing with this material is to get some experience working with formal/mechanical definitions and proofs, especially inductive definitions and recursive proofs. Or should I say recursive definitions and inductive proofs? Whatever, they're the same thing. Strings are a particularly simple and convenient playground for $\left\{\begin{smallmatrix}\text{induct}\\ \text{recurs}\end{smallmatrix}\right\}$ion; we'll see *many* more examples throughout the course. When you read this note, don't just look at the *content* of the definitions and proofs; pay close attention to their *structure* and the *process* for creating them.

1.1 Strings

Fix an arbitrary finite set Σ called the ***alphabet***; the individual elements of Σ are called ***symbols*** or ***characters***. As a notational convention, I will always use lower-case letters near the start of the English alphabet ($a, b, c, \ldots$) as symbol variables, and *never* as explicit symbols. For explicit symbols, I will always use fixed-width upper-case letters (`A`, `B`, `C`, ...), digits (`0`, `1`, `2`, ...),

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

or other symbols (⋄, $, #, •, ...) that are clearly distinguishable from variables. For further emphasis, I will almost always typeset explicit symbols in RED.

A ***string*** (or ***word***) over Σ is a finite sequence of zero or more symbols from Σ. Formally, a string w over Σ is defined recursively as either

- the empty string, denoted by the Greek letter $\boldsymbol{\varepsilon}$ (epsilon), or
- an ordered pair $\boldsymbol{(a, x)}$, where a is a symbol in Σ and x is a string over Σ.

We normally write either $\boldsymbol{a \cdot x}$ or simply $\boldsymbol{ax}$ to denote the ordered pair (a, x). Similarly, we normally write explicit strings as sequences of symbols instead of nested ordered pairs; for example, STRING is convenient shorthand for the formal expression (S, (T, (R, (I, (N, (G, ε)))))). As a notational convention, I will always use lower-case letters near the end of the English alphabet $(\ldots, w, x, y, z)$ for string variables, and SHOUTY⋄RED⋄MONOSPACED⋄TEXT to typeset explicit (non-empty) strings.

The set of all strings over Σ is denoted $\boldsymbol{\Sigma^*}$ (pronounced "sigma star"). It is very important to remember that every element of Σ^* is a *finite* string, although Σ^* itself is an infinite set containing strings of every possible *finite* length.

1.2 Two Recursive Functions

Our first several proofs about strings will involve two natural functions, one giving the length of a string, the other gluing two strings together into a larger string. These two function behave exactly the way you think they do, but if we actually want to *prove* anything about them, we first have to *define* them in a way that supports formal proofs. Because the objects on which these functions act—strings–are defined recursively, the functions must also be defined recursively.

The ***length*** $\boldsymbol{|w|}$ of a string w is the number of symbols in w, defined formally as follows:

$$|w| := \begin{cases} 0 & \text{if } w = \varepsilon, \\ 1 + |x| & \text{if } w = ax. \end{cases}$$

For example, the string FIFTEEN has length 7, the string SEVEN has length 5, and the string 5 has length 1. Although they are formally different objects, we do not normally distinguish between symbols and strings of length 1.

The ***concatenation*** of two strings x and y, denoted either $\boldsymbol{x \bullet y}$ or simply $\boldsymbol{xy}$, is the unique string containing the characters of x in order, followed by the characters in y in order. For example, the string NOWHERE is the concatenation of the strings NOW and HERE; that is, NOW • HERE = NOWHERE. (On the other hand, HERE • NOW = HERENOW.) Formally, concatenation is defined recusively as follows:

$$w \bullet z := \begin{cases} z & \text{if } w = \varepsilon, \\ a \cdot (x \bullet z) & \text{if } w = ax. \end{cases}$$

(Here I'm using a larger dot • to formally distinguish the operator that concatenates two arbitrary strings from from the syntactic sugar · that builds a string from a single character and a string.)

When we describe the concatenation of more than two strings, we normally omit all dots and parentheses, writing $wxyz$ instead of $(w \bullet (x \bullet y)) \bullet z$, for example. This simplification is justified by the fact (which we will prove shortly) that the function • is associative.

1.3 Induction on Strings

Induction is *the* standard technique for proving statements about recursively defined objects. Hopefully you are already comfortable proving statements about *natural numbers* via induction, but induction is actually a far more general technique. Several different variants of induction can be used to prove statements about more general structures; here I describe the variant that I recommend (and actually use in practice). This variant follows two primary design considerations:

- ***The case structure of the proof should mirror the case structure of the recursive definition.*** For example, if you are proving something about all *strings*, your proof should have two cases: Either $w = \varepsilon$, or $w = ax$ for some symbol a and string x.
- ***The inductive hypothesis should be as strong as possible.*** The (strong) inductive hypothesis for statements about natural numbers is *always* "Assume there is no counterexample k such that $k < n$." I recommend adopting a similar inductive hypothesis for strings: "Assume there is no counterexample x such that $|x| < |w|$." Then for the case $w = ax$, we have $|x| = |w| - 1 < |w|$ by definition of $|w|$, so the inductive hypothesis applies to x.

Thus, string-induction proofs have the following boilerplate structure. Suppose we want to prove that every string is perfectly cromulent, whatever that means. The white boxes hide additional proof details that, among other things, depend on the precise definition of "perfectly cromulent".

Proof: Let w be an arbitrary string.
Assume, for every string x such that $|x| < |w|$, that x is perfectly cromulent.
There are two cases to consider.

- Suppose $w = \varepsilon$.

  Therefore, w is perfectly cromulent.
- Suppose $w = ax$ for some symbol a and string x.
  The induction hypothesis implies that x is perfectly cromulent.

  Therefore, w is perfectly cromulent.

In both cases, we conclude that w is perfectly cromulent. □

Here are three canonical examples of this proof structure. When developing proofs in this style, I strongly recommend first ***mindlessly*** writing the green text (the boilerplate) with lots of space for each case, then filling in the red text (the actual theorem and the induction hypothesis), and only then starting to actually think.

Many students are confused (or at least bored and distracted) by the fact that we are proving *mind-bogglingly* obvious facts. If you're one of these students, try to remember that ***the lemmas themselves are not the point.*** Pay close attention to the *structure* of the proofs. Notice how each proof follows the boilerplate described above. Notice how every sentence of the proof follows *mechanically* from earlier sentences, definitions, and the rules of standard logic and arithmetic.

Lemma 1.1. *Adding nothing does nothing: For every string w, we have $w \bullet \varepsilon = w$.*

Proof: Let w be an arbitrary string. Assume that $x \bullet \varepsilon = x$ for every string x such that $|x| < |w|$. There are two cases to consider:

- Suppose $w = \varepsilon$.

$$\begin{aligned} w \bullet \varepsilon &= \varepsilon \bullet \varepsilon && \text{because } w = \varepsilon, \\ &= \varepsilon && \text{by definition of concatenation,} \\ &= w && \text{because } w = \varepsilon. \end{aligned}$$

- Suppose $w = ax$ for some symbol a and string x.

$$\begin{aligned} w \bullet \varepsilon &= (a \cdot x) \bullet \varepsilon && \text{because } w = ax, \\ &= a \cdot (x \bullet \varepsilon) && \text{by definition of concatenation,} \\ &= a \cdot x && \text{by the inductive hypothesis,} \\ &= w && \text{because } w = ax. \end{aligned}$$

In both cases, we conclude that $w \bullet \varepsilon = w$. □

Lemma 1.2. *Concatenation adds length:* $|w \bullet x| = |w| + |x|$ *for all strings* w *and* x.

Proof: Let w and x be arbitrary strings. Assume that $|y \bullet x| = |y| + |x|$ for every string y such that $|y| < |w|$. (Notice that we are using induction only on w, not on x.) There are two cases to consider:

- Suppose $w = \varepsilon$.

$$\begin{aligned} |w \bullet x| &= |\varepsilon \bullet x| && \text{because } w = \varepsilon \\ &= |x| && \text{by definition of } \bullet \\ &= |\varepsilon| + |x| && |\varepsilon| = 0 \text{ by definition of } |\cdot| \\ &= |w| + |x| && \text{because } w = \varepsilon \end{aligned}$$

- Suppose $w = ay$ for some symbol a and string y.

$$\begin{aligned} |w \bullet x| &= |ay \bullet x| && \text{because } w = ay \\ &= |a \cdot (y \bullet x)| && \text{by definition of } \bullet \\ &= 1 + |y \bullet x| && \text{by definition of } |\cdot| \\ &= 1 + |y| + |x| && \text{by the inductive hypothesis} \\ &= |ay| + |x| && \text{by definition of } |\cdot| \\ &= |w| + |x| && \text{because } w = ay \end{aligned}$$

In both cases, we conclude that $|w \bullet x| = |w| + |x|$. □

Lemma 1.3. *Concatenation is* associative*:* $(w \bullet x) \bullet y = w \bullet (x \bullet y)$ *for all strings* w*,* x*, and* y.

Proof: Let w, x, and y be arbitrary strings. Assume that $(z \bullet x) \bullet y = z \bullet (x \bullet y)$ for every string z such that $|z| < |w|$. (Again, we are using induction only on w.) There are two cases to consider.

- Suppose $w = \varepsilon$.

$$\begin{aligned}
(w \bullet x) \bullet y &= (\varepsilon \bullet x) \bullet y && \text{because } w = \varepsilon \\
&= x \bullet y && \text{by definition of } \bullet \\
&= \varepsilon \bullet (x \bullet y) && \text{by definition of } \bullet \\
&= w \bullet (x \bullet y) && \text{because } w = \varepsilon
\end{aligned}$$

- Suppose $w = az$ for some symbol a and some string z.

$$\begin{aligned}
(w \bullet x) \bullet y &= (az \bullet x) \bullet y && \text{because } w = az \\
&= (a \cdot (z \bullet x)) \bullet y && \text{by definition of } \bullet \\
&= a \cdot ((z \bullet x) \bullet y) && \text{by definition of } \bullet \\
&= a \cdot (z \bullet (x \bullet y)) && \text{by the inductive hypothesis} \\
&= az \bullet (x \bullet y) && \text{by definition of } \bullet \\
&= w \bullet (x \bullet y) && \text{because } w = az
\end{aligned}$$

In both cases, we conclude that $(w \bullet x) \bullet y = w \bullet (x \bullet y)$. □

1.4 More Than One Path up the Mountain

This is not the *only* boilerplate that one can use for induction proofs on strings. For example, we can model our case analysis on the following observation, whose easy proof we leave as an exercise (hint, hint): A string $w \in \Sigma^*$ is ***non-empty*** if and only if either

- $w = a$ for some symbol $a \in \Sigma$, or
- $w = x \bullet y$ for some *non-empty* strings x and y.

In the latter case, Lemma 1.2 implies that $|x| < |w|$ and $|y| < |w|$, so in an inductive proof, we can apply the inductive hypothesis to either x or y (or even both).

Here is a proof of Lemma 1.3 that uses this alternative recursive structure:

Proof: Let w, x, and y be arbitrary strings. Assume that $(z \bullet x') \bullet y' = z \bullet (x' \bullet y')$ for all strings x', y', and z such that $|z| < |w|$. (We need a stronger induction hypothesis here than in the previous proofs!) There are ***three*** cases to consider.

- Suppose $w = \varepsilon$.

$$\begin{aligned}
(w \bullet x) \bullet y &= (\varepsilon \bullet x) \bullet y && \text{because } w = \varepsilon \\
&= x \bullet y && \text{by definition of } \bullet \\
&= \varepsilon \bullet (x \bullet y) && \text{by definition of } \bullet \\
&= w \bullet (x \bullet y) && \text{because } w = \varepsilon
\end{aligned}$$

- Suppose w is equal to some symbol a.

$$\begin{aligned}
(w \bullet x) \bullet y &= (a \bullet x) \bullet y && \text{because } w = a \\
&= (a \cdot x) \bullet y && \text{because } a \bullet z = a \cdot z \text{ by definition of } \bullet \\
&= a \cdot (x \bullet y) && \text{by definition of } \bullet \\
&= a \bullet (x \bullet y) && \text{because } a \bullet z = a \cdot z \text{ by definition of } \bullet \\
&= w \bullet (x \bullet y) && \text{because } w = a
\end{aligned}$$

- Suppose $w = u \bullet v$ for some nonempty strings u and v.

$$\begin{aligned}
(w \bullet x) \bullet y &= ((u \bullet v) \bullet x) \bullet y && \text{because } w = u \bullet v \\
&= (u \bullet (v \bullet x)) \bullet y && \text{by the inductive hypothesis, because } |u| < |w| \\
&= u \bullet ((v \bullet x) \bullet y) && \text{by the inductive hypothesis, because } |u| < |w| \\
&= u \bullet (v \bullet (x \bullet y)) && \text{by the inductive hypothesis, because } |v| < |w| \\
&= (u \bullet v) \bullet (x \bullet y) && \text{by the inductive hypothesis, because } |u| < |w| \\
&= w \bullet (x \bullet y) && \text{because } w = u \bullet v
\end{aligned}$$

In ***all three*** cases, we conclude that $(w \bullet x) \bullet y = w \bullet (x \bullet y)$. □

1.5 Indices, Substrings, and Subsequences

Finally, I'll conclude this note by formally defining several other common functions and terms related to strings.

For any string w and any integer $1 \le i \le |w|$, the expression $\boldsymbol{w_i}$ denotes the ith symbol in w, counting from left to right. More formally, w_i is recursively defined as follows:

$$w_i := \begin{cases} a & \text{if } w = ax \text{ and } i = 1, \\ x_{i-1} & \text{if } w = ax \text{ and } i > 1. \end{cases}$$

As one might reasonably expect, w_i is formally undefined if $i < 1$ or $w = \varepsilon$, and therefore (by induction) if $i > |w|$. The integer i is called the ***index*** of w_i.

We sometimes write strings as a concatenation of their constituent symbols using this subscript notation: $w = w_1 w_2 \cdots w_{|w|}$. While completely standard, this notation is slightly misleading, because it *incorrectly* suggests that the string w contains at least three symbols, when in fact w could be a single symbol or even the empty string.

In actual code, subscripts are usually expressed using the bracket notation $\boldsymbol{w[i]}$. Brackets were introduced as a typographical convention over a hundred years ago because subscripts and superscripts[1] were difficult or impossible to type.[2] We sometimes write strings as explicit arrays $w[1..n]$, with the understanding that $n = |w|$. Again, this notation is potentially misleading; always remember that n might be zero; the string/array could be empty.

A ***substring*** of a string w is another string obtained from w by deleting zero or more symbols from the beginning and from the end. Formally, a string y is a substring of w if and only if there are strings x and z such that $w = xyz$. Extending the array notation for strings, we write $\boldsymbol{w[i..j]}$ to denote the substring of w starting at w_i and ending at w_j. More formally, we define

$$w[i..j] := \begin{cases} \varepsilon & \text{if } j < i, \\ w_i \cdot w[i+1..j] & \text{otherwise.} \end{cases}$$

[1] The same bracket notation is also used for bibliographic references, instead of the traditional footnote/endnote superscripts, for exactly the same reasons.

[2] A ***typewriter*** is an obsolete mechanical device loosely resembling a computer keyboard. Pressing a key on a typewriter moves a lever (called a "typebar") that strikes a cloth ribbon full of ink against a piece of paper, leaving the image of a single character. Many historians believe that the ordering of letters on modern keyboards (`QWERTYUIOP`) evolved in the late 1800s, reaching its modern form on the 1874 Sholes & Glidden Type-Writer™, in part to separate many common letter pairs, to prevent typebars from jamming against each other; this is also why the keys on most modern keyboards are arranged in a slanted grid. (The common folk theory that the ordering was deliberately intended to slow down typists doesn't withstand careful scrutiny.) A more recent theory suggests that the ordering was influenced by telegraph[3] operators, who found older alphabetic arrangements confusing, in part because of ambiguities in American Morse Code.

A ***proper substring*** of w is any substring other than w itself. For example, LAUGH is a proper substring of SLAUGHTER. Whenever y is a (proper) substring of w, we also call w a (proper) ***superstring*** of y.

A ***prefix*** of $w[1..n]$ is any substring of the form $w[1..j]$. Equivalently, a string p is a ***prefix*** of another string w if and only if there is a string x such that $px = w$. A ***proper prefix*** of w is any prefix except w itself. For example, DIE is a proper prefix of DIET.

Similarly, a suffix of $w[1..n]$ is any substring of the form $w[i..n]$. Equivalently, a string s is a ***suffix*** of a string w if and only if there is a string x such that $xs = w$. A ***proper suffix*** of w is any suffix except w itself. For example, YES is a proper suffix of EYES, and HE is both a proper prefix and a proper suffix of HEADACHE.

A ***subsequence*** of a string w is a string obtained by deleting zero or more symbols from *anywhere* in w. More formally, z is a subsequence of w if and only if

- $z = \varepsilon$, or
- $w = ax$ for some symbol a and some string x such that z is a subsequence of x.
- $w = ax$ and $z = ay$ for some symbol a and some strings x and y, and y is a subsequence of x.

A ***proper subsequence*** of w is any subsequence of w other than w itself. Whenever z is a (proper) subsequence of w, we also call w a (proper) ***supersequence*** of z.

Substrings and subsequences are not the same objects; don't confuse them! Every substring of w is also a subsequence of w, but not every subsequence is a substring. For example, METAL is a subsequence, but not a substring, of MEATBALL. To emphasize the distinction, we sometimes redundantly refer to substrings of w as ***contiguous*** substrings, meaning all their symbols appear together in w.

Exercises

Most of the following exercises ask for proofs of various claims about strings. Here "prove" means give a complete, self-contained, formal proof by inductive definition-chasing, using the boilerplate structure recommended in Section 1.3. Feel free to use Lemmas 1.1, 1.2, and 1.3 without proof, but don't assume any other facts about strings that you have not actually proved. (Some later exercises rely on results proved in earlier exercises.) Do not appeal to intuition, and do not use the words "obvious" or "clearly" or "just". Most of these claims *are* in fact obvious; the real exercise is understanding and formally expressing *why* they're obvious.

[3] A ***telegraph*** is an obsolete electromechanical communication device consisting of an electrical circuit with a switch at one end and an electromagnet at the other. The sending operator would press and release a key, closing and opening the circuit, originally causing the electromagnet to push a stylus onto a moving paper tape, leaving marks that could be decoded by the receiving operator. (Operators quickly discovered that they could directly decode the clicking sounds made by the electromagnet, and so the paper tape became obsolete almost immediately.) The most common scheme within the US to encode symbols, developed by Alfred Vail and Samuel Morse in 1837, used (mostly) short ($\cdot$) and long ($-$) marks—now called "dots" and "dashes", or "dits" and "dahs"—separated by gaps of various lengths. American Morse code (as it became known) was ambiguous; for example, the letter Z and the string SE were both encoded by the sequence $\cdots\cdot$ ("di-di-dit, dit"). This ambiguity has been blamed for the S key's position on the typewriter keyboard near E and Z.

Vail and Morse were of course not the first people to propose encoding symbols as strings of bits. That honor apparently falls to Francis Bacon, who devised a five-bit binary encoding of the alphabet (except for the letters J and U) in 1605 as the basis for a steganographic code—a method or hiding secret message in otherwise normal text.

Note to instructors: ***Do not assign any of these problems before solving them yourself, especially on exams.*** It's *very* easy to underestimate the difficulty of these problems, or at least the lengths of their solutions, which for exams is a reasonable proxy for difficulty. Also, several later exercises rely implicitly on identities like $\#(a, x \bullet y) = \#(a, x) + \#(a, y)$ that are only proved in earlier exercises. It's unfair to assign these problems to students without telling them which earlier results they can use.

Useful Facts About Strings

1. Let w be an arbitrary string, and let $n = |w|$. Prove each of the following statements.

   (a) w has exactly $n+1$ prefixes.

   (b) w has exactly n proper suffixes.

   (c) w has at most $n(n+1)/2$ distinct substrings. (Why "at most"?)

   (d) w has at most $2^n - 1$ distinct proper subsequences. (Why "at most"?)

2. Prove the following useful identities for all strings w, x, y, and z directly from the definition of $\bullet$, *without* referring to the length of any string.

   (a) If $x \bullet y = x$, then $y = \varepsilon$.

   (b) If $x \bullet y = y$, then $x = \varepsilon$.

   (c) If $x \bullet z = y \bullet z$, then $x = y$.

   (d) If $x \bullet y = x \bullet z$, then $y = z$.

3. Prove the following useful fact about substrings. An arbitrary string x is a substring of another arbitrary string $w = u \bullet v$ if and only if at least one of the following conditions holds:

   - x is a substring of u.
   - x is a substring of v.
   - $x = yz$ where y is a suffix of u and z is a prefix of v.

4. Let w be an arbitrary string, and let $n = |w|$. Prove the following statements for all indices $1 \le i \le j \le k \le n$.

   (a) $|w[i..j]| = j - i + 1$

   (b) $w[i..j] \bullet w[j+1..k] = w[i..k]$

   (c) $w^R[i..j] = (w[i'..j'])^R$ where $i' + j = j' + i = |w| + 1$.

Recursive Functions

5. For any symbol a and any string w, let $\#(a, w)$ denote the number of occurrences of a in w. For example, $\#(\texttt{A}, \texttt{BANANA}) = 3$ and $\#(\texttt{X}, \texttt{FLIBBERTIGIBBET}) = 0$.

   (a) Give a formal recursive definition of the function $\# : \Sigma \times \Sigma^* \to \mathbb{N}$.

(b) Prove that $\#(a, xy) = \#(a, x) + \#(a, y)$ for every symbol a and all strings x and y. Your proof must rely on both your answer to part (a) and the formal recursive definition of string concatenation.

(c) Prove the following identity for all alphabets Σ and all strings $w \in \Sigma^*$:

$$|w| = \sum_{a \in \Sigma} \#(a, w)$$

[Hint: Don't try to use induction on Σ.]

6. The ***reversal*** $\boldsymbol{w^R}$ of a string w is defined recursively as follows:

$$w^R := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ x^R \bullet a & \text{if } w = a \cdot x \end{cases}$$

(a) Prove that $(w^R)^R = w$ for every string w.

(b) Prove that $|w^R| = |w|$ for every string w.

(c) Prove that $(w \bullet x)^R = x^R \bullet w^R$ for all strings w and x.

(d) Prove that $\#(a, w^R) = \#(a, w)$ for every string w and every symbol a. (See Exercise 5.)

7. For any string w and any non-negative integer n, let $\boldsymbol{w^n}$ denote the string obtained by concatenating n copies of w; more formally, we define

$$w^n := \begin{cases} \varepsilon & \text{if } n = 0 \\ w \bullet w^{n-1} & \text{otherwise} \end{cases}$$

For example, $(\texttt{BLAH})^5 = \texttt{BLAHBLAHBLAHBLAHBLAH}$ and $\varepsilon^{374} = \varepsilon$.

(a) Prove that $w^m \bullet w^n = w^{m+n}$ for every string w and all non-negative integers n and m.

(b) Prove that $\#(a, w^n) = n \cdot \#(a, w)$ for every string w, every symbol a, and every non-negative integer n. (See Exercise 5.)

(c) Prove that $(w^R)^n = (w^n)^R$ for every string w and every non-negative integer n.

(d) Prove that for all strings x and y that if $x \bullet y = y \bullet x$, then $x = w^m$ and $y = w^n$ for some string w and some non-negative integers m and n. *[Hint: Careful with ε!]*

8. The ***complement*** $\boldsymbol{w^c}$ of a string $w \in \{\texttt{0}, \texttt{1}\}^*$ is obtained from w by replacing every $\texttt{0}$ in w with a $\texttt{1}$ and vice versa. The complement function can be defined recursively as follows:

$$w^c := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ \texttt{1} \cdot x^c & \text{if } w = \texttt{0}x \\ \texttt{0} \cdot x^c & \text{if } w = \texttt{1}x \end{cases}$$

(a) Prove that $|w| = |w^c|$ for every string w.

(b) Prove that $(x \bullet y)^c = x^c \bullet y^c$ for all strings x and y.

(c) Prove that $\#(1, w) = \#(0, w^c)$ for every string w.

(d) Prove that $(w^R)^c = (w^c)^R$ for every string w.

(e) Prove that $(w^n)^c = (w^c)^n$ for every string w and every non-negative integer n.

9. For any string $w \in \{0, 1, 2\}^*$, let $\boldsymbol{w}^+$ denote the string obtained from w by replacing each symbol a in w by the symbol corresponding to $(a + 1) \bmod 3$. for example, $0102101^+ = 1210212$. This function can be defined more formally as follows:

$$w^+ := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ 1 \cdot x^+ & \text{if } w = 0x \\ 2 \cdot x^+ & \text{if } w = 1x \\ 0 \cdot x^+ & \text{if } w = 2x \end{cases}$$

(a) Prove that $|w| = |w^+|$ for every string $w \in \{0, 1, 2\}^*$.

(b) Prove that $(x \bullet y)^+ = x^+ \bullet y^+$ for all strings $x, y \in \{0, 1, 2\}^*$.

(c) Prove that $\#(1, w^+) = \#(0, w)$ for every string $w \in \{0, 1, 2\}^*$.

(d) Prove that $(w^+)^R = (w^R)^+$ for every string $w \in \{0, 1, 2\}^*$.

10. For any string $w \in \{0, 1\}^*$, let $swap(w)$ denote the string obtained from w by swapping the first and second symbols, the third and fourth symbols, and so on. For example:

$$swap(101) = 011$$
$$swap(100111) = 011011$$
$$swap(10\,11\,00\,01\,10\,1) = 01\,11\,00\,10\,01\,1.$$

The *swap* function can be formally defined as follows:

$$swap(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ w & \text{if } w = 0 \text{ or } w = 1 \\ ba \bullet swap(x) & \text{if } w = abx \text{ for some } a, b \in \{0, 1\} \text{ and } x \in \{0, 1\}^* \end{cases}$$

(a) Prove that $|swap(w)| = |w|$ for every string w.

(b) Prove that $swap(swap(w)) = w$ for every string w.

(c) Prove that $swap(w^R) = (swap(w))^R$ for every string w such that $|w|$ is even. *[Hint: Your proof must invoke* ***four*** *different recursive definitions: reversal w^R, concatenation $\bullet$, length $|w|$, and the swap function!]*

11. For any string $w \in \{0, 1\}^*$, let $sort(w)$ denote the string obtained by sorting the characters in w. For example, $sort(010101) = 000111$. The sort function can be defined recursively as follows:

$$sort(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ 0 \cdot sort(x) & \text{if } w = 0x \\ sort(x) \bullet 1 & \text{if } w = 1x \end{cases}$$

Prove the following for all strings $w, x, y \in \{0, 1\}^*$:

(a) Prove that $sort(w) \in \mathtt{0}^*\mathtt{1}^*$ for every string $w \in \{\mathtt{0}, \mathtt{1}\}^*$.

(b) Prove that $\#(\mathtt{0}, w) = \#(\mathtt{0}, sort(w))$ for every string $w \in \{\mathtt{0}, \mathtt{1}\}^*$.

(c) Prove that $|w| = |sort(w)|$, for every string $w \in \{\mathtt{0}, \mathtt{1}\}^*$.

(d) Prove that $sort(w) = \mathtt{0}^{\#(\mathtt{0},w)}\mathtt{1}^{\#(\mathtt{1},w)}$, for every string $w \in \{\mathtt{0}, \mathtt{1}\}^*$. (In other words, prove that our recursive definition is correct.)

(e) Prove that $sort(w^R) = sort(w)$, for every string $w \in \{\mathtt{0}, \mathtt{1}\}^*$.

12. Consider the following recursively defined function:

$$merge(x, y) := \begin{cases} y & \text{if } x = \varepsilon \\ x & \text{if } y = \varepsilon \\ \mathtt{0} \cdot merge(w, y) & \text{if } x = \mathtt{0}w \\ \mathtt{0} \cdot merge(x, z) & \text{if } y = \mathtt{0}z \\ \mathtt{1} \cdot merge(w, y) & \text{if } x = \mathtt{1}w \text{ and } y = \mathtt{1}z \end{cases}$$

For example:

$$merge(\mathtt{10}, \mathtt{10}) = \mathtt{1010}$$
$$merge(\mathtt{10}, \mathtt{010}) = \mathtt{01010}$$
$$merge(\mathtt{010}, \mathtt{0001100}) = \mathtt{0000101100}$$

(a) Prove that $merge(x, y) \in \mathtt{0}^*\mathtt{1}^*$ for all strings $x, y \in \mathtt{0}^*\mathtt{1}^*$. (The regular expression $\mathtt{0}^*\mathtt{1}^*$ is shorthand for the language $\{\mathtt{0}^a\mathtt{1}^b \mid a, b \geq 0\}$.)

(b) Prove that $sort(x \bullet y) = merge(sort(x), sort(y))$ for all strings $x, y \in \{\mathtt{0}, \mathtt{1}\}^*$. (The *sort* function is defined in the previous exercise.)

13. Consider the following pair of mutually recursive functions on strings:

$$evens(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ odds(x) & \text{if } w = ax \end{cases} \qquad odds(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ a \cdot evens(x) & \text{if } w = ax \end{cases}$$

For example, $evens(\mathtt{MISSISSIPPI}) = \mathtt{ISSIP}$ and $odds(\mathtt{MISSISSIPPI}) = \mathtt{MSISPI}$.

(a) Prove the following identity for all strings w and x:

$$evens(w \bullet x) = \begin{cases} evens(w) \bullet evens(x) & \text{if } |w| \text{ is even,} \\ evens(w) \bullet odds(x) & \text{if } |w| \text{ is odd.} \end{cases}$$

(b) State and prove a similar identity for $odds(w \bullet x)$.

(c) Prove that every string w is a shuffle of $evens(w)$ and $odds(w)$.

14. Consider the following recursively defined function:

$$stutter(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ aa \bullet stutter(x) & \text{if } w = ax \end{cases}$$

For example, $stutter(\mathtt{MISSISSIPPI}) = \mathtt{MMIISSSSIISSSSIIPPPPII}$.

(a) Prove that $|stutter(w)| = 2|w|$ for every string w.

(b) Prove that $evens(stutter(w)) = w$ for every string w.

(c) Prove that $odds(stutter(w)) = w$ for every string w.

(d) Prove that $stutter(w)$ is a shuffle of w and w, for every string w.

(e) Prove that w is a palindrome if and only if $stutter(w)$ is a palindrome, for every string w.

15. Consider the following recursive function:

$$faro(w, z) := \begin{cases} z & \text{if } w = \varepsilon \\ a \cdot faro(z, x) & \text{if } w = ax \end{cases}$$

For example, $faro(\texttt{0011}, \texttt{0101}) = \texttt{00011011}$. (A "faro shuffle" splits a deck of cards into two equal piles and then perfectly interleaves them.)

(a) Prove that $|faro(x, y)| = |x| + |y|$ for all strings x and y.

(b) Prove that $faro(w, w) = stutter(w)$ for every string w.

(c) Prove that $faro(odds(w), evens(w)) = w$ for every string w.

16. For any string w, let $declutter(w)$ denote the string obtained from w by deleting any symbol that equals its immediate successor. For example, $declutter(\texttt{MISSISSIPPI}) = \texttt{MISISIPI}$, and $declutter(\texttt{ABBCCCAAAACCCBBA}) = \texttt{ABCACBA}$.

(a) Given a recursive definition for the function *declutter*.

(b) Using your recursive definition, prove that $declutter(stutter(w)) = declutter(w)$ for every string w.

(c) Using your recursive definition, prove that $declutter(w^R) = (declutter(w))^R$ for every string w.

(d) Using your recursive definition, prove that w is a palindrome if and only if $declutter(w)$ is a palindrome, for every string w.

17. Consider the following recursively defined function

$$hanoi(w) = \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ hanoi(x) \bullet a \bullet hanoi(x) & \text{if } w = ax \end{cases}$$

Prove that $|hanoi(w)| = 2^{|w|} - 1$ for every string w.

18. Consider the following recursively defined function

$$slog(w) = \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ a \cdot slog(evens(w)) & \text{if } w = ax \end{cases}$$

Prove that $|slog(w)| = \lceil \log_2(|w| + 1) \rceil$ for every string w.

19. Consider the following recursively defined function

$$bitrev(w) = \begin{cases} w & \text{if } |w| \leq 1 \\ bitrev(odds(w)) \bullet bitrev(evens(w)) & \text{otherwise} \end{cases}$$

(a) Prove that $|bitrev(w)| = |w|$ for every string w.

*(b) Prove that $bitrev(bitrev(w)) = w$ for every string w such that $|w|$ is a power of 2.

20. The ***binary value*** of any string $w \in \{0, 1\}^*$ is the integer whose binary representation (possibly with leading 0s) is w. The value function can be defined recursively as follows:

$$value(w) := \begin{cases} 0 & \text{if } w = \varepsilon \\ 2 \cdot value(x) & \text{if } w = x \bullet 0 \\ 2 \cdot value(x) + 1 & \text{if } w = x \bullet 1 \end{cases}$$

(a) Prove that $value(w) + value(w^c) = 2^{|w|} - 1$ for every string $w \in \{0, 1\}^*$.

(b) Prove that $value(x \bullet y) = value(x) \cdot 2^{|y|} + value(y)$ for all strings $x, y \in \{0, 1\}^*$.

*(c) Prove that $value(x)$ is divisible by 3 if and only if $value(x^R)$ is divisible by 3.

Recursively Defined Sets

20. Recursively define a set L of strings over the alphabet $\{0, 1\}$ as follows:

- The empty string ε is in L.
- For any two strings x and y in L, the string $0x1y0$ is also in L.
- These are the only strings in L.

(a) Prove that the string 000010101010010100 is in L.

(b) Prove by induction that every string in L has exactly twice as many 0s as 1s. (You may assume the identity $\#(a, xy) = \#(a, x) + \#(a, y)$ for any symbol a and any strings x and y; see Exercise 5(b).)

(c) Give an example of a string with exactly twice as many 0s as 1s that is *not* in L.

21. Recursively define a set L of strings over the alphabet $\{0, 1\}$ as follows:

- The empty string ε is in L.
- For any two strings x and y in L, the string $0x1y$ is also in L.
- For any two strings x and y in L, the string $1x0y$ is also in L.
- These are the only strings in L.

(a) Prove that the string 01000110111001 is in L.

(b) Prove by induction that every string in L has exactly the same number of 0s and 1s. (You may assume the identity $\#(a, xy) = \#(a, x) + \#(a, y)$ for any symbol a and any strings x and y; see Exercise 5(b).)

(c) Prove by induction that L contains every string with the same number of 0s and 1s.

22. Recursively define a set L of strings over the alphabet $\{0, 1\}$ as follows:

- The empty string ε is in L.
- For any strings x in L, the strings $0x1$ and $1x0$ are also in L.
- For any two strings x and y in L, the string $x \bullet y$ is also in L.
- These are the only strings in L.

(a) Prove that the string 01000110111001 is in L.

(b) Prove by induction that every string in L has exactly the same number of 0s and 1s. (You may assume the identity $\#(a, xy) = \#(a, x) + \#(a, y)$ for any symbol a and any strings x and y; see Exercise 5(b).)

(c) Prove by induction that every string with the same number of 0s and 1s is in L.

23. Recursively define a set L of strings over the alphabet $\{0, 1, 2\}$ as follows:

- The empty string ε is in L.
- For any string x in L, the string $0x$ is also in L.
- For any strings x and y in L, the strings $1x2y$ and $2x1y$ are also in L.
- These are the only strings in L.

(a) Prove that the string 001201110220121220 is in L.

(b) For any string $w \in \{0, 1, 2\}^*$, let ***ModThree(w)*** denote the sum of the digits of w modulo 3. This function can be defined recursively as follows:

$$ModThree(w) = \begin{cases} 0 & \text{if } w = \varepsilon \\ ModThree(x) & \text{if } w = 0x \\ (ModThree(x) + 1) \bmod 3 & \text{if } w = 1x \\ (ModThree(x) + 2) \bmod 3 & \text{if } w = 2x \end{cases}$$

For example, $ModThree(2211) = 0$ and $ModThree(001201110220121220) = 0$. Prove that $ModThree(w) = 0$ for every string in $w \in L$.

(c) Find a string $w \in \{0, 1, 2\}^*$ such that $ModThree(w) = 0$ but $w \notin L$. Prove that your answer is correct.

(d) Prove that $\#(1, w) = \#(2, w)$ for every string $w \in L$.

(e) Find a string $w \in \{0, 1, 2\}^*$ such that $\#(1, w) = \#(2, w)$ but $w \notin L$. Prove that your answer is correct.

(f) Prove that $L = \left\{ w \in \{0, 1, 2\}^* \;\middle|\; ModThree(w) = 0 \text{ and } \#(1, w) = \#(2, w) \right\}$.

24. A ***palindrome*** is a string that is equal to its reversal.

(a) Give a recursive definition of a palindrome over the alphabet Σ.

(b) Prove that any string p meets your recursive definition of a palindrome if and only if $p = p^R$.

(c) Using your recursive definition, prove that the strings $w \bullet w^R$ and $w \bullet a \bullet w^R$ are palindromes, for every string w and symbol a.

(d) Using your recursive definition, prove that p^n is a palindrome for every palindrome p and every natural number n. (See Exercise 7.)

(e) Using your recursive definition, prove that for every palindrome p, there is at most one symbol a such that $\#(a, p)$ is odd. (See Exercise 5.)

25. A string $w \in \Sigma^*$ is called a ***shuffle*** of two strings $x, y \in \Sigma^*$ if at least one of the following recursive conditions is satisfied:

- $w = x = y = \varepsilon$.
- $w = aw'$ and $x = ax'$ and w' is a shuffle of x' and y, for some $a \in \Sigma$ and some $w', x' \in \Sigma^*$.
- $w = aw'$ and $y = ay'$ and w' is a shuffle of x and y', for some $a \in \Sigma$ and some $w', y' \in \Sigma^*$.

For example, the string BANANANANANASA is a shuffle of the strings BANANA and ANANAS.

(a) Prove that if w is a shuffle of x and y, then $|w| = |x| + |y|$.

(b) Prove that w is a shuffle of x and y if and only if w^R is a shuffle of x^R and y^R.

26. For any positive integer n, the ***Fibonacci string F_n*** is defined recursively as follows:

$$F_n = \begin{cases} \texttt{0} & \text{if } n = 1, \\ \texttt{1} & \text{if } n = 2, \\ F_{n-2} \bullet F_{n-1} & \text{otherwise.} \end{cases}$$

For example, $F_6 = \texttt{10101101}$ and $F_7 = \texttt{0110110101101}$.

(a) Prove that for every integer $n \geq 2$, the string F_n can also be obtained from F_{n-1} by replacing every occurrence of 0 with 1 and replacing every occurrence of 1 with 01. More formally, prove that $F_n = \mathit{Finc}(F_{n-1})$, where

$$\mathit{Finc}(w) = \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ \texttt{1} \cdot \mathit{Finc}(x) & \text{if } w = \texttt{0}x \\ \texttt{01} \bullet \mathit{Finc}(x) & \text{if } w = \texttt{1}x \end{cases}$$

[Hint: First prove that $\mathit{Finc}(x \bullet y) = \mathit{Finc}(x) \bullet \mathit{Finc}(y)$.]

(b) Prove that the Fibonacci string F_n begins with 1 if and only if n is even.

(c) Prove that 00 is not a substring of any Fibonacci string F_n.

(d) Prove that 111 is not a substring of any Fibonacci string F_n.

*(e) Prove that 01010 is not a substring of any Fibonacci string F_n.

*(f) Find another string w that is not a substring of any Fibonacci string F_n, such that 00 and 111 and 01010 are not substrings of w.

★(g) Find a set of strings F with the following properties:

- No string in F is a substring of any Fibonacci string F_n.
- No string in F is a proper substring of any other string in F.
- For all strings $x \in \{0, 1\}^*$, if x has no substrings in F, then x is a substring of some Fibonacci string F_n.

★(h) Prove that the reversal of each Fibonacci string is a substring of another Fibonacci string. More formally, prove that for every integer $n \geq 0$, the string F_n^R is a substring of F_m for some integer $m \geq n$.

*27. Prove that the following three properties of strings are in fact identical.

- A string $w \in \{0, 1\}^*$ is ***balanced*** if it satisfies one of the following conditions:
  - $w = \varepsilon$,
  - $w = 0x1$ for some balanced string x, or
  - $w = xy$ for some balanced strings x and y.
- A string $w \in \{0, 1\}^*$ is ***erasable*** if it satisfies one of the following conditions:
  - $w = \varepsilon$, or
  - $w = x01y$ for some strings x and y such that xy is erasable. (The strings x and y are not necessarily erasable.)
- A string $w \in \{0, 1\}^*$ is ***conservative*** if it satisfies ***both*** of the following conditions:
  - w has an equal number of 0s and 1s, and
  - no prefix of w has more 0s than 1s.

(a) Prove that every balanced string is erasable.

(b) Prove that every erasable string is conservative.

(c) Prove that every conservative string is balanced.

*[Hint: To develop intuition, it may be helpful to think of 0s as left brackets and 1s as right brackets, but **don't** invoke this intuition in your proofs.]*

*28. A string $w \in \{0, 1\}^*$ is ***equitable*** if it has an equal number of 0s and 1s.

(a) Prove that a string w is equitable if and only if it satisfies one of the following conditions:

- $w = \varepsilon$,
- $w = 0x1$ for some equitable string x,
- $w = 1x0$ for some equitable string x, or
- $w = xy$ for some equitable strings x and y.

(b) Prove that a string w is equitable if and only if it satisfies one of the following conditions:

- $w = \varepsilon$,
- $w = x\mathtt{01}y$ for some strings x and y such that xy is equitable, or
- $w = x\mathtt{10}y$ for some strings x and y such that xy is equitable.

In the last two cases, the individual strings x and y are not necessarily equitable.

(c) Prove that a string w is equitable if and only if it satisfies one of the following conditions:

- $w = \varepsilon$,
- $w = xy$ for some balanced string x and some equitable string y, or
- $w = x^R y$ for some for some balanced string x and some equitable string y.

(See the previous exercise for the definition of "balanced".)

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

But the Lord came down to see the city and the tower the people were building. The Lord said, "If as one people speaking the same language they have begun to do this, then nothing they plan to do will be impossible for them. Come, let us go down and confuse their language so they will not understand each other."

— Genesis 11:6–7 (New International Version)

Soyez réglé dans votre vie et ordinaire comme un bourgeois,
afin d'être violent et original dans vos œuvres.
[Be regular and orderly in your life like a bourgeois,
so that you may be violent and original in your work.]

— Gustave Flaubert, in a letter to Gertrude Tennant (December 25, 1876)

Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.

— Jamie Zawinski, alt.religion.emacs (August 12, 1997)

As far as I am aware this pronunciation is incorrect in all known languages.

— Kenneth Kleene, describing his father Stephen's pronunciation of his last name

2 Regular Languages

2.1 Languages

A ***formal language*** (or just a ***language***) is a set of strings over some finite alphabet Σ, or equivalently, an arbitrary subset of Σ^*. For example, each of the following sets is a language:

- The empty set $\varnothing$.[1]
- The set $\{\varepsilon\}$.
- The set $\{0, 1\}^*$.
- The set $\{\texttt{THE}, \texttt{OXFORD}, \texttt{ENGLISH}, \texttt{DICTIONARY}\}$.
- The set of all subsequences of $\texttt{THE}\diamond\texttt{OXFORD}\diamond\texttt{ENGLISH}\diamond\texttt{DICTIONARY}$.
- The set of all words in *The Oxford English Dictionary*.
- The set of all strings in $\{0, 1\}^*$ with an odd number of 1s.
- The set of all strings in $\{0, 1\}^*$ that represent a prime number in base 13.
- The set of all sequences of turns that solve the Rubik's cube (starting in some fixed configuration)
- The set of all python programs that print "Hello World!"

As a notational convention, I will always use italic upper-case letters (usually L, but also A, B, C, and so on) to represent languages.

Formal languages are not "languages" in the same sense that English, Klingon, and Python are "languages". Strings in a formal language do not necessarily carry any "meaning", nor are they necessarily assembled into larger units ("sentences" or "paragraphs" or "packages") according to some "grammar".

[1] The empty set symbol $\varnothing$ was introduced in 1939 by André Weil, as a member of the pseudonymous mathematical collective Nicholai Bourbaki. The symbol derives from the Norwegian letter Ø, which pronounced like a German ö or a sound of disgust, and *not* from the Greek letter ϕ. Calling the empty set "fie" or "fee" makes the baby Jesus cry.

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

It is ***very*** important to distinguish between three "empty" objects. Many beginning students have trouble keeping these straight.

- $\varnothing$ is the empty *language*, which is a set containing zero strings. $\varnothing$ is not a string.
- $\{\varepsilon\}$ is a language containing exactly one string, which has length zero. $\{\varepsilon\}$ is not empty, and it is not a string.
- ε is the empty *string*, which is a sequence of length zero. ε is not a language.

2.2 Building Languages

Languages can be combined and manipulated just like any other sets. Thus, if A and B are languages over Σ, then their union $A \cup B$, intersection $A \cap B$, difference $A \setminus B$, and symmetric difference $A \oplus B$ are also languages over Σ, as is the complement $\overline{A} := \Sigma^* \setminus A$. However, there are two more useful operators that are specific to sets of *strings*.

The ***concatenation*** of two languages A and B, again denoted $\boldsymbol{A \bullet B}$ or just $\boldsymbol{AB}$, is the set of all strings obtained by concatenating an arbitrary string in A with an arbitrary string in B:

$$A \bullet B := \{xy \mid x \in A \text{ and } y \in B\}.$$

For example, if $A = \{\texttt{HOCUS}, \texttt{ABRACA}\}$ and $B = \{\texttt{POCUS}, \texttt{DABRA}\}$, then

$$A \bullet B = \{\texttt{HOCUSPOCUS}, \texttt{ABRACAPOCUS}, \texttt{HOCUSDABRA}, \texttt{ABRACADABRA}\}.$$

In particular, for every language A, we have

$$\varnothing \bullet A = A \bullet \varnothing = \varnothing \qquad \text{and} \qquad \{\varepsilon\} \bullet A = A \bullet \{\varepsilon\} = A.$$

The ***Kleene closure*** or ***Kleene star***[2] of a language L, denoted $\boldsymbol{L^*}$, is the set of all strings obtained by concatenating a sequence of zero or more strings from L. For example, $\{\texttt{0}, \texttt{11}\}^* = \{\varepsilon, \texttt{0}, \texttt{00}, \texttt{11}, \texttt{000}, \texttt{011}, \texttt{110}, \texttt{0000}, \texttt{0011}, \texttt{0110}, \texttt{1100}, \texttt{1111}, \texttt{00000}, \texttt{00011}, \texttt{011110011011}, \ldots\}$. More formally, L^* is defined recursively as the set of all strings w such that either

- $w = \varepsilon$, or
- $w = xy$, for some strings $x \in L$ and $y \in L^*$.

This definition immediately implies that

$$\varnothing^* = \{\varepsilon\}^* = \{\varepsilon\}.$$

For any other language L, the Kleene closure L^* is infinite and contains arbitrarily long (but *finite!*) strings. Equivalently, L^* can also be defined as the smallest superset of L that contains the empty string ε and is closed under concatenation (hence "closure"). The set of all strings Σ^* is, just as the notation suggests, the Kleene closure of the alphabet Σ (where each symbol is viewed as a string of length 1).

A useful variant of the Kleene closure operator is the ***Kleene plus***, defined as $L^+ := L \bullet L^*$. Thus, L^+ is the set of all strings obtained by concatenating a sequence of ***one*** or more strings from L.

The following identities, which we state here without (easy) proofs, are useful for designing, simplifying, and understanding languages.

[2] named after logician Stephen Cole Kleene, who actually pronounced his last name "***clay***-knee", not "clean" or "cleanie" or "claynuh" or "dimaggio".

Lemma 2.1. *The following identities hold for all languages A, B, and C:*

(a) $A \cup B = B \cup A$.

(b) $(A \cup B) \cup C = A \cup (B \cup C)$.

(c) $\varnothing \bullet A = A \bullet \varnothing = \varnothing$.

(d) $\{\varepsilon\} \bullet A = A \bullet \{\varepsilon\} = A$.

(e) $(A \bullet B) \bullet C = A \bullet (B \bullet C)$.

(f) $A \bullet (B \cup C) = (A \bullet B) \cup (A \bullet C)$.

(g) $(A \cup B) \bullet C = (A \bullet C) \cup (B \bullet C)$.

Lemma 2.2. *The following identities hold for every language L:*

(a) $L^* = \{\varepsilon\} \cup L^+ = L^* \bullet L^* = (L \cup \{\varepsilon\})^* = (L \setminus \{\varepsilon\})^* = \{\varepsilon\} \cup L \cup (L^+ \bullet L^+)$.

(b) $L^+ = L \bullet L^* = L^* \bullet L = L^+ \bullet L^* = L^* \bullet L^+ = L \cup (L^+ \bullet L^+)$.

(c) $L^+ = L^*$ *if and only if* $\varepsilon \in L$.

Lemma 2.3 (Arden's Rule). *For any languages A, B, and L such that $L = A \bullet L \cup B$, we have $A^* \bullet B \subseteq L$. Moreover, if A does not contain the empty string, then $L = A \bullet L \cup B$ if and only if $L = A^* \bullet B$.*

2.3 Regular Languages and Regular Expressions

Intuitively, a language is ***regular*** if it can be constructed from individual strings using any combination of union, concatenation, and unbounded repetition. More formally, a language L is regular if and only if it satisfies one of the following (recursive) conditions:

- L is empty;
- L contains exactly one string (which could be the empty string ε);
- L is the union of two regular languages;
- L is the concatenation of two regular languages; or
- L is the Kleene closure of a regular language.

Regular languages are normally described using a compact notation called ***regular expressions***, which omit braces around one-string sets, use $+$ to represent union instead of $\cup$, and juxtapose subexpressions to represent concatenation instead of using an explicit operator $\bullet$. By convention, in the absence of parentheses, the * operator has highest precedence, followed by the (implicit) concatenation operator, followed by $+$.

For example, the regular expression $\mathtt{10}^*$ is shorthand for the language $\{\mathtt{1}\} \bullet \{\mathtt{0}\}^*$ (containing all strings consisting of a $\mathtt{1}$ followed by zero or more $\mathtt{0}$s), and *not* the language $\{\mathtt{10}\}^*$ (containing all strings of even length that start with $\mathtt{1}$ and alternate between $\mathtt{1}$ and $\mathtt{0}$). As a larger example, the regular expression

$$\mathtt{0}^*\mathtt{0} + \mathtt{0}^*\mathtt{1}(\mathtt{10}^*\mathtt{1} + \mathtt{01}^*\mathtt{0})^*\mathtt{10}^*$$

represents the language

$$(\{\mathtt{0}\}^* \bullet \{\mathtt{0}\}) \cup \Big(\{\mathtt{0}\}^* \bullet \{\mathtt{1}\} \bullet \big((\{\mathtt{1}\} \bullet \{\mathtt{0}\}^* \bullet \{\mathtt{1}\}) \cup (\{\mathtt{0}\} \bullet \{\mathtt{1}\}^* \bullet \{\mathtt{0}\})\big)^* \bullet \{\mathtt{1}\} \bullet \{\mathtt{0}\}^*\Big).$$

Most of the time we do not distinguish between regular expressions and the languages they represent, for the same reason that we do not normally distinguish between the arithmetic expression "2+2" and the integer 4, or the symbol π and the area of the unit circle. However, we sometimes need to refer to regular expressions themselves *as strings*. In those circumstances, we write $\boldsymbol{L(R)}$ to denote the language represented by the regular expression R. String w ***matches*** regular expression R if and only if $w \in L(R)$.

Here are several more examples of regular expressions and the languages they represent.

- $\mathtt{0}^*$ — the set of all strings of $\mathtt{0}$s, including the empty string.
- $\mathtt{00000}^*$ — the set of all strings consisting of at least four $\mathtt{0}$s.
- $(\mathtt{00000})^*$ — the set of all strings of $\mathtt{0}$s whose length is a multiple of 5.
- $(\mathtt{0}+\mathtt{1})^*$ — the set of all binary strings.
- $(\varepsilon+\mathtt{1})(\mathtt{01})^*(\varepsilon+\mathtt{0})$ — the set of all strings of alternating $\mathtt{0}$s and $\mathtt{1}$s, or equivalently, the set of all binary strings that do not contain the substrings $\mathtt{00}$ or $\mathtt{11}$.
- $(\mathtt{0}+\mathtt{1})^*\mathtt{0000}(\mathtt{0}+\mathtt{1})^*$ — the set of all binary strings that contain the substring $\mathtt{0000}$.
- $((\varepsilon+\mathtt{0}+\mathtt{00}+\mathtt{000})\mathtt{1})^*(\varepsilon+\mathtt{0}+\mathtt{00}+\mathtt{000})$ — the set of all binary strings that do not contain the substring $\mathtt{0000}$.
- $((\mathtt{0}+\mathtt{1})(\mathtt{0}+\mathtt{1}))^*$ — the set of all binary strings whose length is even.
- $\mathtt{1}^*(\mathtt{01}^*\mathtt{01}^*)^*$ — the set of all binary strings with an even number of $\mathtt{0}$s.
- $\mathtt{0}+\mathtt{1}(\mathtt{0}+\mathtt{1})^*\mathtt{00}$ — the set of all non-negative binary numerals divisible by 4 and with no redundant leading $\mathtt{0}$s.
- $\mathtt{0}^*\mathtt{0}+\mathtt{0}^*\mathtt{1}(\mathtt{10}^*\mathtt{1}+\mathtt{01}^*\mathtt{0})^*\mathtt{10}^*$ — the set of all non-negative binary numerals divisible by 3, possibly with redundant leading $\mathtt{0}$s.

The last example should ***not*** be obvious. It is straightforward, but *really* tedious, to prove by induction that every string in $\mathtt{0}^*\mathtt{0}+\mathtt{0}^*\mathtt{1}(\mathtt{10}^*\mathtt{1}+\mathtt{01}^*\mathtt{0})^*\mathtt{10}^*$ is the binary representation of a non-negative multiple of 3. It is similarly straightforward, but *even more* tedious, to prove that the binary representation of *every* non-negative multiple of 3 matches this regular expression. In a later note, we will see a systematic method for deriving regular expressions for some languages that avoids (or more accurately, *automates*) this tedium.

Two regular expressions R and R' are ***equivalent*** if they describe the same language. For example, the regular expressions $(\mathtt{0}+\mathtt{1})^*$ and $(\mathtt{1}+\mathtt{0})^*$ are equivalent, because the union operator is commutative. More subtly, the regular expressions $(\mathtt{0}+\mathtt{1})^*$ and $(\mathtt{0}^*\mathtt{1}^*)^*$ and $(\mathtt{00}+\mathtt{01}+\mathtt{10}+\mathtt{11})^*(\mathtt{0}+\mathtt{1}+\varepsilon)$ are all equivalent; intuitively, these three expressions represent *different ways of thinking about* the language $\{\mathtt{0},\mathtt{1}\}^*$. In fact, almost every regular language can be represented by infinitely many distinct but equivalent regular expressions, even if we ignore ultimately trivial equivalences like $L = (L\varnothing)^*L\varepsilon+\varnothing$.

★★★ Give some examples of designing regular expressions via Arden's rule. For example: All strings that don't contain the substring $\mathtt{0011}$.

2.4 Things What Ain't Regular Expressions

Many computing environments and programming languages support patterns called ***regexen*** (singular ***regex***, pluralized like *ox*) that are considerably more general and powerful than regular expressions. Regexen include special symbols representing negation, character classes (for example, upper-case letters, or digits), contiguous ranges of characters, line and word boundaries, limited repetition (as opposed to the unlimited repetition allowed by *), back-references to earlier subexpressions, and even local variables. Despite its obvious etymology, a regex is ***not*** necessarily a regular expression, and it does ***not*** necessarily describe a regular language![3]

Another type of pattern that is often confused with regular expression are ***globs***, which are patterns used in most Unix shells and some scripting languages to represent sets file names. Globs include symbols for arbitrary single characters (`?`), single characters from a specified range (`[a-z]`), arbitrary substrings (`*`), and substrings from a specified finite set (`{foo,ba{r,z}}`). Globs are significantly *less* powerful than regular expressions.

2.5 Regular Expression Trees

Regular expressions are convenient notational shorthand for a more explicit representation of regular languages called ***regular expression trees***. A regular expression tree is formally defined as one of the following:

- A leaf node labeled $\varnothing$.
- A leaf node labeled with a string in Σ^*.
- A node labeled $+$ with two children, each of which is the root of a regular expression tree.
- A node labeled $\bullet$ with two children, each of which is the root of a regular expression tree.
- A node labeled $*$ with one child, which is the root of a regular expression tree.

These cases mirror the definition of regular language exactly. A leaf labeled $\varnothing$ represents the empty language; a leaf labeled with a string represents the language containing only that string; a node labeled $+$ represents the union of the languages represented by its two children; a node labeled $\bullet$ represents the concatenation of the languages represented by its two children; and a node labeled $*$ represents the Kleene closure of the languages represented by its child.

A regular expression tree for $0^*0 + 0^*1(10^*1 + 01^*0)^*10^*$

[3]However, regexen are not all-powerful, either; see http://stackoverflow.com/a/1732454/775369.

The ***size*** of a regular expression is the number of nodes in its regular expression tree. The size of a regular expression could be either larger or smaller than its length as a raw string. On the one hand, concatenation nodes in the tree are not represented by symbols in the string; on the other hand, parentheses in the string are not represented by nodes in the tree. For example, the regular expression $0^*0 + 0^*1(10^*1 + 01^*0)^*10^*$ has size 29, but the corresponding raw string `0*0+0*1(10*1+01*0)*10*` has length 22.

A ***subexpression*** of a regular expression R is another regular expression S whose regular expression tree is a subtree of some regular expression tree for R. A ***proper subexpression*** of R is any subexpression except R itself. Every subexpression of R is also a substring of R, but not every substring is a subexpression. For example, the substring 10^*1 is a proper subexpression of $0^*0 + 0^*1(10^*1 + 01^*0)^*10^*$. However, the substrings $0^*0 + 0^*1$ and $0^*1 + 01^*$ are *not* subexpressions of $0^*0 + 0^*1(10^*1 + 01^*0)^*10^*$, even though they are well-formed regular expressions.

2.6 Proofs about Regular Expressions

The standard strategy for proving properties of regular expressions, just as for any other recursively defined structure, to argue inductively on the recursive *structure* of the expression, rather than considering the regular expression as a raw string.

Induction proofs about regular expressions follow a standard boilerplate that mirrors the recursive definition of regular languages. Suppose we want to prove that every regular expression is perfectly cromulent, whatever that means. The white boxes hide additional proof details that, among other things, depend on the precise definition of "perfectly cromulent". The boilerplate structure is *longer* than the boilerplate for string induction proofs, but don't be fooled into thinking it's *harder*. The five cases in the proof mirror the five cases in the recursive definition of regular language.

Proof: Let R be an arbitrary regular expression.
Assume that **every proper subexpression of R** is perfectly cromulent.
There are five cases to consider.

- Suppose $R = \varnothing$.

  Therefore, R is perfectly cromulent.

- Suppose R is a single string.

  Therefore, R is perfectly cromulent.

- Suppose $R = S + T$ for some regular expressions S and T.
  The induction hypothesis implies that S and T are perfectly cromulent.

  Therefore, R is perfectly cromulent.

- Suppose $R = S \bullet T$ for some regular expressions S and T.
  The induction hypothesis implies that S and T are perfectly cromulent.

Therefore, R is perfectly cromulent.

- Suppose $R = S^*$ for some regular expression. S.
  The induction hypothesis implies that S is perfectly cromulent.

  Therefore, R is perfectly cromulent.

In all cases, we conclude that w is perfectly cromulent. □

Students uncomfortable with structural induction can instead induct on the size of the regular expression (defined as the number of nodes in the corresponding regular expression tree). This variant changes only the statement of inductive hypothesis, not the structure of the proof itself; the rest of the boilerplate is utterly identical.

Proof: Let R be an arbitrary regular expression.
Assume that **every regular expression smaller than R** is perfectly cromulent.
There are five cases to consider.

⋮

In all cases, we conclude that w is perfectly cromulent. □

Here is an example of the structural induction boilerplate in action. Again, this proof is *longer* than a typical induction proof about strings or integers, but each individual case is still just a short exercise in definition-chasing.

Lemma 2.4. *Every regular expression that does not use the symbol $\varnothing$ represents a non-empty language.*

Proof: Let R be an arbitrary regular expression that does not use the symbol $\varnothing$. Assume that every proper subexpression of R that does not use the symbol $\varnothing$ represents a non-empty language. There are five cases to consider, mirroring the definition of R.

- If $R = \varnothing$, we have a contradiction; we can ignore this case.
- If R is a single string w, then $L(R)$ contains the string w. (In fact, $L(R) = \{w\}$.)
- Suppose $R = S + T$ for some regular expressions S and T.
  S does not use the symbol $\varnothing$, because otherwise R would.
  Thus, the inductive hypothesis implies that $L(S)$ is non-empty.
  Choose an arbitrary string $x \in L(S)$.
  Then $L(R) = L(S + T) = L(S) \cup L(T)$ also contains the string x.
- Suppose $R = S \bullet T$ for some regular expressions S and T.
  Neither S nor T uses the symbol $\varnothing$, because otherwise R would.
  Thus, the inductive hypothesis implies that both $L(S)$ and $L(T)$ are non-empty.
  Choose arbitrary strings $x \in L(S)$ and $y \in L(T)$.
  Then $L(R) = L(S \bullet T) = L(S) \bullet L(T)$ contains the string xy.
- Suppose $R = S^*$ for some regular expression S.
  Then $L(R)$ contains the empty string ε.

In every case, the language $L(R)$ is non-empty. □

Similarly, most algorithms that accept regular expressions as input actually require regular expression *trees*, rather than regular expressions as raw *strings*. Fortunately, it is possible to ***parse*** any regular expression of length n into an equivalent regular expression tree in $O(n)$ time. (The details of the parsing algorithm are beyond the scope of this chapter.) Thus, when we see an algorithmic problem that starts "Given a regular expression. . . ", we can assume without loss of generality that we are actually given a regular expression *tree*.

2.7 Proofs about Regular Languages

The same boilerplate also applies to inductive arguments about properties of regular *languages*. Languages themselves are just unadorned sets; they don't have any recursive structure that we build an inductive proof around. In particular, proper subsets of an *infinite* language L are not necessarily "smaller" than L! Rather than trying to argue directly about an arbitrary regular language L, we choose an arbitrary regular expression that represents L, and then build our inductive argument around *the recursive structure of that regular expression*.

Lemma 2.5. *Every non-empty regular language is represented by a regular expression that does not use the symbol* $\varnothing$.

Proof: Let R be an arbitrary regular expression; we need to prove that either $L(R) = \varnothing$ or $L(R) = L(R')$ for some $\varnothing$-free regular expression R'. For every proper subexpression S of R, assume that either $L(S) = \varnothing$ or $L(S) = L(S')$ for some $\varnothing$-free regular expression S'. There are five cases to consider, mirroring the definition of R.

- If $R = \varnothing$, then $L(R) = \varnothing$.
- If R is a single string w, then R is already $\varnothing$-free.
- Suppose $R = S + T$ for some regular expressions S and T. There are four subcases to consider:
  - If $L(S) = L(T) = \varnothing$, then $L(R) = L(S) \cup L(T) = \varnothing$.
  - Suppose $L(S) \neq \varnothing$ and $L(T) = \varnothing$. The inductive hypothesis implies that there is a $\varnothing$-free regular expression S' such that $L(S') = L(S) = L(S) \cup L(T) = L(R)$.
  - Suppose $L(S) = \varnothing$ and $L(T) \neq \varnothing$. The inductive hypothesis implies that there is a $\varnothing$-free regular expression T' such that $L(T') = L(T) = L(S) \cup L(T) = L(R)$.
  - Finally, suppose $L(S) \neq \varnothing$ and $L(T) \neq \varnothing$. The inductive hypothesis implies that there are $\varnothing$-free regular expressions S' and T' such that $L(S') = L(S)$ and $L(T') = L(T)$. The regular expression $S' + T'$ is $\varnothing$-free and $L(S' + T') = L(S') \cup L(T') = L(S) \cup L(T) = L(S + T) = L(R)$.
- Suppose $R = S \bullet T$ for some regular expressions S and T. There are two subcases to consider.
  - If either $L(S) = \varnothing$ or $L(T) = \varnothing$ then $L(R) = L(S) \bullet L(T) = \varnothing$.
  - Otherwise, the inductive hypothesis implies that there are $\varnothing$-free regular expressions S' and T' such that $L(S') = L(S)$ and $L(T') = L(T)$. The regular expression $S' \bullet T'$ is $\varnothing$-free and equivalent to R.
- Suppose $R = S^*$ for some regular expression S. There are two subcases to consider.

- If $L(S) = \varnothing$, then $L(R) = \varnothing^* = \{\varepsilon\}$, so R is represented by the $\varnothing$-free regular expression ε.
- Otherwise, The inductive hypothesis implies that there is a $\varnothing$-free regular expression S' such that $L(S') = L(S)$. The regular expression $(S')^*$ is $\varnothing$-free and equivalent to R.

In all cases, either $L(R) = \varnothing$ or R is equivalent to some $\varnothing$-free regular expression R'. □

2.8 Not Every Language is Regular

You may be tempted to conjecture that *all* languages are regular, but in fact, the following cardinality argument *almost all* languages are *not* regular. To make the argument concrete, let's consider languages over the single-symbol alphabet $\{\diamond\}$.

- Every regular expression over the one-symbol alphabet $\{\diamond\}$ is itself a string over the seven-symbol alphabet $\{\diamond, +, (,), \star, \varepsilon, \emptyset\}$. By interpreting these symbols as the digits 1 through 7, we can interpret any string over this larger alphabet as the base-8 representation of some unique integer. Thus, the set of all regular expressions over $\{\diamond\}$ is *at most* as large as the set of integers, and is therefore countably infinite. It follows that the set of all regular *languages* over $\{\diamond\}$ is also countably infinite.

- On the other hand, for any real number $0 \leq \alpha < 1$, we can define a corresponding language

$$L_\alpha = \left\{ \diamond^n \;\middle|\; \alpha 2^n \bmod 1 \geq 1/2 \right\}.$$

  In other words, L_α contains the string $\diamond^n$ if and only if the $(n+1)$th bit in the binary representation of α is equal to 1. For any distinct real numbers $\alpha \neq \beta$, the binary representations of α and β must differ in some bit, so $L_\alpha \neq L_\beta$. We conclude that the set of ***all*** languages over $\{\diamond\}$ is *at least* as large as the set of real numbers between 0 and 1, and is therefore uncountably infinite.

We will see several explicit examples of non-regular languages in later lectures. In particular, the set of all regular expressions over the alphabet $\{0, 1\}$ is itself a non-regular language over the alphabet $\{0, 1, +, (,), \star, \varepsilon, \emptyset\}$!

Exercises

1. (a) Prove that $\varnothing \bullet L = L \bullet \varnothing = \varnothing$, for every language L.
   (b) Prove that $\{\varepsilon\} \bullet L = L \bullet \{\varepsilon\} = L$, for every language L.
   (c) Prove that $(A \bullet B) \bullet C = A \bullet (B \bullet C)$, for all languages A, B, and C.
   (d) Prove that $|A \bullet B| \leq |A| \cdot |B|$, for all languages A and B. (The second $\cdot$ is multiplication!)
      i. Describe two languages A and B such that $|A \bullet B| < |A| \cdot |B|$.
      ii. Describe two languages A and B such that $|A \bullet B| = |A| \cdot |B|$.
   (e) Prove that L^* is finite if and only if $L = \varnothing$ or $L = \{\varepsilon\}$.
   (f) Prove that if $A \bullet B = B \bullet C$, then $A^* \bullet B = B \bullet C^* = A^* \bullet B \bullet C^*$, for all languages A, B, and C.
   (g) Prove that $(A \cup B)^* = (A^* \bullet B^*)^*$, for all languages A and B.

2. Recall that the reversal w^R of a string w is defined recursively as follows:

$$w^R := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ x^R \bullet a & \text{if } w = a \cdot x \end{cases}$$

The reversal L^R of any language L is the set of reversals of all strings in L:

$$L^R := \left\{ w^R \;\middle|\; w \in L \right\}.$$

(a) Prove that $(A \bullet B)^R = B^R \bullet A^R$ for all languages A and B.

(b) Prove that $(L^R)^R = L$ for every language L.

(c) Prove that $(L^*)^R = (L^R)^*$ for every language L.

3. Prove that each of the following regular expressions is equivalent to $(\mathtt{0} + \mathtt{1})^*$.

(a) $\varepsilon + \mathtt{0}(\mathtt{0} + \mathtt{1})^* + \mathtt{1}(\mathtt{1} + \mathtt{0})^*$

(b) $\mathtt{0}^* + \mathtt{0}^*\mathtt{1}(\mathtt{0} + \mathtt{1})^*$

(c) $((\varepsilon + \mathtt{0})(\varepsilon + \mathtt{1}))^*$

(d) $\mathtt{0}^*(\mathtt{1}\mathtt{0}^*)^*$

(e) $(\mathtt{1}^*\mathtt{0})^*(\mathtt{0}^*\mathtt{1})^*$

4. For each of the following languages in $\{\mathtt{0}, \mathtt{1}\}^*$, describe an equivalent regular expression. There are infinitely many correct answers for each language. (This problem will become significantly simpler after we've seen finite-state machines.)

(a) Strings that end with the suffix $\mathtt{0}^9 = \mathtt{000000000}$.

(b) All strings except $\mathtt{010}$.

(c) Strings that contain the substring $\mathtt{010}$.

(d) Strings that contain the subsequence $\mathtt{010}$.

(e) Strings that do not contain the substring $\mathtt{010}$.

(f) Strings that do not contain the subsequence $\mathtt{010}$.

(g) Strings that contain an even number of occurrences of the substring $\mathtt{010}$.

*(h) Strings that contain an even number of occurrences of the substring $\mathtt{000}$.

(i) Strings in which every occurrence of the substring $\mathtt{00}$ appears before every occurrence of the substring $\mathtt{11}$.

(j) Strings w such that *in every prefix of w*, the number of $\mathtt{0}$s and the number of $\mathtt{1}$s differ by at most 1.

*(k) Strings w such that *in every prefix of w*, the number of $\mathtt{0}$s and the number of $\mathtt{1}$s differ by at most 2.

*(l) Strings in which the number of $\mathtt{0}$s and the number of $\mathtt{1}$s differ by a multiple of 3.

*(m) Strings that contain an even number of $\mathtt{1}$s and an odd number of $\mathtt{0}$s.

★(n) Strings that represent a number divisible by 5 in binary.

5. For any string w, define $stutter(w)$ as follows:

$$stutter(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ aa \bullet stutter(x) & \text{if } w = ax \text{ for some symbol } a \text{ and string } x \end{cases}$$

Let L be an arbitrary regular language.

(a) Prove that the language $stutter(L) = \{stutter(w) \mid w \in L\}$ is also regular.

*(b) Prove that the language $stutter^{-1}(L) = \{w \mid stutter(w) \in L\}$ is also regular.

6. Recall that the ***reversal*** $\boldsymbol{w^R}$ of a string w is defined recursively as follows:

$$w^R = \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ x \bullet a & \text{if } w = ax \text{ for some symbol } a \text{ and some string } x \end{cases}$$

The reversal L^R of a language L is defined as the set of reversals of all strings in L:

$$L^R := \left\{ w^R \;\middle|\; w \in L \right\}$$

(a) Prove that $(L^*)^R = (L^R)^*$ for every language L.

(b) Prove that the reversal of any regular language is also a regular language. (You may assume part (a) even if you haven't proved it yet.)

You may assume the following facts without proof:

- $L^* \bullet L^* = L^*$ for every language L.
- $(w^R)^R = w$ for every string w.
- $(x \bullet y)^R = y^R \bullet x^R$ for all strings x and y.

[Hint: Yes, all three proofs use induction, but induction on what? And yes, all ***three*** *proofs.]*

7. This problem considers two special classes of regular expressions.

- A regular expression R is ***plus-free*** if and only if it never uses the $+$ operator.
- A regular expression R is ***top-plus*** if and only if either
  - R is plus-free, or
  - $R = S + T$, where S and T are top-plus.

For example, $\mathtt{1}((\mathtt{0}^*\mathtt{10})^*\mathtt{1})^*\mathtt{0}$ is plus-free and (therefore) top-plus; $\mathtt{01}^*\mathtt{0} + \mathtt{10}^*\mathtt{1} + \varepsilon$ is top-plus but not plus-free, and $\mathtt{0}(\mathtt{0} + \mathtt{1})^*(\mathtt{1} + \varepsilon)$ is neither top-plus nor plus-free.

Recall that two regular expressions R and S are ***equivalent*** if they describe exactly the same language: $L(R) = L(S)$.

(a) Prove that for any top-plus regular expressions R and S, there is a top-plus regular expression that is equivalent to RS.

(b) Prove that for any top-plus regular expression R, there is a ***plus-free*** regular expression S such that R^* and S^* are equivalent.

(c) Prove that for any regular expression, there is an equivalent top-plus regular expression.

You may assume the following facts without proof, for all regular expressions A, B, and C:

- $A(B+C)$ is equivalent to $AB+AC$.
- $(A+B)C$ is equivalent to $AC+BC$.
- $(A+B)^*$ is equivalent to $(A^*B^*)^*$.

8. (a) Describe and analyze an efficient algorithm to determine, given a regular expression R, whether $L(R)=\varnothing$.

   (b) Describe and analyze an efficient algorithm to determine, given a regular expression R, whether $L(R)=\{\varepsilon\}$. *[Hint: Use part (a).]*

   (c) Describe and analyze an efficient algorithm to determine, given a regular expression R, whether $L(R)$ is finite. *[Hint: Use parts (a) and (b).]*

   In each problem, assume you are given R as a regular expression *tree*, not just a raw string.

Life only avails, not the having lived. Power ceases in the instant of repose;
it resides in the moment of transition from a past to a new state,
in the shooting of the gulf, in the darting to an aim.

— Ralph Waldo Emerson, "Self Reliance", *Essays, First Series* (1841)

O Marvelous! what new configuration will come next?
I am bewildered with multiplicity.

— William Carlos Williams, "At Dawn" (1914)

3 Finite-State Machines

3.1 Intuition

Suppose we want to determine whether a given string $w[1..n]$ of bits represents a multiple of 5 in binary. After a bit of thought, you might realize that you can read the bits in w one at a time, from left to right, keeping track of the value modulo 5 of the prefix you have read so far.

```
MultipleOf5(w[1..n]):
  rem ← 0
  for i ← 1 to n
      rem ← (2 · rem + w[i]) mod 5
  if rem = 0
      return True
  else
      return False
```

Aside from the loop index i, which we need just to read the entire input string, this algorithm has a single local variable *rem*, which has only four different values: 0, 1, 2, 3, or 4.

This algorithm already runs in $O(n)$ time, which is the best we can hope for—after all, we have to read every bit in the input—but we can speed up the algorithm *in practice*. Let's define a ***change*** or ***transition*** function $\delta : \{0,1,2,3,4\} \times \{\texttt{0}, \texttt{1}\} \to \{0,1,2,3,4\}$ as follows:

$$\delta(q,a) = (2q + a) \bmod 5.$$

(Here I'm implicitly converting the symbols `0` and `1` to the corresponding integers 0 and 1.) Since we already know all values of the transition function, we can store them in a precomputed table, and then replace the computation in the main loop of MultipleOf5 with a simple array lookup.

We can also modify the return condition to check for different values modulo 5. To be completely general, we replace the final if-then-else lines with another array lookup, using an array $A[0..4]$ of booleans describing which final mod-5 values are "acceptable".

After both of these modifications, our algorithm looks like one of the following, depending on whether we want something iterative or recursive (with $q = 0$ in the initial call):

```
DoSomethingCool(w[1..n]):
  q ← 0
  for i ← 1 to n
      q ← δ[q, w[i]]
  return A[q]
```

```
DoSomethingCool(q, w):
  if w = ε
      return A[q]
  else
      decompose w = a · x
      return DoSomethingCool(δ(q, a), x)
```

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

If we want to use our new DoSomethingCool algorithm to implement MultipleOf5, we simply give the arrays δ and A the following hard-coded values:

q	$\delta[q, \texttt{0}]$	$\delta[q, \texttt{1}]$	$A[q]$
0	0	1	True
1	2	3	False
2	4	0	False
3	1	2	False
4	3	4	False

We can also visualize the behavior of DoSomethingCool by drawing a directed graph, whose vertices represent possible values of the variable q—the possible ***states*** of the algorithm—and whose edges are labeled with input symbols to represent transitions between states. Specifically, the graph includes the labeled directed edge $p \xrightarrow{a} q$ if and only if $\delta(p, a) = q$. To indicate the proper return value, we draw the "acceptable" final states using doubled circles. Here is the resulting graph for MultipleOf5:

State-transition graph for MultipleOf5

If we run the MultipleOf5 algorithm on the string `00101110110` (representing the number 374 in binary), the algorithm performs the following sequence of transitions:

$$0 \xrightarrow{\texttt{0}} 0 \xrightarrow{\texttt{0}} 0 \xrightarrow{\texttt{1}} 1 \xrightarrow{\texttt{0}} 2 \xrightarrow{\texttt{1}} 0 \xrightarrow{\texttt{1}} 1 \xrightarrow{\texttt{1}} 3 \xrightarrow{\texttt{0}} 1 \xrightarrow{\texttt{1}} 3 \xrightarrow{\texttt{1}} 2 \xrightarrow{\texttt{0}} 4$$

Because the final state is not the "acceptable" state 0, the algorithm correctly returns False. We can also think of this sequence of transitions as a walk in the graph, which is completely determined by the start state 0 and the sequence of edge labels; the algorithm returns True if and only if this walk ends at an "acceptable" state.

3.2 Formal Definitions

The object we have just described is an example of a ***finite-state machine***. A finite-state machine is a formal model of any system/machine/algorithm that can exist in a finite number of ***states*** and that transitions among those states based on sequence of ***input*** symbols.

Finite-state machines are also known as ***deterministic finite-state automata***, abbreviated ***DFAs***. The word "deterministic" means that the behavior of the machine is completely *determined* by the input string; we'll discuss nondeterministic automata in the next lecture. The word "automaton" (the singular of "automata") comes from ancient Greek αὐτόματος meaning "self-acting", from the roots αὐτό- ("self") and -ματος ("thinking, willing", the root of Latin *mentus*).

Formally, every finite-state machine consists of five components:

- An arbitrary finite set Σ, called the ***input alphabet***.

- Another arbitrary finite set $\boldsymbol{Q}$, whose elements are called ***states***.[1]
- An arbitrary ***transition*** function $\boldsymbol{\delta} : Q \times \Sigma \to Q$.
- A ***start state*** $\boldsymbol{s} \in Q$.
- A subset $\boldsymbol{A} \subseteq Q$ of ***accepting states***.

The behavior of a finite-state machine is governed by an ***input string*** w, which is a finite sequence of symbols from the input alphabet Σ. The machine ***reads*** the symbols in w one at a time in order (from left to right). At all times, the machine has a *current state* q; initially q is the machine's start state s. Each time the machine reads a symbol a from the input string, its current state ***transitions*** from q to $\delta(q, a)$. After all the characters have been read, the machine ***accepts*** w if the current state is in A and ***rejects*** w otherwise. In other words, every finite state machine runs the algorithm DoSomethingCool!

More formally, we extend the transition function $\delta : Q \times \Sigma \to Q$ of any finite-state machine to a function $\delta^* : Q \times \Sigma^* \to Q$ that transitions on *strings* as follows:

$$\delta^*(q, w) := \begin{cases} q & \text{if } w = \varepsilon, \\ \delta^*(\delta(q, a), x) & \text{if } w = ax. \end{cases}$$

Finally, a finite-state machine ***accepts*** a string w if and only if $\delta^*(s, w) \in A$, and ***rejects*** w otherwise. (Compare this definition with the recursive formulation of DoSomethingCool!)

For example, our final MultipleOf5 algorithm is a DFA with the following components:

- input alphabet: $\Sigma = \{\texttt{0}, \texttt{1}\}$
- state set: $Q = \{0, 1, 2, 3, 4\}$
- transition function: $\delta(q, a) = (2q + a) \bmod 5$
- start state: $s = 0$
- accepting states: $A = \{0\}$

This machine rejects the string `00101110110`, because

$$\begin{aligned} \delta^*(0, \texttt{00101110110}) &= \delta^*(\delta(0, \texttt{0}), \texttt{0101110110}) \\ = \delta^*(0, \texttt{0101110110}) &= \delta^*(\delta(0, \texttt{0}), \texttt{101110110}) \\ = \delta^*(0, \texttt{101110110}) &= \delta^*(\delta(0, \texttt{1}), \texttt{01110110}) = \cdots \\ &\vdots \\ \cdots = \delta^*(1, \texttt{110}) &= \delta^*(\delta(1, \texttt{1}), \texttt{10}) \\ = \delta^*(3, \texttt{10}) &= \delta^*(\delta(3, \texttt{1}), \texttt{0}) \\ = \delta^*(2, \texttt{0}) &= \delta^*(\delta(3, \texttt{0}), \varepsilon) \\ = \delta^*(4, \varepsilon) &= 4 \notin A. \end{aligned}$$

[1] It's unclear why we use the letter Q to refer to the state set, and lower-case q to refer to a generic state, but that is now the firmly-established notational standard. Although the formal study of finite-state automata began much earlier, its modern formulation was established in a 1959 paper by Michael Rabin and Dana Scott, for which they won the Turing award. Rabin and Scott called the set of states S, used lower-case s for a generic state, and called the start state s_0. On the other hand, in the 1936 paper for which the Turing award was named, Alan Turing used $q_1, q_2, \ldots, q_R$ to refer to states (or "m-configurations") of a generic Turing machine. Turing may have been mirroring the standard notation Q for configuration spaces in classical mechanics, also of uncertain origin.

We have already seen a more graphical representation of this entire sequence of transitions:

$$0 \xrightarrow{0} 0 \xrightarrow{0} 0 \xrightarrow{1} 1 \xrightarrow{0} 2 \xrightarrow{1} 0 \xrightarrow{1} 1 \xrightarrow{1} 3 \xrightarrow{0} 1 \xrightarrow{1} 3 \xrightarrow{1} 2 \xrightarrow{0} 4$$

The arrow notation is easier to read and write for specific examples, but surprisingly, most people actually find the more formal functional notation easier to use in formal proofs. Try them both!

We can equivalently define a DFA as a directed graph whose vertices are the states Q, whose edges are labeled with symbols from Σ, such that every vertex has exactly one outgoing edge with each label. In our drawings of finite state machines, the start state s is always indicated by an incoming arrow, and the accepting states A are always indicted by doubled circles. By induction, for any string $w \in \Sigma^*$, this graph contains a unique walk that starts at s and whose edges are labeled with the symbols in w in order. The machine accepts w if this walk ends at an accepting state. This graphical formulation of DFAs is incredibly useful for developing intuition and even designing DFAs. For proofs, it's largely a matter of taste whether to write in terms of extended transition functions or labeled graphs, but (as much as I wish otherwise) I actually find it easier to write ***correct*** proofs using the functional formulation.

3.3 Another Example

The following drawing shows a finite-state machine with input alphabet $\Sigma = \{0, 1\}$, state set $Q = \{s, t\}$, start state s, a single accepting state t, and the transition function

$$\delta(s, 0) = s, \quad \delta(s, 1) = t, \quad \delta(t, 0) = t, \quad \delta(t, 1) = s.$$

A simple finite-state machine.

For example, the two-state machine M at the top of this page accepts the string 00101110100 after the following sequence of transitions:

$$s \xrightarrow{0} s \xrightarrow{0} s \xrightarrow{1} t \xrightarrow{0} t \xrightarrow{1} s \xrightarrow{1} t \xrightarrow{1} s \xrightarrow{0} s \xrightarrow{1} t \xrightarrow{0} t \xrightarrow{0} t.$$

The same machine M rejects the string 11101101 after the following sequence of transitions:

$$s \xrightarrow{1} t \xrightarrow{1} s \xrightarrow{1} t \xrightarrow{0} t \xrightarrow{1} s \xrightarrow{1} t \xrightarrow{0} t \xrightarrow{1} s.$$

Finally, M rejects the empty string, because the start state s is not an accepting state.

From these examples and others, it is easy to conjecture that the language of M is the set of all strings of 0s and 1s with an odd number of 1s. So let's prove it!

Proof (tedious case analysis): Let $\#(a, w)$ denote the number of times symbol a appears in string w. We will prove the following stronger claims by induction, for any string w.

$$\delta^*(s, w) = \begin{cases} s & \text{if } \#(1, w) \text{ is even} \\ t & \text{if } \#(1, w) \text{ is odd} \end{cases} \qquad \text{and} \qquad \delta^*(t, w) = \begin{cases} t & \text{if } \#(1, w) \text{ is even} \\ s & \text{if } \#(1, w) \text{ is odd} \end{cases}$$

Let's begin. Let w be an arbitrary string. Assume that for any string x that is shorter than w, we have $\delta^*(s, x) = s$ and $\delta^*(t, x) = t$ if x has an even number of 1s, and $\delta^*(s, x) = t$ and $\delta^*(t, x) = s$ if x has an odd number of 1s. There are five cases to consider.

- If $w = \varepsilon$, then w contains an even number of $\mathtt{1}$s and $\delta^*(s, w) = s$ and $\delta^*(t, w) = t$ by definition.
- Suppose $w = \mathtt{1}x$ and $\#(\mathtt{1}, w)$ is even. Then $\#(\mathtt{1}, x)$ is odd, which implies

$$\begin{aligned}
\delta^*(s, w) &= \delta^*(\delta(s, \mathtt{1}), x) && \text{by definition of } \delta^* \\
&= \delta^*(t, x) && \text{by definition of } \delta \\
&= s && \text{by the inductive hypothesis}
\end{aligned}$$

$$\begin{aligned}
\delta^*(t, w) &= \delta^*(\delta(t, \mathtt{1}), x) && \text{by definition of } \delta^* \\
&= \delta^*(s, x) && \text{by definition of } \delta \\
&= T && \text{by the inductive hypothesis}
\end{aligned}$$

  Since the remaining cases are similar, I'll omit the line-by-line justification.

- If $w = \mathtt{1}x$ and $\#(\mathtt{1}, w)$ is odd, then $\#(\mathtt{1}, x)$ is even, so the inductive hypothesis implies

$$\begin{aligned}
\delta^*(s, w) &= \delta^*(\delta(s, \mathtt{1}), x) = \delta^*(t, x) = t \\
\delta^*(t, w) &= \delta^*(\delta(t, \mathtt{1}), x) = \delta^*(s, x) = s
\end{aligned}$$

- If $w = \mathtt{0}x$ and $\#(\mathtt{1}, w)$ is even, then $\#(\mathtt{1}, x)$ is even, so the inductive hypothesis implies

$$\begin{aligned}
\delta^*(s, w) &= \delta^*(\delta(s, \mathtt{0}), x) = \delta^*(s, x) = s \\
\delta^*(t, w) &= \delta^*(\delta(t, \mathtt{0}), x) = \delta^*(t, x) = t
\end{aligned}$$

- Finally, if $w = \mathtt{0}x$ and $\#(\mathtt{1}, w)$ is odd, then $\#(\mathtt{1}, x)$ is odd, so the inductive hypothesis implies

$$\begin{aligned}
\delta^*(s, w) &= \delta^*(\delta(s, \mathtt{0}), x) = \delta^*(s, x) = t \\
\delta^*(t, w) &= \delta^*(\delta(t, \mathtt{0}), x) = \delta^*(t, x) = s
\end{aligned}$$

□

Notice that this proof contains $|Q|^2 \cdot |\Sigma| + |Q|$ separate inductive arguments. For every pair of states p and q, we must argue about the language of all strings w such that $\delta^*(p, w) = q$, and we must consider every possible first symbol in w. We must also argue about $\delta(p, \varepsilon)$ for every state p. Each of those arguments is typically straightforward, but it's easy to get lost in the deluge of cases.

For this particular proof, however, we can reduce the number of cases by switching from tail recursion to *head* recursion. The following identity holds for all strings $x \in \Sigma^*$ and symbols $a \in \Sigma$:

$$\boxed{\delta^*(q, xa) = \delta(\delta^*(q, x), a)}$$

We leave the inductive proof of this identity as a straightforward exercise (hint, hint).

Proof (clever renaming, head induction): Let's rename the states with the integers 0 and 1 instead of s and t. Then the transition function can be described concisely as $\boldsymbol{\delta(q, a) = (q + a) \bmod 2}$. We claim that for every string w, we have $\delta^*(0, w) = \#(\mathtt{1}, w) \bmod 2$.

Let w be an arbitrary string, and assume that for any string x that is shorter than w that $\delta^*(0, x) = \#(\mathtt{1}, x) \bmod 2$. There are only two cases to consider: either w is empty or it isn't.

- If $w = \varepsilon$, then $\delta^*(0, w) = 0 = \#(\mathtt{1}, w) \bmod 2$ by definition.

- Otherwise, $w = xa$ for some string x and some symbol a, and we have

$$\begin{aligned}
\delta^*(0, w) &= \delta(\delta^*(0, x), a) && \text{by definition of } \delta^* \\
&= \delta(\#(\mathtt{1}, x) \bmod 2, a) && \text{by the inductive hypothesis} \\
&= (\#(\mathtt{1}, x) \bmod 2 + a) \bmod 2 && \text{by definition of } \delta \\
&= (\#(\mathtt{1}, x) + a) \bmod 2 && \text{by definition of mod 2} \\
&= (\#(\mathtt{1}, x) + \#(\mathtt{1}, a)) \bmod 2 && \text{because } \#(\mathtt{1}, \mathtt{0}) = 0 \text{ and } \#(\mathtt{1}, \mathtt{1}) = 1 \\
&= (\#(\mathtt{1}, xa)) \bmod 2 && \text{by definition of } \# \\
&= (\#(\mathtt{1}, w)) \bmod 2 && \text{because } w = xa \qquad \square
\end{aligned}$$

Hmmm. This "clever" proof is certainly shorter than the earlier brute-force proof, but is it actually *better*? Simpler? More intuitive? Easier to understand? I'm skeptical. Sometimes brute force really is more effective.

3.4 Real-World Examples

Finite-state machines were first formally defined in the mid-20th century, but people have been building automata for centuries, if not millennia. Many of the earliest records about automata are clearly mythological—for example, the brass giant Talus created by Hephaestus to guard Crete against intruders—but others are more believable, such as King-Shu's construction of a flying magpie from wood and bamboo in China around 500BCE.

Perhaps the most common examples of finite-state automata are ***clocks***. For example, the Swiss railway clock designed by Hans Hilfiker in 1944 has hour and minute hands that can indicate any time between 1:00 and 12:59. The minute hands advance discretely once per minute when they receive an electrical signal from a central master clock.[2] Thus, a Swiss railway clock is a finite-state machine with 720 states, one input symbol, and a simple transition function:

$$\begin{aligned}
Q &= \{(h, m) \mid 0 \le h11 \text{ and } 0 \le m \le 59\} \\
\Sigma &= \{\mathsf{tick}\} \\
\delta((h, m), \mathsf{tick}) &= \begin{cases} (h, m+1) & \text{if } m < 59 \\ (h+1, 0) & \text{if } h < 11 \text{ and } m = 59 \\ (0, 0) & \text{if } h = 11 \text{ and } m = 59 \end{cases}
\end{aligned}$$

This clock doesn't *quite* match our abstraction, because there's no "start" state or "accepting" states, unless perhaps you consider the "accepting" state to be the time when your train arrives.

A more playful example of a finite-state machine is the ***Rubik's cube***, a well-known mechanical puzzle invented independently by Ernő Rubik in Hungary and Terutoshi Ishigi in Japan in the mid-1970s. This puzzle has precisely 519,024,039,293,878,272,000 distinct configurations. In the unique *solved* configuration, each of the six faces of the cube shows exactly one color. We can change the configuration of the cube by rotating one of the six faces of the cube by 90 degrees, either clockwise or counterclockwise. The cube has six faces (front, back, left, right, up, and down), so there are exactly twelve possible turns, typically represented by the symbols $\mathsf{R}, \mathsf{L}, \mathsf{F}, \mathsf{B}, \mathsf{U}, \mathsf{D}, \bar{\mathsf{R}}, \bar{\mathsf{L}}, \bar{\mathsf{F}}, \bar{\mathsf{B}}, \bar{\mathsf{U}}, \bar{\mathsf{D}}$, where the letter indicates which face to turn and the presence or absence of a bar over the letter

[2] A second hand was added to the Swiss Railway clocks in the mid-1950s, which sweeps continuously around the clock in approximately 58½ seconds and then pauses at 12:00 until the next minute signal "to bring calm in the last moment and ease punctual train departure". Let's ignore that.

indicates turning counterclockwise or clockwise, respectively. Thus, we can represent a Rubik's cube as a finite-state machine with 519,024,039,293,878,272,000 states and an input alphabet with 12 symbols; or equivalently, as a directed graph with 519,024,039,293,878,272,000 vertices, each with 12 outgoing edges. In practice, the number of states is *far* too large for us to actually draw the machine or explicitly specify its transition function; nevertheless, the number of states is still finite. If we let the start state s and the sole accepting state be the solved state, then the language of this finite state machine is the set of all move sequences that leave the cube unchanged.

Three finite-state machines.

3.5 A Brute-Force Design Example

As usual in algorithm design, there is no purely mechanical recipe—no *automatic* method—no *algorithm*—for building DFAs in general. Here I'll describe one systematic approach that works reasonably well, although it tends to produce DFAs with many more states than necessary.

3.5.1 DFAs are Algorithms

The basic approach is to try to *construct an algorithm* that looks like MultipleOf5: A simple for-loop through the symbols, using a *constant* number of variables, where each variable (except the loop index) has only a *constant* number of possible values. Here, "constant" means an actual number that is not a function of the input size n. You should be able to compute the number of possible values for each variable *at compile time*.

For example, the following algorithm determines whether a given string in $\Sigma = \{0, 1\}$ contains the substring 11.

```
Contains11(w[1..n]):
  found ← False
  for i ← 1 to n
      if i = 1
          last2 ← w[1]
      else
          last2 ← w[i − 1] · w[i]
      if last2 = 11
          found ← True
  return found
```

Aside from the loop index, this algorithm has exactly two variables.

- A boolean flag *found* indicating whether we have seen the substring `11`. This variable has exactly two possible values: TRUE and FALSE.
- A string *last2* containing the last (up to) three symbols we have read so far. This variable has exactly 7 possible values: ε, `0`, `1`, `00`, `01`, `10`, and `11`.

Thus, altogether, the algorithm can be in at most $2 \times 7 = 14$ possible states, one for each possible pair (*found*, *last2*). Thus, we can encode the behavior of CONTAINS11 as a DFA with fourteen states, where the start state is (FALSE, ε) and the accepting states are all seven states of the form (TRUE, $*$). The transition function is described in the following table (split into two parts to save space):

q	$\delta[q,0]$	$\delta[q,1]$
(FALSE, ε)	(FALSE, `0`)	(FALSE, `1`)
(FALSE, `0`)	(FALSE, `00`)	(FALSE, `01`)
(FALSE, `1`)	(FALSE, `10`)	(**TRUE**, `11`)
(FALSE, `00`)	(FALSE, `00`)	(FALSE, `01`)
(FALSE, `01`)	(FALSE, `10`)	(**TRUE**, `11`)
(FALSE, `10`)	(FALSE, `00`)	(FALSE, `01`)
(FALSE, `11`)	(FALSE, `10`)	(**TRUE**, `11`)

q	$\delta[q,0]$	$\delta[q,1]$
(TRUE, ε)	(TRUE, `0`)	(TRUE, `1`)
(TRUE, `0`)	(TRUE, `00`)	(TRUE, `01`)
(TRUE, `1`)	(TRUE, `10`)	(TRUE, `11`)
(TRUE, `00`)	(TRUE, `00`)	(TRUE, `01`)
(TRUE, `01`)	(TRUE, `10`)	(TRUE, `11`)
(TRUE, `10`)	(TRUE, `00`)	(TRUE, `01`)
(TRUE, `11`)	(TRUE, `10`)	(TRUE, `11`)

For example, given the input string `1001011100`, this DFA performs the following sequence of transitions and then accepts.

$$(\text{FALSE}, \varepsilon) \xrightarrow{1} (\text{FALSE}, 1) \xrightarrow{0} (\text{FALSE}, 10) \xrightarrow{0} (\text{FALSE}, 00) \xrightarrow{1}$$
$$(\text{FALSE}, 01) \xrightarrow{0} (\text{FALSE}, 10) \xrightarrow{1} (\text{FALSE}, 01) \xrightarrow{1}$$
$$(\text{TRUE}, 11) \xrightarrow{1} (\text{TRUE}, 11) \xrightarrow{0} (\text{TRUE}, 10) \xrightarrow{0} (\text{TRUE}, 00)$$

3.5.2 . . . but Algorithms can be Wasteful

You can probably guess that the brute-force DFA we just constructed has considerably more states than necessary, especially after seeing its transition graph:

Our brute-force DFA for strings containing the substring `11`

For example, the state (FALSE, `11`) has no incoming transitions, so we can just delete it. (This state would indicate that we've never read `11`, but the last two symbols we read were `11`, which

is impossible!) More significantly, we don't need actually to remember both of the last two symbols, but only the penultimate symbol, because the last symbol is the one we're currently reading. This observation allows us to reduce the number of states from fourteen to only six.

A less brute-force DFA for strings containing the substring 11

But even this DFA has more states than necessary. Once the flag part of the state is set to TRUE, we know the machine will eventually accept, so we might as well merge all the accepting states together. More subtly, because both transitions out of (FALSE, 0) and (FALSE, ε) lead to the same states, we can merge those two states together as well. After all these optimizations, we obtain the following DFA with just three states:

- The start state, which indicates that the machine has not read the substring 11 and did not just read the symbol 1.
- An intermediate state, which indicates that the machine has not read the substring 11 but just read the symbol 1.
- A unique accept state, which indicates that the machine has read the substring 11.

This is the smallest possible DFA for this language.

A minimal DFA for superstrings of 11

While it is important not to use an excessive number of states when we design DFAs—too many states makes a DFA hard to understand—there is really no point in trying to reduce DFAs *by hand* to the absolute minimum number of states. Clarity is much more important than brevity (especially in this class), and DFAs with too *few* states can *also* be hard to understand. At the end of this note, I'll describe an efficient algorithm that automatically transforms any given DFA into an equivalent DFA with the fewest possible states.

3.6 Combining DFAs: The Product Construction

Now suppose we want to accept all strings that contain both 00 and 11 as substrings, in either order. Intuitively, we'd like to run two DFAs in parallel—the DFA M_{00} to detect superstrings of 00 and a similar DFA M_{11} obtained from M_{00} by swapping $0 \leftrightarrow 1$ everywhere—and then accept the input string if and only if *both* of these DFAs accept.

In fact, we can encode precisely this "parallel computation" into a single DFA using the following *product construction* first proposed by Edward Moore in 1956:

- The states of the new DFA are all ordered pairs (p, q), where p is a state in M_{00} and q is a state in M_{11}.
- The start state of the new DFA is the pair (s, s'), where s is the start state of M_{00} and s' is the start state of M_{11}.
- The new DFA includes the transition $(p, q) \xrightarrow{a} (p', q')$ if and only if M_{00} contains the transition $p \xrightarrow{a} p'$ and M_{11} contains the transition $q \xrightarrow{a} q'$.
- Finally, (p, q) is an accepting state of the new DFA if and only if p is an accepting state in M_{00} and q is an accepting state in M_{11}.

The resulting nine-state DFA is shown on the next page, with the two factor DFAs M_{00} and M_{11} shown in gray for reference. (The state (a, a) can be removed, because it has no incoming transition, but let's not worry about that now.)

Building a DFA for the language of strings containing both 00 and 11.

More generally, let $M_1 = (\Sigma, Q_1, \delta_1, s_1, A_1)$ be an arbitrary DFA that accepts some language L_1, and let $M_2 = (\Sigma, Q_2, \delta_2, s_2, A_2)$ be an arbitrary DFA that accepts some language L_2 (over the same alphabet Σ). We can construct a third DFA $M = (\Sigma, Q, \delta, s, A)$ that accepts the intersection language $L_1 \cap L_2$ as follows.

$$\begin{aligned} Q &:= Q_1 \times Q_2 = \big\{(p, q) \bigm| p \in Q_1 \text{ and } q \in Q_2\big\} \\ \delta((p, q), a) &:= \big(\delta_1(p, a),\ \delta_2(q, a)\big) \\ s &:= (s_1, s_2) \\ A &:= A_1 \times A_2 = \big\{(p, q) \bigm| p \in A_1 \text{ and } q \in A_2\big\} \end{aligned}$$

To convince ourselves that this product construction is actually correct, let's consider the extended transition function $\delta^* \colon (Q \times Q') \times \Sigma^* \to (Q \times Q')$, which acts on strings instead of individual symbols. Recall that this function is defined recursively as follows:

$$\delta^*\big((p, q), w\big) := \begin{cases} (p, q) & \text{if } w = \varepsilon, \\ \delta^*\big(\delta((p, q), a),\ x\big) & \text{if } w = ax. \end{cases}$$

This function behaves exactly as we should expect:

Lemma 3.1. *$\delta^*((p,q),w) = \big(\delta_1^*(p,w),\ \delta_2^*(q,w)\big)$ for any string w.*

Proof: Let w be an arbitrary string. Assume $\delta^*((p,q),x) = \big(\delta_1^*(p,x),\ \delta_2^*(q,x)\big)$ for every string x that is shorter than w. As usual, there are two cases to consider.

- First suppose $w = \varepsilon$:

$$\begin{aligned}
\delta^*\big((p,q),\varepsilon\big) &= (p,q) && \text{by the definition of } \delta^* \\
&= \big(\delta_1^*(p,\varepsilon),\ q\big) && \text{by the definition of } \delta_1^* \\
&= \big(\delta_1^*(p,e),\ \delta_2^*(q,\varepsilon)\big) && \text{by the definition of } \delta_2^*
\end{aligned}$$

- Now suppose $w = ax$ for some symbol a and some string x:

$$\begin{aligned}
\delta^*\big((p,q),ax\big) &= \delta^*\big(\delta((p,q),a),\ x\big) && \text{by the definition of } \delta^* \\
&= \delta^*\big((\delta_1(p,a),\ \delta_2(q,a)),\ x\big) && \text{by the definition of } \delta \\
&= \big(\delta_1^*((\delta_1(p,a),\ x),\ \delta_2^*(\delta_2(q,a),x)\big) && \text{by the induction hypothesis} \\
&= \big(\delta_1^*(p,ax),\ \delta_2^*(q,ax)\big) && \text{by the definitions of } \delta_1^* \text{ and } \delta_2^*.
\end{aligned}$$

In both cases, we conclude that $\delta^*((p,q),w) = \big(\delta_1^*(p,w),\ \delta_2^*(q,w)\big)$. □

An immediate consequence of this lemma is that for every string w, we have $\delta^*(s,w) \in A$ if and only if both $\delta_1^*(s_1,w) \in A_1$ and $\delta_2^*(s_2,w) \in A_2$. In other words, M accepts w if and only if ***both*** M_1 accepts w ***and*** M_2 accept w, as required.

As usual, this construction technique does not necessarily yield *minimal* DFAs. For example, in our first example of a product DFA, illustrated above, the central state (a,a) cannot be reached by any other state and is therefore redundant. Whatever.

Similar product constructions can be used to build DFAs that accept any other boolean combination of languages; in fact, the only part of the construction that changes is the choice of accepting states. For example:

- To accept the union $L_1 \cup L_2$, define $A = \big\{(p,q) \bigm| p \in A_1 \textbf{ or } q \in A_2\big\}$.
- To accept the difference $L_1 \setminus L_2$, define $A = \big\{(p,q) \bigm| p \in A_1 \textbf{ but } q \notin A_2\big\}$.
- To accept the symmetric difference $L_1 \oplus L_2$, define $A = \big\{(p,q) \bigm| p \in A_1 \textbf{ xor } q \in A_2\big\}$.

Examples of these constructions are shown on the next page.

Moreover, by cascading this product construction, we can construct DFAs that accept arbitrary boolean combinations of arbitrary finite collections of regular languages.

3.7 Automatic Languages and Closure Properties

The ***language*** of a finite state machine M, denoted $\boldsymbol{L(M)}$, is the set of all strings in Σ^* that M accepts. More formally, if $M = (\Sigma, Q, \delta, s, A)$, then

$$L(M) \ := \ \big\{w \in \Sigma^* \bigm| \delta^*(s,w) \in A\big\}.$$

We call a language ***automatic*** if it is the language of some finite state machine. Our product construction examples let us prove that the set of automatic languages is ***closed*** under simple boolean operations.

DFAs for (a) strings that contain `00` or `11`, (b) strings that contain either `00` or `11` but not both, and (c) strings that contain `11` if they contain `00`. These DFAs are identical except for their choices of accepting states.

Theorem 3.2. *Let L and L' be arbitrary automatic languages over an arbitrary alphabet Σ.*

- $\overline{L} = \Sigma^* \setminus L$ *is automatic.*
- $L \cup L'$ *is automatic.*
- $L \cap L'$ *is automatic.*
- $L \setminus L'$ *is automatic.*
- $L \oplus L'$ *is automatic.*

Eager students may have noticed that a Google search for the phrase "automatic language" turns up ***no*** results that are relevant for this class, except perhaps this lecture note. That's because "automatic" is just a synonym for "regular"! This equivalence was first observed by Stephen Kleene (the inventor of regular expressions) in 1956.

Theorem 3.3 (Kleene). *For any regular expression R, there is a DFA M such that $L(R) = L(M)$. For any DFA M, there is a regular expression R such that $L(M) = L(R)$.*

Unfortunately, we don't yet have all the tools we need to prove Kleene's theorem; we'll return to the proof in the next lecture note, after we have introduced *nondeterministic* finite-state machines. The proof is actually constructive—there are explicit algorithms that transform arbitrary DFAs into equivalent regular expressions and vice versa.[3]

This equivalence between regular and automatic languages implies that the set of ***regular*** languages is also closed under simple boolean operations. The union of two regular languages is regular *by definition*, but it's much less obvious that *every* boolean combination of regular languages can also be described by regular expressions.

Corollary 3.4. *Let L and L' be arbitrary* **regular** *languages over an arbitrary alphabet Σ.*

- $\overline{L} = \Sigma^* \setminus L$ *is regular.*
- $L \cap L'$ *is regular.*
- $L \setminus L'$ *is regular.*
- $L \oplus L'$ *is regular.*

Conversely, because concatenations and Kleene closures of regular languages are regular *by definition*, we can immediately conclude that concatenations and Kleene closures of automatic languages are automatic.

[3]These conversion algorithms run in exponential time in the worst case, but that's unavoidable. There are regular languages whose smallest accepting DFA is exponentially larger than their smallest regular expression, and there are regular languages whose smallest regular expression is exponentially larger than their smallest accepting DFA.

Corollary 3.5. *Let L and L' be arbitrary automatic languages.*

- *$L \bullet L'$ is automatic.*
- *L^* is automatic.*

These results give us several options to prove that a given languages is regular or automatic. We can either (1) build a regular expression that describes the language, (2) build a DFA that accepts the language, or (3) build the language from simpler pieces from other regular/automatic languages. (Later we'll see a fourth option, and possibly even a fifth.)

3.8 Proving a Language is Not Regular

But now suppose we're faced with a language L where none of these techniques seem to work. How would we prove L is *not* regular? By Theorem **??**, it suffices to prove that there is no finite-state automaton that accepts L. Equivalently, we need to prove that any automaton that accepts L requires infinitely many states. That may sound tricky, what with the "infinitely many", but there's actually a fairly simple technique to prove exactly that.

3.8.1 Distinguishing Suffixes

Perhaps the single most important feature of DFAs is that they have no memory other than the current state. Once a DFA enters a particular state, all *future* transitions depend only on that state and *future* input symbols; *past* input symbols are simply forgotten.

For example, consider our very first DFA, which accepts the binary representations of integers divisible by 5.

DFA accepting binary multiples of 5.

The strings `0010` and `11011` both lead this DFA to state 2, although they follow different transitions to get there. Thus, for any string z, the strings `0010`z and `11011`z also lead to the same state in this DFA. In particular, `0010`z leads to the accepting state if and only if `11011`z leads to the accepting state. It follows that `0010`z is divisible by 5 if and only if `11011`z is divisible by 5.

More generally, any DFA $M = (\Sigma, Q, s, A, \delta)$ defines an equivalence relation over Σ^*, where two strings x and y are equivalent if and only if they lead to the same state, or more formally, if $\delta^*(s, x) = \delta^*(s, y)$. If x and y are equivalent strings, then for any string z, the strings xz and yz are also equivalent. In particular, M accepts xz if and only if M accepts yz. Thus, if L is the language accepted by M, then $xz \in L$ if and only if $yz \in L$. In short, if the *machine* can't distinguish between x and y, then the *language* can't distinguish between xz and yz for any suffix z.

Now let's turn the previous argument on its head. Let L be an arbitrary language, and let x and y be arbitrary strings. A ***distinguishing suffix*** for x and y (with respect to L) is a third string z such that *exactly one* of the strings xz and yz is in L. If x and y have a distinguishing

suffix z, then in *any* DFA that accepts L, the strings xz and yz must lead to different states, and therefore the strings x and y must lead to different states!

For example, let L_5 denote the the set of all strings over $\{0, 1\}$ that represent multiples of 5 in binary. Then the strings $x = 01$ and $y = 0011$ are distinguished by the suffix $z = 01$:

$$xz = 01 \bullet 01 = 0101 \in L_5 \qquad \text{(because } 0101_2 = 5\text{)}$$
$$yz = 0011 \bullet 01 = 001101 \notin L_5 \qquad \text{(because } 001101_2 = 13\text{)}$$

It follows that in ***every*** DFA that accepts L_5, the strings 01 and 0011 lead to different states. Moreover, since neither 01 nor 0011 belong to L_5, every DFA that accepts L_5 must have at least two *non-accepting* states, and therefore at least three states overall.

3.8.2 Fooling Sets

A ***fooling set*** for a language L is a set F of strings such that *every* pair of strings in F has a distinguishing suffix. For example, $F = \{0, 1, 10, 11, 100\}$ is a fooling set for the language L_5 of binary multiples of 5, because each pair of strings in F has a distinguishing suffix:

- 0 distinguishes 0 and 1;
- 0 distinguishes 0 and 10;
- 0 distinguishes 0 and 11;
- 0 distinguishes 0 and 100;
- 1 distinguishes 1 and 10;
- 01 distinguishes 1 and 11;
- 01 distinguishes 1 and 100;
- 1 distinguishes 10 and 11;
- 1 distinguishes 10 and 100;
- 11 distinguishes 11 and 100.

Each of these five strings leads to a different state, for *any* DFA M that accepts L_5. Thus, *every* DFA that accepts the language L_5 has at least five states. And hey, look, we already have a DFA for L_5 with five states, so that's the best we can do!

More generally, for *any* language L, and *any* fooling set F for L, *every* DFA that accepts L must have at least $|F|$ states. In particular, if the fooling set F is *infinite*, then every DFA that accepts L must have an *infinite* number of states. But there's no such thing as a ***finite***-state machine with an ***infinite*** number of states!

If L has an infinite fooling set, then L is not regular.

This is arguably both the simplest and most powerful method for proving that a language is non-regular. Here are a few canonical examples of the fooling-set technique in action.

Lemma 3.6. *The language* $L = \{0^n 1^n \mid n \geq 0\}$ *is not regular.*

Proof: Consider the infinite set $F = \{0^n \mid n \geq 0\}$, or more simply $F = 0^*$.

Let x and y be arbitrary distinct strings in F.

The definition of F implies $x = 0^i$ and $y = 0^j$ for some integers $i \neq j$.

The suffix $z = 1^i$ distinguishes x and y, because $xz = 0^i 1^i \in L$, but $yz = 0^j 1^i \notin L$.
Thus, *every* pair of distinct strings in F has a distinguishing suffix.
In other words, F is a fooling set for L.
Because F is infinite, L cannot be regular. □

Lemma 3.7. *The language $L = \{ww^R \mid w \in \Sigma^*\}$ of even-length palindromes is not regular.*

Proof: Let F denote the set 0^*1, and let x and y be arbitrary distinct strings in F. Then we must have $x = 0^i 1$ and $y = 0^j 1$ for some integers $i \neq j$. The suffix $z = 10^i$ distinguishes x and y, because $xz = 0^i 110^i \in L$, but $yz = 0^i 110^j \notin L$. We conclude that F is a fooling set for L. Because F is infinite, L cannot be regular. □

Lemma 3.8. *The language $L = \{0^{2^n} \mid n \geq 0\}$ is not regular.*

Proof ($F = L$): Let x and y be arbitrary distinct strings in L. Then we must have $x = 0^{2^i}$ and $y = 0^{2^j}$ for some integers $i \neq j$. The suffix $z = 0^{2^i}$ distinguishes x and y, because $xz = 0^{2^i+2^i} = 0^{2^{i+1}} \in L$, but $yz = 0^{2^i+2^j} \notin L$. We conclude that L itself is a fooling set for L. Because L is infinite, L cannot be regular. □

Proof ($F = 0^*$): Let x and y be arbitrary distinct strings in 0^*. Then we must have $x = 0^i$ and $y = 0^j$ for some integers $i \neq j$; without loss of generality, assume $i < j$. Let k be any positive integer such that $2^k > j$. Consider the suffix $z = 0^{2^k - i}$. We have $xz = 0^{i+(2^k-i)} = 0^{2^k} \in L$, but $yz = 0^{j+(2^k-i)} = 0^{2^k-i+j} \notin L$, because

$$2^k \;<\; 2^k - i + j \;<\; 2^k + j \;<\; 2^k + 2^k \;=\; 2^{k+1}.$$

Thus, z is a distinguishing suffix for x and y. We conclude that 0^* is a fooling set for L. Because L is infinite, L cannot be regular. □

Proof ($F = 0^*$ again): Let x and y be arbitrary distinct strings in 0^*. Then we must have $x = 0^i$ and $y = 0^j$ for some integers $i \neq j$; without loss of generality, assume $i < j$. Let k be any positive integer such that $2^{k-1} > j$. Consider the suffix $z = 0^{2^k - j}$. We have $xz = 0^{i+(2^k-j)} = 0^{2^k-j+i} \notin L$, because

$$2^{k-1} \;<\; 2^k - 2^{k-1} + i \;<\; 2^k - j + i \;<\; 2^k.$$

On the other hand, $yz = 0^{j+(2^k-j)} = 0^{2^k} \in L$. Thus, z is a distinguishing suffix for x and y. We conclude that 0^* is a fooling set for L. Because L is infinite, L cannot be regular. □

The previous examples show the flexibility of this proof technique; a single non-regular language can have many different infinite fooling sets,[4] and each pair of strings in any fooling set can have many different distinguishing suffixes. Fortunately, we only have to find *one* infinite set F and *one* distinguishing suffix for each pair of strings in F.

Lemma 3.9. *The language $L = \{0^p \mid p \text{ is prime}\}$ is not regular.*

[4]At some level, this observation is trivial. If F is an infinite fooling set for L, then every infinite subset of F is also an infinite fooling set for L!

Proof ($F = 0^*$): Again, we use 0^* as our fooling set, but but the actual argument is somewhat more complicated than in our earlier examples.

Let x and y be arbitrary distinct strings in 0^*. Then we must have $x = 0^i$ and $y = 0^j$ for some integers $i \neq j$; without loss of generality, assume that $i < j$. Let p be any prime number larger than i. Because $p + 0(j-i)$ is prime and $p + p(j-i) > p$ is not, there must be a positive integer $k \leq p$ such that $p + (k-1)(j-i)$ is prime but $p + k(j-i)$ is not. Then I claim that the suffix $z = 0^{p+(k-1)j-ki}$ distinguishes x and y:

$$xz = 0^i\, 0^{p+(k-1)j-ki} = 0^{p+(k-1)(j-i)} \in L \qquad \text{because } p + (k-1)(j-i) \text{ is prime;}$$
$$yz = 0^j\, 0^{p+(k-1)j-ki} = 0^{p+k(j-i)} \notin L \qquad \text{because } p + k(j-i) \text{ is not prime.}$$

(Because $i < j$ and $i < p$, the suffix $0^{p+(k-1)j-ki} = 0^{(p-i)+(k-1)(j-i)}$ has positive length and therefore *actually exists!*) We conclude that 0^* is indeed a fooling set for L, which implies that L is not regular. □

Proof ($F = L$): Let x and y be arbitrary distinct strings in L. Then we must have $x = 0^p$ and $y = 0^q$ for some primes $p \neq q$; without loss of generality, assume $p < q$.

Now consider strings of the form $0^{p+k(q-p)}$. Because $p + 0(q-p)$ is prime and $p + p(q-p) > p$ is not prime, there must be a non-negative integer $k < p$ such that $p + k(p-q)$ is prime but $p + (k+1)(p-q)$ is not prime. I claim that the suffix $z = 0^{k(q-p)}$ distinguishes x and y:

$$xz = 0^p\, 0^{k(q-p)} = 0^{p+k(p-q)} \in L \qquad \text{because } p + k(p-q) \text{ is prime;}$$
$$yz = 0^q\, 0^{k(q-p)} = 0^{p+(k+1)(q-p)} \notin L \qquad \text{because } p + (k+1)(p-q) \text{ is not prime.}$$

We conclude that L is a fooling set for itself!! Because L is infinite, L cannot be regular! □

Obviously the most difficult part of this technique is coming up with an appropriate fooling set. Fortunately, *most* languages L—in particular, almost all languages that students are asked to prove non-regular on homeworks or exams—fall into one of two categories:

- Some simple regular language like 0^* or 10^*1 or $(01)^*$ is a fooling set for L. In particular, the fooling set is a regular language with one Kleene star and no $+$.
- The language L itself is a fooling set for L.

The most important point to remember is that ***you choose*** the fooling set F, and you can use that fooling set to effectively impose additional structure on the language L.

★★★

I'm not sure yet how to express this effectively, but here is some more intuition about choosing fooling sets and distinguishing suffixes.

As a sanity check, try to write an *algorithm* to recognize strings in L, as described at the start of this note, where the only variable that can take on an unbounded number of values is the loop index i. (I should probably rewrite that template as a while-loop or tail recursion, but anyway....) If you succeed, the language is regular. But if you fail, it's probably because there are counters of string variables that you can't get rid of. ***One of those unavoidable counters is the basis for your fooling set.***

For example, any algorithm that recognizes the language $\{0^n1^n2^n \mid n \geq 0\}$ "obviously" has to count 0s and 1s in the input string. (We can avoid counting 2s by decrementing the 0 counter.) Because the 0s come first in the string, this intuition suggests using strings of the form 0^n as our fooling set and matching strings of the form 1^n2^n as distinguishing suffixes. (This is a rare example of an "obvious" fact that is actually true.)

It's also important to remember that when you choose the fooling set, you can effectively impose additional structure that isn't present in the language already. For example, to prove that the language $L = \{w \in (0+1)^* \mid \#(0,w) = (1,w)\}$ is not regular, we can use strings of the form 0^n as our fooling set and matching strings of the form 1^n as distinguishing suffixes, ***exactly*** as we did for $\{0^n1^n \mid n \geq 0\}$. The fact that L contains strings that start with 1 is irrelevant. There may be more equivalence classes that our proof doesn't find, but since we found an infinite set of equivalence class, we don't care.

At some level, this fooling set proof is implicitly considering the simpler language $L \cap 0^*1^* = \{0^n1^n \mid n \geq 0\}$. If L were regular, then $L \cap 0^*1^*$ would also be regular, because regular languages are closed under intersection.

*3.9 The Myhill-Nerode Theorem

The fooling set technique implies a *necessary* condition for a language to be accepted by a DFA—the language must have no infinite fooling sets. In fact, this condition is also *sufficient*. The following powerful theorem was first proved by Anil Nerode in 1958, strengthening a 1957 result of John Myhill.[5] We write $x \equiv_L y$ if $xz \in L \iff yz \in L$ for all strings z.

The Myhill-Nerode Theorem. *For any language L, the following are equal:*
(a) the minimum number of states in a DFA that accepts L,
(b) the maximum size of a fooling set for L, and
(c) the number of equivalence classes of $\equiv_L$.
In particular, L is accepted by a DFA if and only if every fooling set for L is finite.

Proof: Let L be an arbitrary language.

We have already proved that the size of any fooling set for L is at most the number of states in any DFA that accepts L, so (a)≥(b). It also follows directly from the definitions that $F \subseteq \Sigma^*$ is a fooling set for L if and only if F contains at most one string in each equivalence class of $\equiv_L$; thus, (b)=(c). We complete the proof by showing that (a)≤(c).

We have already proved that if $\equiv_L$ has an infinite number of equivalence classes, there is no DFA that accepts L, so assume that the number of equivalence classes is finite. For any string w,

[5] Myhill considered the finer equivalence relation $x \sim_L y$, meaning $wxz \in L$ if and only if $wyz \in L$ for all strings w and z, and proved that L is regular if and only if $\sim_L$ defines a finite number of equivalence classes. Like most of Myhill's early automata research, this result appears in an unpublished Air Force technical report. The modern Myhill-Nerode theorem appears (in an even more general form) as a minor lemma in Nerode's 1958 paper, which (not surprisingly) does not cite Myhill.

let $[w]$ denote its equivalence class. We define a DFA $M_{\equiv} = (\Sigma, Q, s, A, \delta)$ as follows:

$$\begin{aligned} Q &:= \big\{[w] \;\big|\; w \in \Sigma^*\big\} \\ s &:= [\varepsilon] \\ A &:= \big\{[w] \;\big|\; w \in L\big\} \\ \delta([w], a) &:= [w \bullet a] \end{aligned}$$

We claim that this DFA accepts the language L; this claim completes the proof of the theorem.

But before we can prove anything about this DFA, we first need to verify that it is actually well-defined. Let x and y be two strings such that $[x] = [y]$. By definition of L-equivalence, for any string z, we have $xz \in L$ if and only if $yz \in L$. It immediately follows that for any symbol $a \in \Sigma$ and any string z', we have $xaz' \in L$ if and only if $yaz' \in L$. Thus, by definition of L-equivalence, we have $[xa] = [ya]$ for every symbol $a \in \Sigma$. We conclude that the function δ is indeed well-defined.

An easy inductive proof implies that $\delta^*([\varepsilon], x) = [x]$ for every string x. Thus, M accepts string x if and only if $[x] = [w]$ for some string $w \in L$. But if $[x] = [w]$, then by definition (setting $z = \varepsilon$), we have $x \in L$ if and only if $w \in L$. So M accepts x if and only if $x \in L$. In other words, M accepts L, as claimed, so the proof is complete. □

*3.10 Minimal Automata

Given a DFA $M = (\Sigma, Q, s, A, \delta)$, suppose we want to find another DFA $M' = (\Sigma, Q', s', A', \delta')$ with the fewest possible states that accepts the same language. In this final section, we describe an efficient algorithm to minimize DFAs, first described (in slightly different form) by Edward Moore in 1956. We analyze the running time of Moore's in terms of two parameters: $n = |Q|$ and $\sigma = |\Sigma|$.

In the preprocessing phase, we find and remove any states that cannot be reached from the start state s; this filtering can be performed in $O(n\sigma)$ time using any graph traversal algorithm. So from now on we assume that all states are reachable from s.

Now we recursively define two states p and q in the remaining DFA to be ***distingushable***, written $\boldsymbol{p \not\sim q}$, if at least one of the following conditions holds:

- $p \in A$ and $q \notin A$,
- $p \notin A$ and $q \in A$, or
- $\delta(p, a) \not\sim \delta(q, a)$ for some $a \in \Sigma$.

Equivalently, $p \not\sim q$ if and only if there is a string z such that exactly one of the states $\delta^*(p, z)$ and $\delta^*(q, z)$ is accepting. (Sound familiar?) Intuitively, the main algorithm assumes that all states are equivalent until proven otherwise, and then repeatedly looks for state pairs that can be proved distinguishable.

The main algorithm maintains a two-dimensional table, indexed by the states, where $Dist[p, q] = \textsc{True}$ indicates that we have proved states p and q are distinguishable. Initially, for all states p and q, we set $Dist[p, q] \leftarrow \textsc{True}$ if $p \in A$ and $q \notin A$ or vice versa, and $Dist[p, q] = \textsc{False}$ otherwise. Then we repeatedly consider each pair of states and each symbol to find more distinguishable pairs, until we make a complete pass through the table without modifying it. The table-filling algorithm can be summarized as follows:

```
MinDFATable(Σ, Q, s, A, δ):
  for all p ∈ Q
      for all q ∈ Q
          if (p ∈ A and q ∉ A) or (p ∉ A and q ∈ A)
              Dist[p, q] ← True
          else
              Dist[p, q] ← False
  notdone ← True
  while notdone
      notdone ← False
      for all p ∈ Q
          for all q ∈ Q
              if Dist[p, q] = False
              for all a ∈ Σ
                  if Dist[δ(p, a), δ(q, a)]
                      Dist[p, q] ← True
                      notdone ← True
  return Dist
```

The algorithm must eventually halt, because there are only a finite number of entries in the table that can be marked. In fact, the main loop is guaranteed to terminate after at most n iterations, which implies that the entire algorithm runs in $\boldsymbol{O(\sigma n^3)}$ time. Once the table is filled,[6] any two states p and q such that $Dist(p,q) = \textsc{False}$ are equivalent and can be merged into a single state. The remaining details of constructing the minimized DFA are straightforward.

★★★ Need to prove that the main loop terminates in at most n iterations.

With more care, Moore's minimization algorithm can be modified to run in $O(\sigma n^2)$ time. A faster DFA minimization algorithm, due to John Hopcroft, runs in $O(\sigma n \log n)$ time.

Example

To get a better idea how this algorithm works, let's visualize its execution on our earlier brute-force DFA for strings containing the substring `11`. This DFA has four unreachable states: (False, `11`), (True, ε), (True, `0`), and (True, `1`). We remove these states, and relabel the remaining states for easier reference. (In an actual implementation, the states would almost certainly be represented by indices into an array anyway, not by mnemonic labels.)

The main algorithm initializes (the bottom half of) a 10×10 table as follows. (In the following figures, cells marked $\times$ have value True and blank cells have value False.)

[6] More experienced readers should be enraged by the mere suggestion that any algorithm merely *fills in a table*, as opposed to *evaluating a recurrence*. This algorithm is no exception. Consider the boolean function $Dist(p,q,k)$, which equals True if and only if p and q can be distinguished by some string of length at most k. This function obeys the following recurrence:

$$Dist(p,q,k) = \begin{cases} (p \in A) \oplus (q \in A) & \text{if } k = 0, \\ Dist(p,q,k-1) \vee \bigvee_{a \in \Sigma} Dist\big(\delta(p,a), \delta(q,a), k-1\big) & \text{otherwise.} \end{cases}$$

Moore's "table-filling" algorithm is just a space-efficient dynamic programming algorithm to evaluate this recurrence.

Our brute-force DFA for strings containing the substring 11, after removing all four unreachable states

	0	1	2	3	4	5	6	7	8
1									
2									
3									
4									
5									
6	×	×	×	×	×	×			
7	×	×	×	×	×	×			
8	×	×	×	×	×	×			
9	×	×	×	×	×	×			

In the first iteration of the main loop, the algorithm discovers several distinguishable pairs of states. For example, the algorithm sets $Dist[0,2] \leftarrow \textsc{True}$ because $Dist[\delta(0,1),\delta(2,1)] = Dist[2,9] = \textsc{True}$. After the iteration ends, the table looks like this:

	0	1	2	3	4	5	6	7	8
1									
2	×	×							
3			×						
4	×	×		×					
5			×		×				
6	×	×	×	×	×	×			
7	×	×	×	×	×	×			
8	×	×	×	×	×	×			
9	×	×	×	×	×	×			

The second iteration of the while loop makes no further changes to the table—We got lucky!—so the algorithm terminates.

The final table implies that the 10 states of our DFA fall into exactly three equivalence classes: $\{0,1,3,5\}$, $\{2,4\}$, and $\{6,7,8,9\}$. Replacing each equivalence class with a single state gives us the three-state DFA that we already discovered.

Exercises

1. For each of the following languages in $\{0,1\}^*$, describe a deterministic finite-state machine that accepts that language. There are infinitely many correct answers for each language. "Describe" does not necessarily mean "draw".

   (a) Only the string 0110.

Equivalence classes of states in our DFA, and the resulting minimal equivalent DFA.

(b) Every string except `0110`.

(c) Strings that contain the substring `0110`.

(d) Strings that do not contain the substring `0110`.

*(e) Strings that contain an even number of occurrences of the substring `0110`. (For example, this language contains the strings `0110110` and `01011`.)

(f) Strings that contain the *subsequence* `0110`.

(g) Strings that do not contain the *subsequence* `0110`.

*(h) Strings that contain an even number of occurrences of the *subsequence* `0110`.

(i) Strings that contain an even number of `1`s and an odd number of `0`s.

(j) Every string that represents a number divisible by 7 in binary.

(k) Every string whose reversal represents a number divisible by 7 in binary.

(l) Strings in which the substrings `01` and `10` appear the same number of times.

(m) Strings such that in every prefix, the number of `0`s and the number of `1`s differ by at most 1.

(n) Strings such that in every prefix, the number of `0`s and the number of `1`s differ by at most 4.

(o) Strings that end with $\texttt{0}^{10} = \texttt{0000000000}$.

(p) All strings in which the number of `0`s is even if and only if the number of `1`s is *not* divisible by 3.

(q) All strings that are both the binary representation of an integer divisible by 3 and the ternary (base-3) representation of an integer divisible by 4.

(r) Strings in which the number of `1`s is even, the number of `0`s is divisible by 3, the overall length is divisible by 5, the binary value is divisible by 7, the binary value of the reversal is divisible by 11, and does not contain thirteen `1`s in a row. *[Hint: This is more tedious than difficult.]*

*(s) Strings w such that $\binom{|w|}{2} \bmod 6 = 4$.

⋆(t) Strings w such that $F_{\#(10,w)} \bmod 10 = 4$, where $\#(10, w)$ denotes the number of times 10 appears as a substring of w, and as usual F_n is the nth Fibonacci number:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

★(u) Strings w such that $F_{\#(1\cdots 0,w)} \bmod 10 = 4$, where $\#(1\cdots 0, w)$ denotes the number of times 10 appears as a *subsequence* of w, and as usual F_n is the nth Fibonacci number:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

2. (a) Let $L \subseteq 0^*$ be an arbitrary *unary* language. Prove that L^* is regular.
   (b) Prove that there is a binary language $L \subseteq (0+1)^*$ such that L^* is not regular.

3. Prove that none of the following languages is automatic.
   (a) $\{0^{n^2} \mid n \geq 0\}$
   (b) $\{0^{n^3} \mid n \geq 0\}$
   (c) $\{0^{f(n)} \mid n \geq 0\}$, where $f(n)$ is *any* fixed polynomial in n with degree at least 2.
   (d) $\{0^n \mid n \text{ is composite}\}$
   (e) $\{0^n 1 0^n \mid n \geq 0\}$
   (f) $\{0^m 1^n \mid m \neq n\}$
   (g) $\{0^m 1^n \mid m < 3n\}$
   (h) $\{0^{2n} 1^n \mid n \geq 0\}$
   (i) $\{w \in (0+1)^* \mid \#(0,w) = \#(1,w)\}$
   (j) $\{w \in (0+1)^* \mid \#(0,w) < \#(1,w)\}$
   (k) $\{0^m 1^n \mid m/n \text{ is an integer}\}$
   (l) $\{0^m 1^n \mid m \text{ and } n \text{ are relatively prime}\}$
   (m) $\{0^m 1^n \mid n - m \text{ is a perfect square}\}$
   (n) $\{w\#w \mid w \in (0+1)^*\}$
   (o) $\{ww \mid w \in (0+1)^*\}$
   (p) $\{w\#0^{|w|} \mid w \in (0+1)^*\}$
   (q) $\{w0^{|w|} \mid w \in (0+1)^*\}$
   (r) $\{xy \mid x, y \in (0+1)^* \text{ and } |x| = |y| \text{ but } x \neq y\}$
   (s) $\{0^m 1^n 0^{m+n} \mid m, n \geq 0\}$
   (t) $\{0^m 1^n 0^{mn} \mid m, n \geq 0\}$
   (u) Strings in which the substrings 00 and 11 appear the same number of times.

(v) Strings of the form $w_1\#w_2\#\cdots\#w_n$ for some $n \geq 2$, where $w_i \in \{0,1\}^*$ for every index i, and $w_i = w_j$ for some indices $i \neq j$.

(w) The set of all palindromes in $(0+1)^*$ whose length is divisible by 7.

(x) $\{w \in (0+1)^* \mid w \text{ is the binary representation of a perfect square}\}$

★(y) $\{w \in (0+1)^* \mid w \text{ is the binary representation of a prime number}\}$

4. For each of the following languages over the alphabet $\Sigma = \{0,1\}$, either prove that the language is regular (by constructing an appropriate DFA or regular expression) or prove that the language is not regular (using fooling sets). Recall that Σ^+ denotes the set of all *nonempty* strings over Σ. *[Hint: Believe it or not, most of these languages* ***are*** *actually regular.]*

(a) $\left\{0^n w 1^n \;\middle|\; w \in \Sigma^* \text{ and } n \geq 0\right\}$

(b) $\left\{0^n 1^n w \;\middle|\; w \in \Sigma^* \text{ and } n \geq 0\right\}$

(c) $\left\{w 0^n 1^n x \;\middle|\; w, x \in \Sigma^* \text{ and } n \geq 0\right\}$

(d) $\left\{0^n w 1^n x \;\middle|\; w, x \in \Sigma^* \text{ and } n \geq 0\right\}$

(e) $\left\{0^n w 1 x 0^n \;\middle|\; w, x \in \Sigma^* \text{ and } n \geq 0\right\}$

(f) $\left\{0^n w 0^n \;\middle|\; w \in \Sigma^+ \text{ and } n > 0\right\}$

(g) $\left\{w 0^n w \;\middle|\; w \in \Sigma^+ \text{ and } n > 0\right\}$

(h) $\left\{wxw \;\middle|\; w, x \in \Sigma^*\right\}$

(i) $\left\{wxw \;\middle|\; w, x \in \Sigma^+\right\}$

(j) $\left\{wxw^R \;\middle|\; w, x \in \Sigma^+\right\}$

(k) $\left\{wwx \;\middle|\; w, x \in \Sigma^+\right\}$

(l) $\left\{ww^R x \;\middle|\; w, x \in \Sigma^+\right\}$

(m) $\left\{wxwy \;\middle|\; w, x, y \in \Sigma^+\right\}$

(n) $\left\{wxw^R y \;\middle|\; w, x, y \in \Sigma^+\right\}$

(o) $\left\{xwwy \;\middle|\; w, x, y \in \Sigma^+\right\}$

(p) $\left\{xww^R y \;\middle|\; w, x, y \in \Sigma^+\right\}$

(q) $\left\{wxxw \;\middle|\; w, x \in \Sigma^+\right\}$

*(r) $\left\{wxw^R x \;\middle|\; w, x \in \Sigma^+\right\}$

(s) All strings w such that no prefix of w is a palindrome.

(t) All strings w such that no prefix of w with length at least 3 is a palindrome.

(u) All strings w such that no substring of w with length at least 3 is a palindrome.

(v) All strings w such that no prefix of w with positive even length is a palindrome.

(w) All strings w such that no substring of w with positive even length is a palindrome.

(x) Strings in which the substrings 00 and 11 appear the same number of times.

(y) Strings in which the substrings 01 and 10 appear the same number of times.

5. Let F and L be arbitrary infinite languages in $\{0, 1\}^*$.

   (a) Suppose for any two distinct strings $x, y \in F$, there is a string $w \in \Sigma^*$ such that $wx \in L$ and $wy \notin L$. (We can reasonably call w a *distinguishing* ***prefix*** for x and y.) Prove that L cannot be regular. *[Hint: The reversal of a regular language is regular.]*

   *(b) Suppose for any two distinct strings $x, y \in F$, there are two (possibly equal) strings $w, z \in \Sigma^*$ such that $wxz \in L$ and $wyz \notin L$. Prove that L cannot be regular.

The art of art, the glory of expression
and the sunshine of the light of letters is simplicity.
Nothing is better than simplicity . . . .
nothing can make up for excess or for the lack of definiteness.

— Walt Whitman, Preface to *Leaves of Grass* (1855)

Freedom of choice
Is what you got.
Freedom from choice
Is what you want.

— Devo, "Freedom of Choice", *Freedom of Choice* (1980)

4 Nondeterminism

4.1 Nondeterministic State Machines

The following diagram shows something that looks like a finite-state machine over the alphabet {0, 1}, but on closer inspection, it is not consistent with our earlier definitions. On one hand, there are two transitions out of s for each input symbol. On the other hand, states a and b are each missing an outgoing transition.

A nondeterministic finite-state automaton

Nevertheless, there is a sense in which this machine "accepts" the set of all strings that contain either 00 or 11 as a substring. Imagine that when the machine reads a symbol in state s, it makes a ***choice*** about which transition to follow. If the input string contains the substring 00, then it is *possible* for the machine to end in the accepting state c, by *choosing* to move into state a when it reads a 0 immediately before another 0. Similarly, if the input string contains the substring 11, it is *possible* for the machine to end in the accepting state c. On the other hand, if the input string does not contain either 00 or 11—or in other words, if the input alternates between 0 and 1—there are no choices that lead the machine to the accepting state. If the machine incorrectly chooses to transition to state a and then reads a 1, or transitions to b and then reads 0, it explodes; the only way to avoid an explosion is to stay in state s.

This object is an example of a ***nondeterministic finite-state automaton***, or ***NFA***, so named because its behavior is not uniquely *determined* by the input string. Formally, every NFA has five components:

- An arbitrary finite set Σ, called the ***input alphabet***.
- Another arbitrary finite set Q, whose elements are called ***states***.
- An arbitrary ***transition*** function $\delta : Q \times \Sigma \to 2^Q$.
- A ***start state*** $s \in Q$.

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

- A subset $\boldsymbol{A} \subseteq Q$ of ***accepting states***.

The only difference from the formal definition of *deterministic* finite-state automata is the domain of the transition function. In a DFA, the transition function always returns a single state; in an NFA, the transition function returns a *set* of states, which could be empty, or all of Q, or anything in between.

Just like DFAs, the behavior of an NFA is governed by an ***input string*** $w \in \Sigma^*$, which the machine reads one symbol at a time, from left to right. Unlike DFAs, however, an NFA does not maintain a single current state, but rather a *set* of current states. Whenever the NFA reads a symbol a, its set of current states changes from C to $\delta(C, a) := \bigcup_{q \in C} \delta(q, a)$. After all symbols have been read, the NFA ***accepts*** w if its current state set contains *at least one* accepting state and ***rejects*** w otherwise. In particular, if the set of current states ever becomes empty, it will stay empty forever, and the NFA will reject.

More formally, we define the function $\delta^* \colon Q \times \Sigma^* \to 2^Q$ that transitions on *strings* as follows:

$$\delta^*(q, w) := \begin{cases} \{q\} & \text{if } w = \varepsilon, \\ \displaystyle\bigcup_{r \in \delta(q,a)} \delta^*(r, x) & \text{if } w = ax. \end{cases}$$

The NFA $(Q, \Sigma, \delta, s, A)$ ***accepts*** $w \in \Sigma^*$ if and only if $\delta^*(s, w) \cap A \neq \varnothing$.

We can equivalently define an NFA as a directed graph whose vertices are the states Q, whose edges are labeled with symbols from Σ. We no longer require that every vertex has exactly one outgoing edge with each label; it may have several such edges or none. An NFA accepts a string w if the graph contains *at least one* walk from the start state to an accepting state whose label is w.

★★★

It's arguably more natural to an arbitrary set of start states $S \subseteq Q$ instead of just one. Then an NFA accepts a string w if and only if there is a sequence of transitions consistent with w from *some* start state to *some* accepting state, or more formally if $\delta^*(S, q) \cap A \neq \varnothing$. Change the definition and chase through all the theorems? Or prove equivalence and bounce back and forth, like we already do for ε-transitions?

4.2 Intuition

There are at least three useful ways to think about non-determinism.

Clairvoyance. Whenever an NFA reads symbol a in state q, it *chooses* the next state from the set $\delta(q, a)$, always *magically* choosing a state that leads to the NFA accepting the input string, unless no such choice is possible. As the BSD fortune file put it, "Nondeterminism means never having to say you're wrong."[1] Of course real machines can't actually look into the future; that's why I used the word "magic".

Parallel threads. An arguably more "realistic" view is that when an NFA reads symbol a in state q, it spawns an independent execution thread for each state in $\delta(q, a)$. In particular, if $\delta(q, a)$ is empty, the current thread simply dies. The NFA accepts if *at least one* thread is in an accepting state after it reads the last input symbol.

[1] This sentence is a riff on a horrible aphorism that was (sadly) popular in the US in the 70s and 80s. Fortunately, everyone seems to have forgotten the original saying, except maybe for that one time it was mocked on *The Simpsons*. Ah, who am I kidding? Nobody remembers *The Simpsons* either.

Equivalently, we can imagine that when an NFA reads symbol a in state q, it branches into several parallel universes, one for each state in $\delta(q, a)$. If $\delta(q, a)$ is empty, the NFA destroys the universe (including itself). Similarly, if the NFA finds itself in a non-accepting state when the input ends, the NFA destroys the universe. Thus, when the input is gone, only universes in which the NFA somehow chose a path to an accept state still exist. One slight disadvantage of this metaphor is that if an NFA reads a string that is not in its language, it destroys *all* universes.

Proofs/oracles. Finally, we can treat NFAs not as a mechanism for *computing* something, but as a mechanism for *verifying proofs*. If we want to *prove* that a string w contains one of the suffixes `00` or `11`, it suffices to demonstrate a single walk in our example NFA that starts at s and ends at c, and whose edges are labeled with the symbols in w. Equivalently, whenever the NFA faces a nontrivial choice, the prover can simply tell the NFA which state to move to next.

This intuition can be formalized as follows. Consider a *deterministic* finite state machine whose input alphabet is the product $\Sigma \times \Omega$ of an ***input*** alphabet Σ and an ***oracle*** alphabet Ω. Equivalently, we can imagine that this DFA reads simultaneously from two strings of the same length: the *input* string w and the *oracle* string ω. In either formulation, the transition function has the form $\delta : Q \times (\Sigma \times \Omega) \to Q$. As usual, this DFA accepts the pair $(w, \omega) \in (\Sigma \times \Omega)^*$ if and only if $\delta^*(s, (w, \omega)) \in A$. Finally, M ***nondeterministically accepts*** the string $w \in \Sigma^*$ if there is an oracle string $\omega \in \Omega^*$ with $|\omega| = |w|$ such that $(w, \omega) \in L(M)$.

4.3 ε-Transitions

It is fairly common for NFAs to include so-called ***ε-transitions***, which allow the machine to change state without reading an input symbol. An NFA with ε-transitions accepts a string w if and only if there is a sequence of transitions $s \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_\ell} q_\ell$ where the final state q_ℓ is accepting, each a_i is either ε or a symbol in Σ, and $a_1 a_2 \cdots a_\ell = w$.

For example, consider the following NFA with ε-transitions. (For this example, we indicate the ε-transitions using large red arrows; we won't normally do that.) This NFA deliberately has more ε-transitions than necessary.

A (rather silly) NFA with ε-transitions

The NFA starts as usual in state s. If the input string is `100111`, the the machine might non-deterministically choose the following transitions and then accept.

$$s \xrightarrow{\varepsilon} s \xrightarrow{1} s \xrightarrow{\varepsilon} d \xrightarrow{\varepsilon} a \xrightarrow{0} b \xrightarrow{0} c \xrightarrow{\varepsilon} d \xrightarrow{1} e \xrightarrow{1} f \xrightarrow{\varepsilon} e \xrightarrow{1} f \xrightarrow{\varepsilon} c \xrightarrow{\varepsilon} g$$

More formally, the transition function in an NFA with ε-transitions has a slightly larger domain $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$. The ***$\varepsilon$-reach*** of a state $q \in Q$ consists of all states r that satisfy one of the following conditions:

- either $r = q$,

- or $r \in \delta(q', \varepsilon)$ for some state q' in the ε-reach of q.

In other words, r is in the ε-reach of q if there is a (possibly empty) sequence of ε-transitions leading from q to r. For example, in the example NFA above, the ε-reach of state f is $\{a, c, d, f, g\}$.

Now we redefine the extended transition function $\delta^* : Q \times \Sigma^* \to 2^Q$, which transitions on arbitrary strings, as follows:

$$\delta^*(p, w) := \begin{cases} \varepsilon\text{-}reach(p) & \text{if } w = \varepsilon, \\ \displaystyle\bigcup_{r \in \varepsilon\text{-}reach(p)} \bigcup_{q \in \delta(r,a)} \delta^*(q, x) & \text{if } w = ax. \end{cases}$$

If we abuse notation by writing $\delta(S, a) = \bigcup_{q \in S} \delta(q, a)$ and $\delta^*(S, w) = \bigcup_{q \in S} \delta^*(q, w)$ and $\varepsilon\text{-}reach(S) = \bigcup_{q \in S} \varepsilon\text{-}reach(q)$ for any subset of states $S \subseteq Q$, this definition simplifies as follows:

$$\delta^*(p, w) := \begin{cases} \varepsilon\text{-}reach(p) & \text{if } w = \varepsilon, \\ \delta^*(\delta(\varepsilon\text{-}reach(p), a), x) & \text{if } w = ax. \end{cases}$$

Finally, as usual, an NFA with ε-transitions accepts a string w if and only if $\delta^*(s, w)$ contains at least one accepting state.

Although it may appear at first that ε-transitions give us a more powerful set of machines, NFAs with and without ε-transitions are actually equivalent. Given an NFA $M = (\Sigma, Q, s, A, \delta)$ with ε-transitions, we can construct an equivalent NFA $M' = (\Sigma, Q', s', A', \delta')$ without ε-transitions as follows:

$$\begin{aligned} Q' &:= Q \\ s' &= s \\ A' &= \left\{ q \in Q \;\middle|\; \varepsilon\text{-}reach(q) \cap A \neq \varnothing \right\} \\ \delta'(q, a) &= \delta(\varepsilon\text{-}reach(q), a) \end{aligned}$$

Straightforward definition-chasing now implies that M and M' accept exactly the same language. Thus, whenever we reason about or design NFAs, we are free to either allow or forbid ε-transitions, whichever is more convenient for the task at hand.

For example, our previous NFA with ε-transitions can be transformed into an equivalent NFA without ε-transitions, as shown in the figure below. The NFA on the right has two unreachable states a and d, but whatever.

A (rather silly) NFA with ε-transitions, and an equivalent NFA without ε-transitions

★★★

This reduction might be easier to understand incrementally.

- For every transition pair $p \xrightarrow{\varepsilon} q \xrightarrow{a} r$, add a direct transition $p \xrightarrow{a} r$. This addition does not change the accepted language.
- For each transition $p \xrightarrow{\varepsilon} q$ where q is an accepting state, make p an accepting state. This modification does not change the accepted language.
- When no more of the previous modifications are possible, delete all ε-transitions. This modification does not change the accepted language.

4.4 Kleene's Theorem

We are now finally in a position to prove the following fundamental fact, first observed by Steven Kleene in 1951:

Theorem 4.1. *A language L can be described by a regular expression if and only if L is the language accepted by a DFA.*

We will prove Kleene's fundamental theorem in four stages:

- Every DFA can be transformed into an equivalent NFA.
- Every NFA can be transformed into an equivalent DFA.
- Every regular expression can be transformed into an equivalent NFA.
- Every NFA can be transformed into an equivalent regular expression.

The first of these four transformations is completely trivial; a DFA is just a special type of NFA where the transition function always returns a single state. Unfortunately, the other three transformations require a bit more work.

4.5 NFA to DFA: The Subset Construction

In the parallel-thread model of NFA execution, an NFA does not have a single current state, but rather a *set* of current states. The evolution of this set of states is *determined* by a modified transition function $\delta' \colon 2^Q \times \Sigma \to 2^Q$, defined by setting $\delta'(P, a) := \bigcup_{p \in P} \delta(p, a)$ for any set of states $P \subseteq Q$ and any symbol $a \in \Sigma$. When the NFA finishes reading its input string, it accepts if and only if the current set of states intersects the set A of accepting states.

This formulation makes the NFA completely deterministic! We have just shown that any NFA $M = (\Sigma, Q, s, A, \delta)$ is equivalent to a DFA $M' = (\Sigma, Q', s', A', \delta')$ defined as follows:

$$\begin{aligned} Q' &:= 2^Q \\ s' &:= \{s\} \\ A' &:= \big\{ S \subseteq Q \bigm| S \cap A \neq \varnothing \big\} \\ \delta'(q', a) &:= \bigcup_{p \in q'} \delta(p, a) \qquad \text{for all } q' \subseteq Q \text{ and } a \in \Sigma. \end{aligned}$$

Similarly, any NFA with ε-transitions is equivalent to a DFA defined as follows:

$$\begin{aligned} Q' &:= 2^Q \\ s' &:= \{s\} \end{aligned}$$

$$A' := \{S \subseteq Q \mid \varepsilon\text{-}reach(S) \cap A \neq \varnothing\}$$
$$\delta'(q', a) := \bigcup_{p \in q'} \bigcup_{r \in \varepsilon\text{-}reach(p)} \delta(r, a) \qquad \text{for all } q' \subseteq Q \text{ and } a \in \Sigma.$$

This conversion from NFA to DFA is often called the ***subset construction***, but that name is somewhat misleading; it's not a "construction" so much as a change in perspective.

For example, the subset construction converts the 4-state NFA on the first page of this note into the following 16-state DFA. To simplify notation, I've named each DFA state using a simple string, omitting the braces and commas from the corresponding subset of NFA states; for example, DFA state *sbc* corresponds to the subset $\{s, b, c\}$ of NFA states.

The 16-state DFA obtained from our first 4-state NFA by the subset construction.
Only the five yellow states are reachable from the start state.

An obvious disadvantage of this "construction" is that it (usually) leads to DFAs with far more states than necessary, in part because many states cannot even be reached from the start state. In the example above, there are eleven unreachable states; only five states are reachable from s.

Incremental Subset Construction

Instead of building the entire subset DFA and then discarding the unreachable states, we can avoid the unreachable states from the beginning by constructing the DFA incrementally, essentially by performing a breadth-first search of the DFA graph.

To execute this algorithm by hand, we prepare a table with $|\Sigma| + 3$ columns, with one row for each DFA state we discover. In order, these columns record the following information:

- The DFA state (as a subset of NFA states)
- The ε-reach of the corresponding subset of NFA states
- Whether the DFA state is accepting (that is, whether the ε-reach intersects A)
- The output of the transition function for each symbol in Σ.

We start with DFA-state $\{s\}$ in the first row and first column. Whenever we discover an unexplored state in one of the last $|\Sigma|$ columns, we copy it to the left column in a new row. To reduce notational clutter, we write all subsets of NFA states without braces or commas.

For example, given the NFA with ε-transitions from Section 4.3, the standard subset construction would produce a DFA with 256 states, but the incremental subset construction produces an nine-state DFA, described by the following table and illustrated on the next page. We would fill in the first row, for the starting DFA state s, as follows:

- The ε-reach of NFA state s is $\{s, a, d\}$, so we write *sad* in the first column.
- None of the NFA states $\{s, a, d\}$ is an accepting state, so $\{s\}$ is not an accepting state of the DFA, so we do *not* check the second column.
- Next, $\delta'(\{s, a, d\}, 0) = \delta(s, 0) \cup \delta(a, 0) \cup \delta(d, 0) = \{s\} \cup \{b\} \cup \varnothing = \{s, b\}$, so we write *sb* in the third column. Because *sb* does not already appear in the first column in any existing row, we have discovered a new DFA state! We start a new row for DFA state *sb*.
- Finally, $\delta'(\{s, a, d\}, 1) = \delta(s, 1) \cup \delta(a, 1) \cup \delta(d, 1) = \{s\} \cup \varnothing \cup \{e\} = \{s, e\}$, so we write *se* in the fourth column, and we start a new row for the new DFA state *se*.

We now have two new rows to fill in, corresponding to states *sb* and *se*. The algorithm continues filling in rows (and discovering new rows) until all rows are filled, ending with the following table:

q'	ε-reach(q')	$q' \in A'$?	$\delta'(q', 0)$	$\delta'(q', 1)$
s	*sad*		*sb*	*se*
sb	*sabd*		*sbc*	*se*
se	*sade*		*sb*	*sef*
sbc	*sabcdg*	✓	*sbcg*	*seg*
sef	*sacdefg*	✓	*sbg*	*sefg*
sbcg	*sabcdg*	✓	*sbcg*	*seg*
seg	*sadeg*	✓	*sbg*	*sefg*
sbg	*sabdg*	✓	*sbcg*	*seg*
sefg	*sacdefg*	✓	*sbg*	*sefg*

An eight-state NFA with ε-transitions, and the output of the incremental subset construction for that NFA.

Although it avoids unreachable states, the incremental subset algorithm still gives us a DFA with far more states than necessary, intuitively because it keeps looking for `00` and `11` substrings even after it's already found one. After all, after the NFA finds both `00` and `11` as substrings, it doesn't kill all the other parallel execution threads, because it *can't*. NFAs often have significantly fewer states than equivalent DFAs, but that efficiency also makes them kind of stupid.

4.6 Regular Expression to NFA: Thompson's Algorithm

We now turn to the core of Kleene's theorem, which claims that regular languages (described by regular expressions) and automatic languages (accepted by finite-state automata) are the same.

Lemma 4.2. *Every regular language is accepted by a nondeterministic finite-state automaton.*

Proof: In fact, we will prove the following stronger claim: Every regular language is accepted by an NFA *with exactly one accepting state, which is different from its start state*. The following construction was first described by Ken Thompson in 1968. Thompson's algorithm actually proves a stronger statement: For any regular language L, there is an NFA that accepts L that has exactly one accepting state t, which is distinct from the starting state s.

Let R be an arbitrary regular expression over an arbitrary finite alphabet Σ. Assume that for any sub-expression S of R, the language described by S is accepted by an NFA with one accepting state distinct from its start state, which we denote pictorially by . There are six cases to consider—three base cases and three recursive cases—mirroring the recursive definition of a regular expression.

- If $R = \varnothing$, then $L(R) = \varnothing$ is accepted by the trivial NFA .
- If $R = \varepsilon$, then $L(R) = \{\varepsilon\}$ is accepted by a different trivial NFA .
- If $R = a$ for some symbol $a \in \Sigma$, then $L(R) = \{a\}$ is accepted by the NFA . (The case where R is a single *string* with length greater than 1 reduces to the single-symbol case by concatenation, as described in the next case.)
- Suppose $R = ST$ for some regular expressions S and T. The inductive hypothesis implies that the languages $L(S)$ and $L(T)$ are accepted by NFAs and , respectively. Then $L(R) = L(ST) = L(S) \bullet L(T)$ is accepted by the NFA , built by connecting the two component NFAs in series.
- Suppose $R = S + T$ for some regular expressions S and T. The inductive hypothesis implies that the language $L(S)$ and $L(T)$ are accepted by NFAs and , respectively. Then $L(R) = L(S + T) = L(S) \cup L(T)$ is accepted by the NFA , built by connecting the two component NFAs in parallel with new start and accept states.
- Finally, suppose $R = S^*$ for some regular expression S. The inductive hypothesis implies that the language $L(S)$ is accepted by an NFA . Then the language $L(R) = L(S^*) = L(S)^*$ is accepted by the NFA .

In all cases, the language $L(R)$ is accepted by an NFA with one accepting state, which is different from its start state, as claimed. □

As an example, given the regular expression $(0 + 10^*1)^*$ of strings containing an even number of 1s, Thompson's algorithm produces a 14-state NFA shown on the next page. As this example shows, Thompson's algorithm tends to produce NFAs with many redundant states. Fortunately, just as there are for DFAs, there are algorithms that can reduce any NFA to an equivalent NFA with the smallest possible number of states.

Interestingly, applying the incremental subset algorithm to Thompson's NFA tends to yield a DFA with relatively *few* states, in part because the states in Thompson's NFA tend to have large ε-reach, and in part because relatively few of those states are the targets of non-ε-transitions. Starting with the Thompson's NFA for $(0 + 10^*1)^*$, for example, the incremental subset construction yields a DFA with just five states.

The NFA constructed by Thompson's algorithm for the regular expression $(0 + 10^*1)^*$.
The four non-ε-transitions are drawn with with bold red arrows for emphasis.

q'	ε-reach(q')	$q' \in A'$?	$\delta'(q', 0)$	$\delta'(q', 1)$
s	*sabjm*	✓	k	c
k	*sabjklm*	✓	k	c
c	*cdegh*		f	i
f	*defgh*		f	i
i	*sabjilm*	✓	k	c

The DFA computed by the incremental subset algorithm from Thompson's NFA for $(0 + 10^*1)^*$.

This DFA can be further simplified to just two states, by observing that all three accepting states are equivalent, and that both non-accepting states are equivalent. But still, five states is pretty good, especially compared with the $2^{14} = 16384$ states that the naïve subset construction would yield!

4.7 Another Example

Here is another example of all the algorithms we've seen so far, starting with the regular expression $(0 + 1)^*(00 + 11)(0 + 1)^*$, which describes the language accepted by our very first example NFA. Thompson's algorithm constructs the following 18-state monster:

Thompson's NFA for the regular expression $(0 + 1)^*(00 + 11)(0 + 1)^*$,
with the ε-reach of the start state s highlighted.

Given this NFA as input, the incremental subset construction computes the following table, leading to a DFA with just nine states. Yeah, the ε-reaches get a bit ridiculous; unfortunately, this *is* typical for Thompson's NFA. As usual, the resulting DFA has far more states than necessary.

q'	ε-reach(q')	$q' \in A'$?	$\delta'(q', 0)$	$\delta'(q', 1)$
s	*sabdghim*		*cj*	*en*
cj	*sabdfghijkm*		*cjl*	*en*
en	*sabdfghmno*		*cj*	*enp*
cjl	*sabdfghijklmqrtuwz*	✓	*cjlv*	*enx*
enp	*sabdfghmnopqrtuwz*	✓	*cjv*	*enpx*
cjlv	*sabdfghijklmqrtuvwyz*	✓	*cjlv*	*enx*
enx	*sabdfghmnopqrtuwxyz*	✓	*cjv*	*enpx*
cjv	*sabdfghijkmrtuvwyz*	✓	*cjlv*	*enx*
enpx	*sabdfghmnopqrtuwxyz*	✓	*cjv*	*enpx*

The DFA computed by the incremental subset algorithm from Thompson's NFA for $(0+1)^*(00+11)(0+1)^*$.

Finally, the DFA-minimization algorithm from the previous lecture note correctly discovers that all six accepting states of the incremental-subset DFA are equivalent, and thus reduces the DFA to just four states.

The minimal DFA that accepts the language $(0+1)^*(00+11)(0+1)^*$.

*4.8 NFA to Regular Expression: Han and Wood's Algorithm

The only component of Kleene's theorem left to prove is that every language accepted by an NFA is regular. I'll describe a relatively recent proof, due Yo-Sub Han and Derick Wood in 2005[2], that is morally equivalent to Kleene's 1951 argument, but uses more modern standard notation.

Recall that a standard NFA can be represented by a state-transition graph, whose vertices are the states and whose edges represent possible transitions. Each edge is labeled with a single symbol in Σ. A string $w \in \Sigma^*$ is accepted if and only if there is a sequence of transitions

$$s \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_\ell} q_\ell$$

[2]Yo-Sub Han* and Derick Wood. The generalization of generalized automata: Expression automata. *International Journal of Foundations of Computer Science* 16(3):499–510, 2005.

where the final state q_ℓ is accepting and $a_1 a_2 \cdots a_\ell = w$.

We've already seen that NFAs can be generalized to include ε-transitions; we can push this generalization further. A ***string NFA*** allows each transition $p \to q$ to be labeled with an *arbitrary string* $x(p \to q) \in \Sigma^*$. We are allowed to transition from state p to state q if the label $x(p \to q)$ is a *prefix* of the remaining input. Thus, a string $w \in \Sigma^*$ is accepted if and only if there is a sequence of transitions

$$s \xrightarrow{x_1} q_1 \xrightarrow{x_2} q_2 \xrightarrow{x_3} \cdots \xrightarrow{x_\ell} q_\ell$$

where the final state q_ℓ is accepting, and $x_1 \bullet x_2 \bullet \cdots \bullet x_\ell = w$. Thus, an NFA with ε-transitions is just a string NFA where every label has length 0 or 1. Any string NFA can be converted into an equivalent standard NFA, by subdividing each edge $p \to q$ into a path of length $|x(p \to q)|$ (unless $x(p \to q) = \varepsilon$).

Finally, Han and Wood define an ***expression automaton*** as a finite-state machine where each transition $p \to q$ is labeled with an arbitrary *regular expression* $R(p \to q)$. We can transition from state p to state q if *any* prefix of the remaining input matches the regular expression $R(p \to q)$. Thus, a string $w \in \Sigma^*$ is accepted by an expression automaton if and only if there is a sequence of transitions

$$s \xrightarrow{R_1} q_1 \xrightarrow{R_2} q_2 \xrightarrow{R_3} \cdots \xrightarrow{R_\ell} q_\ell$$

where the final state q_ℓ is accepting, and we can write $w = x_1 \bullet x_2 \bullet \cdots \bullet x_\ell = w$, where each substring x_i matches the corresponding regular expression R_i.

More formally, an expression automaton consists of the following components:

- A finite set Σ called the ***input alphabet***
- Another finite set Q whose elements are called ***states***
- A unique ***start state*** $s \in Q$
- A unique ***target state*** $t \in Q \setminus \{s\}$
- A ***transition function*** $R\colon (Q \setminus \{t\}) \times (Q \setminus \{s\}) \to \mathit{Reg}(\Sigma)$, where $\mathit{Reg}(\Sigma)$ is the set of regular expressions over Σ.

The requirement that the start and target states are unique and disatinct is not essential to the model. We impose this requirement for convenience of the equivalence proof; it can be easily enforced using ε-transitions.

Expression automata are even more nondeterministic than NFAs. A single string could match several (even infinitely many) transition sequences from s to t, and it could match each of those sequences in several (even infinitely many) different ways. A string w is accepted if *any* decomposition of w into a sequence of substrings matches *any* sequence of transitions from s to t. Conversely, a string might match *no* state sequences, in which case the string is rejected.

Two extreme special cases of expression automata are already familiar. First, every regular language is clearly the language of an expression automaton with exactly two states. Second, with only minor modifications, any DFA or NFA can be converted into an expression automaton with trivial transition expressions. Thompson's algorithm can be used to transform any expression automaton into a standard NFA (with ε-transitions), by recursively expanding any nontrivial transition expression. To complete the proof of Kleene's theorem, we show how to convert an arbitrary expression automaton into a regular expression, by repeatedly deleting vertices.

Lemma 4.3. *Every expression automaton accepts a regular language.*

Proof: Let $E = (Q, \Sigma, R, s, t)$ be an arbitrary expression automaton. Assume that any expression automaton with fewer states than E accepts a regular language. There are two cases to consider, depending on the number of states in Q:

- If $Q = \{s, t\}$, then trivially, E accepts the regular language $R(s \to t)$.
- On the other hand, suppose Q has more than two states; fix an arbitrary state $q \in Q \setminus \{s, t\}$. We modify the automaton, without changing its language, so that state q is redundant and can be removed. Define a new transition function $R' : Q \times Q \to Reg(\Sigma)$ by setting

$$R'(p \to r) \;:=\; R(p \to r) + R(p \to q) R(q \to q)^* R(q \to r).$$

  With this modified transition function in place, any string w that matches the sequence $p \to q \to q \to \cdots \to q \to r$ with any number of q's also matches the single transition $p \to r$. Thus, by induction, if w matches a sequence of states, it also matches the subsequence obtained by removing all q's. Let E' be the expression automaton with states $Q' = Q \setminus \{q\}$ that uses this modified transition function R'. This new automaton accepts exactly the same strings as the original automaton E. Because E' has fewer states than E, the inductive hypothesis implies E' accepts a regular language.

In both cases, we conclude that E accepts a regular language. □

This proof can be mechanically translated into an algorithm to convert any DFA or NFA into an equivalent regular expression, via a sequence of expression automata with fewer and fewer states, but increasingly complex transition expressions.

One step in Kleene's/Han and Wood's reduction algorithm.

The figure on the next page shows Han and Wood's algorithm in action, starting with a DFA that accepts the binary representations of non-negative integers divisible by 3, possibly with extra leading `0`s. (State i means the binary number we've read so far is congruent to i mod 3.) First we convert the DFA into an expression automaton by adding a new accept state. (We don't need to add a new start state, because there are no transitions the original start state s.) Then we remove state'2, then state 0, and finally state 1, updating the transition expressions between any remaining states at each iteration. For the sake of clarity, edges $p \to q$ with $R(p \to q) = \varnothing$ are omitted from the figures. The final regular expression $\mathtt{00}^* + (\mathtt{00}^*\mathtt{1} + \mathtt{1})(\mathtt{10}^*\mathtt{1} + \mathtt{01}^*\mathtt{0})^*\mathtt{10}^*$ can be slightly simplified to $\mathtt{0}^*\mathtt{0} + \mathtt{0}^*\mathtt{1}(\mathtt{10}^*\mathtt{1} + \mathtt{01}^*\mathtt{0})^*\mathtt{10}^*$, which is precisely the regular expression we gave for this language back in Lecture Note 2!

Given an NFA with n states (including s and t), Han and Wood's algorithm iteratively removes $n-2$ states, updating $O(n^2)$ transition expressions in each iteration. If the concatenation and Kleene star operations could be performed in constant time, the resulting algorithm would run in $O(n^3)$ time. However, in the worst case, the transition expressions grows in length by roughly a factor of 4 in each iteration, so the final expression has length $\Theta(4^n)$. If we insist on representing the expressions as explicit strings, the worst-case running time is actually $\Theta(4^n)$.

Converting a DFA into an equivalent regular expression using Han and Wood's algorithm.

4.9 Regular Language Transformations

We have already seen that many functions of regular languages are themselves regular: unions, concatenations, and Kleene closure by definition. and intersections and differences by product construction on DFA. However, the set of regular languages is closed under a much richer class of functions.

Suppose we wanted to prove that regular languages are closed under some function f; that is, for every regular language L, we want to prove that the language $f(L)$ is also regular. There are two general techniques to prove such a statement:

- Describe an algorithm that transforms an arbitrary regular expression R into a new regular expression R' such that $L(R') = f(L(R))$.
- Describe an algorithm that transforms an arbitrary DFA M into a new NFA M' such that $L(M') = f(L(M))$.

The equivalence between regular expressions and finite automata implies that *in principle* we can always use either technique, but in practice, the second one is far more powerful and usually simpler. The asymmetry in the second technique is important. We start with a DFA for L to impose as much structure as possible in the input; we aim for an NFA with ε-transitions to give ourselves as much freedom as possible in the output.[3]

For our first example, I'll describe proofs using both techniques.

Lemma 4.4. *For any regular language L, the language $L^R = \{w^R \mid w \in L\}$ is also regular.*

Proof (regular expression to regular expression): Let R be an arbitrary regular expression such that $L = L(R)$. Assume for any proper subexpression S of R that $L(S)^R$ is regular. There are five cases to consider, mirroring the recursive definition of regular expressions:

- If $R = \varnothing$, then $L^R = L = \varnothing$, so $L(R) = L^R$.
- Suppose R consists of a single word w. Let $R' = w^R$. Then $L(R') = \{w^R\} = L^R$.

[3]We could give ourselves even more freedom by constructing an *expression* automaton, but creativity thrives on constraint.

- Suppose $R = A + B$. The inductive hypothesis implies that there are regular expressions A' and B' such that $L(A') = L(A)^R$ and $L(B') = L(B)^R$. Let $R' = A' + B'$. Then $L(R') = L(A') \cup L(B') = L(A)^R \cup L(B)^R = (L(A) \cup L(B))^R = L^R$.
- Suppose $R = A \bullet B$. The inductive hypothesis implies that there are regular expressions A' and B' such that $L(A') = L(A)^R$ and $L(B') = L(B)^R$. Let $R' = B' \bullet A'$. Then $L(R') = L(B') \bullet L(A') = L(B)^R \bullet L(A)^R = (L(A) \bullet L(B))^R = L^R$.
- Finally, suppose $R = A^*$. The inductive hypothesis implies that there is a regular expression A' such that $L(A') = L(A)^R$. Let $R' = (A')^*$. Then $L(R') = L(A')^* = (L(A)^R)^* = (L(A)^*)^R = L^R$.

In all cases, we have constructed a regular expression R' such that $L(R') = L^R$. We conclude that L^R is regular. □

Careful readers may be unsatisfied with the previous proof, because it assumes several "obvious" properties of string and language reversal. Specifically, for all strings x and y and all languages L and L', we assumed the following:

- $(x \bullet y)^R = y^R \bullet x^R$
- $(L \cdot L')^R = (L')^R \cdot L^R$.
- $(L \cup L')^R = L^R \cup (L')^R$.
- $(L^*)^R = (L^R)^*$.

All of these claims are all easy to prove by inductive definition-chasing.

Proof (DFA to NFA): Let $M = (\Sigma, Q, s, A, \delta)$ be an arbitrary DFA that accepts L. We construct an NFA $M^R = (\Sigma, Q^R, s^R, A^R, \delta^R)$ with ε-transitions that accepts L^R, intuitively by reversing every transition in M, and swapping the roles of the start state and the accepting states. Because M does not have a unique accepting state, we need to introduce a special start state s^R, with ε-transitions to each accepting state in M. These are the only ε-transitions in M^R.

$$
\begin{aligned}
Q^R &= Q \cup \{s^R\} \\
A^R &= \{s\} \\
\delta^R(s^R, \varepsilon) &= A \\
\delta^R(s^R, a) &= \varnothing && \text{for all } a \in \Sigma \\
\delta^R(q, \varepsilon) &= \varnothing && \text{for all } q \in Q \\
\delta^R(q, a) &= \big\{p \;\big|\; q \in \delta(p, a)\big\} && \text{for all } q \in Q \text{ and } a \in \Sigma
\end{aligned}
$$

Routine inductive definition-chasing now implies that the reversal of any sequence $q_0 \to q_1 \to \cdots \to q_\ell$ of transitions in M is a valid sequence $q_\ell \to q_{\ell-1} \to \cdots \to q_0$ of transitions in M^R. Because the transitions retain their labels (but reverse directions), it follows that M accepts any string w if and only if M^R accepts w^R.

We conclude that the NFA M^R accepts L^R, so L^R must be regular. □

Lemma 4.5. *For any regular language L, the language half$(L) := \{w \mid ww \in L\}$ is also regular.*

Proof: Let $M = (\Sigma, Q, s, A, \delta)$ be an arbitrary DFA that accepts L.

Intuitively, we construct an NFA M' that reads its input string w and simulates the original DFA M reading the input string ww. Our overall strategy has three parts:

- First M' non-deterministically *guesses* the state $h = \delta^*(s, w)$ that M will reach after reading input w. (We can't just run M on input w to compute the correct state h, because that would consume the input string!)
- Then M' runs two copies of M in parallel (using a product construction): a "left" copy starting at s and a "right" copy starting at the (guessed) halfway state h.
- Finally, when M' is done reading w, it accepts if and only if the first copy of M actually stopped in state h (so our initial guess was correct) and the second copy of M stopped in an accepting state. That is, M' accepts if and only if $\delta^*(s, w) = h$ and $\delta^*(h, w) \in A$.

To implement this strategy, M' needs to maintain *three* states of M: the state of the left copy of M, the guess h for the halfway state, and the state of the right copy of M. The first and third states evolve according to the transition function δ, but the second state never changes. Finally, to implement the non-deterministic guessing, M' includes a special start state s' with ε-transitions to every triple of the form (s, h, h).

Summing up, our new NFA $M' = (\Sigma, Q', s', A', \delta')$ is formally defined as follows.

$$
\begin{aligned}
Q' &= (Q \times Q \times Q) \cup \{s'\} \\
A' &= \big\{(h, h, q) \;\big|\; h \in Q \text{ and } q \in A\big\} \\
\delta'(s', \varepsilon) &= \big\{(s, h, h) \;\big|\; h \in Q\big\} \\
\delta'(s', a) &= \varnothing && \text{for all } a \in \Sigma \\
\delta'((p, h, q), \varepsilon) &= \varnothing && \text{for all } p, h, q \in Q \\
\delta'((p, h, q), a) &= \big\{(\delta(p, a), h, \delta(q, a))\big\} && \text{for all } p, h, q \in Q \text{ and } a \in \Sigma
\end{aligned}
$$

□

Exercises

1. For each of the following regular expressions, describe or draw two finite-state machines:
   - An NFA that accepts the same language, constructed using Thompson's algorithm.
   - An equivalent DFA, built from the previous NFA using the incremental subset construction. For each state in your DFA, identify the corresponding subset of states in your NFA. Your DFA should have no unreachable states.

   (a) $(\mathtt{01} + \mathtt{10})^*(\mathtt{0} + \mathtt{1} + \varepsilon)$
   (b) $(\varepsilon + \mathtt{1})(\mathtt{01})^*(\varepsilon + \mathtt{0})$
   (c) $\mathtt{1}^* + (\mathtt{10})^* + (\mathtt{100})^*$
   (d) $(\varepsilon + \mathtt{0} + \mathtt{00})(\mathtt{1} + \mathtt{10} + \mathtt{100})^*$
   (e) $((\mathtt{0} + \mathtt{1})(\mathtt{0} + \mathtt{1}))^*$
   (f) $\varepsilon + \mathtt{0}(\mathtt{0} + \mathtt{1})^* + \mathtt{1}(\mathtt{1} + \mathtt{0})^*$

2. The accepting language of an NFA $M = (\Sigma, Q, s, A, \delta)$ is defined as follows:

$$L(M) := \big\{w \in \Sigma^* \;\big|\; \delta^*(s, w) \cap A \neq \varnothing\big\}.$$

   Kleene's theorem (described here as Han and Wood's algorithm) implies that $L(M)$ is regular. Prove that the following languages associated with M are also regular:

(a) $L^{\forall}(M) := \{w \in \Sigma^* \mid A \subseteq \delta^*(s,w)\}$. That is, a string w is in the language $L^{\forall}(M)$ if and only if $\delta^*(s,w)$ contains *every* accepting states.

(b) $L^{\subseteq}(M) := \{w \in \Sigma^* \mid \delta^*(s,w) \subseteq A\}$. That is, a string w is in the language $L^{\subseteq}(M)$ if and only if $\delta^*(s,w)$ contains *only* accepting states.

(c) $L^{=}(M) := \{w \in \Sigma^* \mid \delta^*(s,w) = A\}$. That is, a string w is in the language $L^{=}(M)$ if and only if $\delta^*(s,w)$ is exactly the set of accepting states.

3. A certain professor who really should know better once woke up in the middle of the night with a startling revelation—Thompson's algorithm doesn't need all those ε-transitions! Filled with the certainty that only sleep deprivation can bring, he ran to his laptop and quickly changed two cases in his description of Thompson's algorithm.

- When $R = S \bullet T$, instead of connecting the accept state of to the start state of with an ε-transition, we can ***just*** identify those two states to build the simpler NFA !
- When $R = S^*$, instead of introducing two new states and four ε-transitions, we can ***just*** add two ε-transitions between the start and accept states of to build the simpler NFA !

Satisfied with his simplification, he thanked the penguin who gave him the idea, and then flew his hat back into the ocean marshmallows, where a giant man with the head of a dog gave him the power of bread. The next morning, while he was proudly teaching his new simplified proof for the first time, he realized his horrible mistake.

Prove that *neither* of the professor's optimizations is actually correct.

(a) Find a regular expression R, such that the NFA constructed from R by Thompson's algorithm ***with only the first modification*** accepts strings that are not in $L(R)$.

(b) Find a regular expression R, such that the NFA constructed from R by Thompson's algorithm ***with only the second modification*** accepts strings that are not in $L(R)$.

4. A ***Moore machine*** is a variant of a finite-state automaton that produces output; Moore machines are sometimes called finite-state *transducers*. For purposes of this problem, a Moore machine formally consists of six components:

- A finite set Σ called the input alphabet
- A finite set Γ called the output alphabet
- A finite set Q whose elements are called states
- A start state $s \in Q$
- A transition function $\delta : Q \times \Sigma \to Q$
- An output function $\omega : Q \to \Gamma$

More intuitively, a Moore machine is a graph with a special start vertex, where every node (state) has one outgoing edge labeled with each symbol from the input alphabet, and each node (state) is additionally labeled with a symbol from the output alphabet.

The Moore machine reads an input string $w \in \Sigma^*$ one symbol at a time. For each symbol, the machine changes its state according to the transition function δ, and then outputs the symbol $\omega(q)$, where q is the new state. Formally, we recursively define a *transducer* function $\omega^* : \Sigma^* \times Q \to \Gamma^*$ as follows:

$$\omega^*(w,q) = \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ \omega(\delta(a,q)) \cdot \omega^*(x, \delta(a,q)) & \text{if } w = ax \end{cases}$$

Given the input string $w \in \Sigma^*$, the machine outputs the string $\omega^*(w,s) \in \Gamma^*$. To simplify notation, we define $M(w) = \omega^*(w,s)$.

Finally, the ***output language $L^\circ(M)$*** of a Moore machine M is the set of all strings that the machine can output:

$$L^\circ(M) := \left\{ M(w) \;\middle|\; w \in \Sigma^* \right\}$$

(a) Let M be an arbitrary Moore machine. Prove that $L^\circ(M)$ is a regular language.

(b) Let M be an arbitrary Moore machine whose input alphabet Σ and output alphabet Γ are identical. Prove that the language

$$L^=(M) = \left\{ w \in \Sigma^* \;\middle|\; M(w) = w \right\}$$

is regular. Strings in $L^=(M)$ are also called *fixed points* of the function $M : \Sigma^* \to \Sigma^*$.

*(c) As in part (b), let M be an arbitrary Moore machine whose input and output alphabets are identical. Prove that the language $\left\{ w \in \Sigma^* \;\middle|\; M(M(w)) = w \right\}$ is regular.

[Hint: Parts (a) and (b) are easier than they look!]

5. A ***Mealy machine*** is a variant of a finite-state automaton that produces output; Mealy machines are sometimes called finite-state *transducers*. For purposes of this problem, a Mealy machine formally consists of six components:

- A finite set Σ called the input alphabet
- A finite set Γ called the output alphabet
- A finite set Q whose elements are called states
- A start state $s \in Q$
- A transition function $\delta : Q \times \Sigma \to Q$
- An output function $\omega : Q \times \Sigma \to \Gamma$

More intuitively, a Mealy machine is a graph with a special start vertex, where every node (state) has one outgoing edge labeled with each symbol from the input alphabet, and each edge (transition) is additionally labeled with a symbol from the output alphabet. (Mealy machines are closely related to *Moore* machines, which produce output at each *state* instead of at each transition.)

The Mealy machine reads an input string $w \in \Sigma^*$ one symbol at a time. For each symbol, the machine changes its state according to the transition function δ, and simultaneously outputs a symbol according the output function ω. Formally, we recursively define a *transducer* function $\omega^* : Q \times \Sigma^* \to \Gamma^*$ as follows:

$$\omega^*(q, w) = \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ \omega(q, a) \cdot \omega^*(\delta(q, a), x) & \text{if } w = ax \end{cases}$$

Given any input string $w \in \Sigma^*$, the machine outputs the string $\omega^*(w, s) \in \Gamma^*$. To simplify notation, we define $M(w) = \omega^*(w, s)$.

Finally, the ***output language $L^\circ(M)$*** of a Mealy machine M is the set of all strings that the machine can output:

$$L^\circ(M) := \{M(w) \mid w \in \Sigma^*\}$$

(a) Let M be an arbitrary Mealy machine. Prove that $L^\circ(M)$ is a regular language.

(b) Let M be an arbitrary Mealy machine whose input alphabet Σ and output alphabet Γ are identical. Prove that the language

$$L^=(M) = \{w \in \Sigma^* \mid w = \omega^*(s, w)\}$$

is regular. $L^=(M)$ consists of all strings w such that M outputs w when given input w; these are also called *fixed points* for the transducer function ω^*.

*(c) As in part (b), let M be an arbitrary Mealy machine whose input and output alphabets are identical. Prove that the language $\left\{w \in \Sigma^* \;\middle|\; M(M(w)) = w\right\}$ is regular.

[Hint: Parts (a) and (b) are easier than they look!]

6. Let $L \subseteq \Sigma^*$ be an arbitrary regular language. Prove that the following languages are regular. Assume $\# \in \Sigma$.

(a) $censor(L) := \left\{\#^{|w|} \;\middle|\; w \in L\right\}$

(b) $dehash(L) = \left\{dehash(w) \;\middle|\; w \in L\right\}$, where $dehash(w)$ is the subsequence of w obtained by deleting every $\#$.

(c) $insert\#(L) := \left\{x\#y \;\middle|\; xy \in L\right\}$.

(d) $delete\#(L) := \left\{xy \;\middle|\; x\#y \in L\right\}$.

(e) $prefix(L) := \{x \in \Sigma^* \mid xy \in L \text{ for some } y \in \Sigma^*\}$

(f) $suffix(L) := \{y \in \Sigma^* \mid xy \in L \text{ for some } x \in \Sigma^*\}$

(g) $substring(L) := \{y \in \Sigma^* \mid xyz \in L \text{ for some } x, z \in \Sigma^*\}$

(h) $superstring(L) := \{xyz \mid y \in L \text{ and } x, z \in \Sigma^*\}$

(i) $cycle(L) := \{xy \mid x, y \in \Sigma^* \text{ and } yx \in L\}$

(j) $prefmax(L) := \{x \in L \mid xy \in L \iff y = \varepsilon\}$.

(k) $sufmin(L) := \{xy \in L \mid y \in L \iff x = \varepsilon\}$.

(l) $minimal(L) := \{w \in L \mid \text{no proper substring of } w \text{ is in } L\}$.

(m) $maximal(L) := \{w \in L \mid \text{no proper superstring of } w \text{ is in } L\}$.

(n) $evens(L) := \{evens(w) \mid w \in L\}$, where $even(w)$ is the subsequence of w containing every even-indexed symbol. For example, $evens(\texttt{EVENINDEX}) = \texttt{VNNE}$.

(o) $evens^{-1}(L) := \{w \in \Sigma^* \mid evens(w) \in L\}$.

(p) $subseq(L) := \{x \in \Sigma^* \mid x \text{ is a subsequence of some } y \in L\}$

(q) $superseq(L) := \{x \in \Sigma^* \mid \text{some } y \in L \text{ is a subsequence of } x\}$

(r) $swap(L) := \{swap(w) \mid w \in L\}$, where $swap(w)$ is defined recursively as follows:

$$swap(w) = \begin{cases} w & \text{if } |w| \leq 1 \\ ba \bullet swap(x) & \text{if } w = abx \text{ for some } a, b \in \Sigma \text{ and } x \in \Sigma^* \end{cases}$$

(s) $oneswap(L) := \left\{xbay \;\middle|\; xaby \in L \text{ where } a, b \in \Sigma \text{ and } x, y \in \Sigma^*\right\}$.

(t) $left(L) := \{x \in \Sigma^* \mid xy \in L \text{ for some } y \in \Sigma^* \text{ where } |x| = |y|\}$

(u) $right(L) := \{y \in \Sigma^* \mid xy \in L \text{ for some } x \in \Sigma^* \text{ where } |x| = |y|\}$

(v) $middle(L) := \{y \in \Sigma^* \mid xyz \in L \text{ for some } x, z \in \Sigma^* \text{ where } |x| = |y| = |z|\}$

(w) $halfseq(L) := \{w \in \Sigma^* \mid w \text{ is a subsequence of some string } x \in L \text{ where } |x| = 2 \cdot |w|\}$

(x) $third(L) := \{w \in \Sigma^* \mid www \in L\}$

(y) $palin(L) := \left\{w \in \Sigma^* \;\middle|\; ww^R \in L\right\}$

(z) $drome(L) := \left\{w \in \Sigma^* \;\middle|\; w^R w \in L\right\}$

7. Let L and L' be arbitrary regular languages over the alphabet $\{\texttt{0}, \texttt{1}\}$. Prove that the following languages are also regular:

(a) $L \sqcap L' := \left\{x \sqcap y \;\middle|\; x \in L \text{ and } y \in L' \text{ and } |x| = |y|\right\}$, where $x \sqcap y$ denotes bitwise-and. For example, $\texttt{0011} \sqcap \texttt{0101} = \texttt{0001}$.

(b) $L \sqcup L' := \left\{x \sqcup y \;\middle|\; x \in L \text{ and } y \in L' \text{ with } |x| = |y|\right\}$, where $x \sqcup y$ denotes bitwise-or. For example, $\texttt{0011} \sqcup \texttt{0101} = \texttt{0111}$.

(c) $L \boxplus L' := \left\{x \boxplus y \;\middle|\; x \in L \text{ and } y \in L' \text{ with } |x| = |y|\right\}$, where $x \boxplus y$ denotes bitwise-exclusive-or. For example, $\texttt{0011} \boxplus \texttt{0101} = \texttt{0110}$.

(d) $faro(L, L') := \left\{faro(x, z) \;\middle|\; x \in L \text{ and } z \in L' \text{ with } |x| = |z|\right\}$, where

$$faro(x, z) := \begin{cases} z & \text{if } x = \varepsilon \\ a \cdot faro(z, y) & \text{if } x = ay \end{cases}$$

For example, $faro(\texttt{0011}, \texttt{0101}) = \texttt{00011011}$.

(e) $shuffles(L, L') := \bigcup_{w \in L, y \in L'} shuffles(w, y)$, where $shuffles(w, y)$ is the set of all strings obtained by shuffling w and y, or equivalently, all strings in which w and y are

complementary subsequences. Formally:

$$shuffles(w, y) = \begin{cases} \{y\} & \text{if } w = \varepsilon \\ \{w\} & \text{if } y = \varepsilon \\ \{a\} \bullet shuffles(x, y) \cup \{b\} \bullet shuffles(w, z) & \text{if } w = ax \text{ and } y = bz \end{cases}$$

For example, $shuffles(\texttt{01}, \texttt{10}) = \{\texttt{0101}, \texttt{0110}, \texttt{1001}, \texttt{1010}\}$ and $shuffles(\texttt{00}, \texttt{11}) = \{\texttt{0011}, \texttt{0101}, \texttt{1001}, \texttt{0110}, \texttt{1010}, \texttt{1100}\}$.

8. (a) Let $inc\colon \{\texttt{0}, \texttt{1}\}^* \to \{\texttt{0}, \texttt{1}\}^*$ denote the *increment* function, which transforms the binary representation of an arbitrary integer n into the binary representation of $n + 1$, truncated to the same number of bits. For example:

$$inc(\texttt{0010}) = \texttt{0011} \qquad inc(\texttt{0111}) = \texttt{1000} \qquad inc(\texttt{1111}) = \texttt{0000} \qquad inc(\varepsilon) = \varepsilon$$

Let $L \subseteq \{\texttt{0}, \texttt{1}\}^*$ be an arbitrary regular language. Prove that $inc(L) = \{inc(w) \mid w \in L\}$ is also regular.

(b) Let $dbl\colon \{\texttt{0}, \texttt{1}\}^* \to \{\texttt{0}, \texttt{1}\}^*$ denote the *doubling* function, which transforms the binary representation of an arbitrary integer n into the binary representation of $2n$, truncated to the same number of bits. For example:

$$dbl(\texttt{0010}) = \texttt{0100} \qquad dbl(\texttt{0111}) = \texttt{1110} \qquad dbl(\texttt{1111}) = \texttt{1110} \qquad dbl(\varepsilon) = \varepsilon$$

Let $L \subseteq \{\texttt{0}, \texttt{1}\}^*$ be an arbitrary regular language. Prove that $dbl(L) = \{dbl(w) \mid w \in L\}$ is also regular.

*(c) Let $tpl\colon \{\texttt{0}, \texttt{1}\}^* \to \{\texttt{0}, \texttt{1}\}^*$ denote the *tripling* function, which transforms the binary representation of an arbitrary integer n into the binary representation of $3n$, truncated to the same number of bits. For example:

$$tpl(\texttt{0010}) = \texttt{0110} \qquad tpl(\texttt{0111}) = \texttt{0101} \qquad tpl(\texttt{1111}) = \texttt{1101} \qquad tpl(\varepsilon) = \varepsilon$$

Let $L \subseteq \{\texttt{0}, \texttt{1}\}^*$ be an arbitrary regular language. Prove that $tpl(L) = \{tpl(w) \mid w \in L\}$ is also regular. *[Hint: It may be easier to consider the language $tpl(L^R)^R$ first.]*

*9. Let $L \subseteq \Sigma^*$ be an arbitrary regular language. Prove that the following languages are regular.

(a) $sqrt(L) := \left\{x \in \Sigma^* \;\middle|\; xy \in L \text{ for some } y \in \Sigma^* \text{ such that } |y| = |x|^2\right\}$

(b) $log(L) := \left\{x \in \Sigma^* \;\middle|\; xy \in L \text{ for some } y \in \Sigma^* \text{ such that } |y| = 2^{|x|}\right\}$

(c) $flog(L) := \left\{x \in \Sigma^* \;\middle|\; xy \in L \text{ for some } y \in \Sigma^* \text{ such that } |y| = F_{|x|}\right\}$, where F_n is the nth Fibonacci number.

*10. Let $L \subseteq \Sigma^*$ be an arbitrary regular language. Prove that the following languages are regular.

(a) $somerep(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for some } n \geq 0\}$

(b) $\mathit{allreps}(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for every } n \geq 0\}$

(c) $\mathit{manyreps}(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for infinitely many } n \geq 0\}$

(d) $\mathit{fewreps}(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for finitely many } n \geq 0\}$

(e) $\mathit{powers}(L) := \left\{w \in \Sigma^* \;\middle|\; w^{2^n} \in L \text{ for some } n \geq 0\right\}$

★(f) $\mathit{whatthe}_N(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for some } n \in N\}$, where N is an ***arbitrary*** fixed set of non-negative integers. *[Hint: You only have to prove that an accepting NFA exists; you don't have to describe how to construct it.]*

[Hint: For each of these languages, there is an accepting NFA with at most q^q states, where q is the number of states in some DFA that accepts L.]

*11. For any string $w \in (\mathtt{0} + \mathtt{1})^*$, let $\langle w \rangle_2$ denote the integer represented by w in binary. For example:

$$\langle \varepsilon \rangle_2 = 0 \qquad \langle \mathtt{0010} \rangle_2 = 2 \qquad \langle \mathtt{0111} \rangle_2 = 7 \qquad \langle \mathtt{1111} \rangle_2 = 15$$

Let L and L' be arbitrary regular languages over the alphabet $\{\mathtt{0}, \mathtt{1}\}$. Prove that the following language is also regular:

$$\left\{w \in (\mathtt{0} + \mathtt{1})^* \;\middle|\; \langle w \rangle_2 = \langle x \rangle_2 + \langle y \rangle_2 \text{ for some strings } x \in L \text{ and } y \in L'\right\}$$

★12. Let $L \subseteq \Sigma^*$ be an arbitrary regular language. Prove that the following languages are regular.

(a) $\mathit{repsqrt}(L) = \left\{w \in \Sigma^* \;\middle|\; w^{|w|} \in L\right\}$.

(b) $\mathit{replog}(L) = \left\{w \in \Sigma^* \;\middle|\; w^{2^{|w|}} \in L\right\}$.

(c) $\mathit{repflog}(L) = \left\{w \in \Sigma^* \;\middle|\; w^{F_{|w|}} \in L\right\}$, where F_n is the nth Fibonacci number.

[Hint: The NFAs for these languages use a **LOT** *of states. Let $M = (\Sigma, Q, s, A, \delta)$ be a DFA that accepts L. Imagine that you somehow know $\delta^*(q, w)$ in advance, for every state $q \in Q$. Ha, ha, ha! Mine is an evil laugh!]*

The product of mental labor — science — always stands far below its value, because the labor-time necessary to reproduce it has no relation at all to the labor-time required for its original production. For example, a schoolboy can learn the binomial theorem in an hour.

— Karl Marx, *Theories of Surplus Value* (1863)

*Imagine a piano keyboard, eh, 88 keys, only 88 and yet, and yet, hundreds of new melodies, new tunes, new harmonies are being composed upon hundreds of different keyboards every day in Dorset alone. Our language, tiger, our language: Hundreds of thousands of available words, frillions of legitimate new ideas, so that I can say the following sentence and be utterly sure that nobody has ever said it before in the history of human communication: "**Hold the newsreader's nose squarely, waiter, or friendly milk will countermand my trousers.**" Perfectly ordinary words, but never before put in that precise order. A unique child delivered of a unique mother.*

— Stephen Fry, *A Bit of Fry and Laurie*, Series 1, Episode 3 (1989)

5 Context-Free Languages and Grammars

5.1 Definitions

Intuitively, a language is regular if it can be built from individual strings by concatenation, union, and repetition. In this note, we consider a wider class of ***context-free*** languages, which are languages that can be built from individual strings by concatenation, union, and *recursion*.

Formally, a language is context-free if and only if it has a certain type of recursive description known as a ***context-free grammar***, which is a structure with the following components:

- A finite set Σ, whose elements are called ***symbols*** or ***terminals***.
- A finite set Γ disjoint from Σ, whose elements are called ***non-terminals***.
- A finite set R of ***production rules*** of the form $A \to w$, where $A \in \Gamma$ is a non-terminal and $w \in (\Sigma \cup \Gamma)^*$ is a string of symbols and variables.
- A ***starting*** non-terminal, typically denoted S.

For example, the following eight production rules describe a context free grammar with terminals $\Sigma = \{0, 1\}$ and non-terminals $\Gamma = \{S, A, B, C\}$:

$$\begin{array}{llll} S \to A & A \to 0A & B \to B1 & C \to \varepsilon \\ S \to B & A \to 0C & B \to C1 & C \to 0C1 \end{array}$$

Normally we write grammars more compactly by combining the right sides of all rules for each non-terminal into one list, with alternatives separated by vertical bars.[1] For example, the previous grammar can be written more compactly as follows:

$$\begin{aligned} S &\to A \mid B \\ A &\to 0A \mid 0C \\ B &\to B1 \mid C1 \\ C &\to \varepsilon \mid 0C1 \end{aligned}$$

[1] Yes, this means we now have *three* symbols $\cup$, $+$, and $\mid$ with exactly the same meaning. Sigh.

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

For the rest of this lecture, I will *almost* always use the following notational conventions.

- Monospaced digits ($\mathtt{0}$, $\mathtt{1}$, $\mathtt{2}$, ...), and symbols ($\diamond$, $\mathtt{\$}$, $\mathtt{\#}$, $\bullet$, ...) are explicit terminals.
- Early lower-case Latin letters ($a, b, c, \ldots$) represent unknown or arbitrary terminals in Σ.
- Upper-case Latin letters ($A, B, C, \ldots$) and the letter S represent non-terminals in Γ.
- Late lower-case Latin letters ($\ldots, w, x, y, z$) represent strings in $(\Sigma \cup \Gamma)^*$, whose characters could be either terminals or non-terminals.

We can ***apply*** a production rule to a string in $(\Sigma \cup \Gamma)^*$ by replacing any instance of the non-terminal on the left of the rule with the string on the right. More formally, for any strings $x, y, z \in (\Sigma \cup \Gamma)^*$ and any non-terminal $A \in \Gamma$, applying the production rule $A \to y$ to the string xAz yields the string xyz. We use the notation $\boldsymbol{xAz \rightsquigarrow xyz}$ to describe this application. For example, we can apply the rule $C \to \mathtt{0}C\mathtt{1}$ to the string $\mathtt{00}C\mathtt{1}BAC\mathtt{0}$ in two different ways:

$$\mathtt{00}\underbrace{C}\mathtt{1}BAC\mathtt{0} \rightsquigarrow \mathtt{00}\underbrace{\mathtt{0}C\mathtt{1}}\mathtt{1}BAC\mathtt{0} \qquad \mathtt{00}C\mathtt{1}BA\underbrace{C}\mathtt{0} \rightsquigarrow \mathtt{00}C\mathtt{1}BA\underbrace{\mathtt{0}C\mathtt{1}}\mathtt{0}$$

More generally, for any strings $x, z \in (\Sigma \cup \Gamma)^*$, we say that ***z* derives from *x***, written $\boldsymbol{x \rightsquigarrow^* z}$, if we can transform x into z by applying a finite sequence of production rules, or more formally, if either

- $x = z$, or
- $x \rightsquigarrow y$ and $y \rightsquigarrow^* z$ for some string $y \in (\Sigma \cup \Gamma)^*$.

Straightforward definition-chasing implies that, for any strings $w, x, y, z \in (\Sigma \cup \Gamma)^*$, if $x \rightsquigarrow^* y$, then $wxz \rightsquigarrow^* wyz$.

The ***language L(w)*** of any string $w \in (\Sigma \cup \Gamma)^*$ is the set of all strings in Σ^* that derive from w:

$$L(w) := \{x \in \Sigma^* \mid w \rightsquigarrow^* x\}.$$

The language ***generated by*** a context-free grammar G, denoted $\boldsymbol{L(G)}$, is the language of its starting non-terminal. Finally, a language is ***context-free*** if it is generated by some context-free grammar.

Context-free grammars are sometimes used to model natural languages. In this context, the symbols are *words*, and the strings in the languages are *sentences*. For example, the following grammar describes a simple subset of English sentences. (Here I diverge from the usual notation conventions. Strings in ⟨angle brackets⟩ are non-terminals, and regular strings are terminals.)

⟨sentence⟩ → ⟨noun phrase⟩⟨verb phrase⟩⟨noun phrase⟩
⟨noun phrase⟩ → ⟨adjective phrase⟩⟨noun⟩
⟨adj. phrase⟩ → ⟨article⟩ | ⟨possessive⟩ | ⟨adjective phrase⟩⟨adjective⟩
⟨verb phrase⟩ → ⟨verb⟩ | ⟨adverb⟩⟨verb phrase⟩

⟨noun⟩ → dog | trousers | daughter | nose | homework | time lord | pony | ⋯
⟨article⟩ → the | a | some | every | that | ⋯
⟨possessive⟩ → ⟨noun phrase⟩'s | my | your | his | her | ⋯
⟨adjective⟩ → friendly | furious | moist | green | severed | timey-wimey | little | ⋯
⟨verb⟩ → ate | found | wrote | killed | mangled | saved | invented | broke | ⋯
⟨adverb⟩ → squarely | incompetently | barely | sort of | awkwardly | totally | ⋯

5.2 Parse Trees

It is often useful to visualize derivations of strings in $L(G)$ using a ***parse tree***. The parse tree for a string $w \in L(G)$ is a rooted ordered tree where

- Each leaf is labeled with a terminal or the empty string ε. Concatenating these in order from left to right yields the string w.
- Each internal node is labeled with a non-terminal. In particular, the root is labeled with the start non-terminal S.
- For each internal node v, there is a production rule $A \to \omega$ where A is the label of v and the symbols in ω are the labels of the children of v in order from left to right.

In other words, the production rules of the grammar describe *template trees* that can be assembled into larger parse trees. For example, the simple grammar on the previous page has the following templates, one for each production rule:

The same grammar gives us the following parse tree for the string `000011`:

Our more complicated "English" grammar gives us parse trees like the following:

Any parse tree that contains at least one node with more than one non-terminal child corresponds to several different derivations. For example, when deriving an "English" sentence, we have a choice of whether to expand the first ⟨noun phrase⟩ ("your furious green time lord") before or after the second ("my dog's trousers").

A string w is ***ambiguous*** with respect to a grammar if there is more than one parse tree for w, and a grammar G is ***ambiguous*** is some string is ambiguous with respect to G. Neither of the previous example grammars is ambiguous. However, the grammar $S \to \mathtt{1} \mid S\mathtt{+}S$ is ambiguous, because the string $\mathtt{1+1+1+1}$ has five different parse trees:

A context-free language L is ***inherently ambiguous*** if every context-free grammar that generates L is ambiguous. The language generated by the previous grammar (the regular language $(\mathtt{1+})^*\mathtt{1}$) is *not* inherently ambiguous, because the unambiguous grammar $S \to \mathtt{1} \mid \mathtt{1+}S$ generates the same language.

5.3 From Grammar to Language

Let's figure out the language generated by our first example grammar

$$S \to A \mid B \qquad A \to \mathtt{0}A \mid \mathtt{0}C \qquad B \to B\mathtt{1} \mid C\mathtt{1} \qquad C \to \varepsilon \mid \mathtt{0}C\mathtt{1}.$$

Since the production rules for non-terminal C do not refer to any other non-terminal, let's begin by figuring out $L(C)$. After playing around with the smaller grammar $C \to \varepsilon \mid \mathtt{0}C\mathtt{1}$ for a few seconds, you can probably guess that its language is $\{\varepsilon, \mathtt{01}, \mathtt{0011}, \mathtt{000111}, \ldots\}$, that is, the set all of strings of the form $\mathtt{0}^n\mathtt{1}^n$ for some integer n. For example, we can derive the string $\mathtt{00001111}$ from the start non-terminal S using the following derivation:

$$C \rightsquigarrow \mathtt{0}C\mathtt{1} \rightsquigarrow \mathtt{00}C\mathtt{11} \rightsquigarrow \mathtt{000}C\mathtt{111} \rightsquigarrow \mathtt{0000}C\mathtt{1111} \rightsquigarrow \mathtt{0000}\varepsilon\mathtt{1111} = \mathtt{00001111}$$

The same derivation can be viewed as the following parse tree:

In fact, it is not hard to *prove* by induction that $L(C) = \{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$ as follows. As usual when we prove that two sets X and Y are equal, the proof has two stages: one stage to prove $X \subseteq Y$, the other to prove $Y \subseteq X$.

Lemma 5.1. *$C \rightsquigarrow^* \mathtt{0}^n\mathtt{1}^n$ for every non-negative integer n.*

Proof: Fix an arbitrary non-negative integer n. Assume that $C \rightsquigarrow^* \mathtt{0}^k\mathtt{1}^k$ for every non-negative integer $k < n$. There are two cases to consider.

- If $n = 0$, then $0^n 1^n = \varepsilon$. The rule $C \to \varepsilon$ implies that $C \rightsquigarrow \varepsilon$ and therefore $C \rightsquigarrow^* \varepsilon$.
- Suppose $n > 0$. The inductive hypothesis implies that $C \rightsquigarrow^* 0^{n-1}1^{n-1}$. Thus, the rule $C \to 0C1$ implies that $C \rightsquigarrow 0C1 \rightsquigarrow^* 0(0^{n-1}1^{n-1})1 = 0^n 1^n$.

In both cases, we conclude that that $C \rightsquigarrow^* 0^n 1^n$, as claimed. □

Lemma 5.2. *For every string $w \in L(C)$, we have $w = 0^n 1^n$ for some non-negative integer n.*

Proof: Fix an arbitrary string $w \in L(C)$. Assume that for any string $x \in L(C)$ such that $|x| < |w|$, we have $x = 0^k 1^k$ for some non-negative integer k. There are two cases to consider, one for each production rule.

- If $w = \varepsilon$, then $w = 0^0 1^0$.
- Otherwise, $w = 0x1$ for some string $x \in L(C)$. Because $|x| = |w| - 2 < |w|$, the inductive hypothesis implies that $x = 0^k 1^k$ for some integer k. Then we have $w = 0^{k+1} 1^{k+1}$.

In both cases, we conclude that $w = 0^n 1^n$ for some non-negative integer n, as claimed. □

The first proof uses induction on strings, following the boilerplate proposed in the very first lecture; in particular, the case analysis mirrors the recursive definition of "string". The second proof uses *structural induction* on the parse tree of the string $0^n 1^n$; the case analysis mirrors the recursive definition of the language of S, as described by the production rules. In both proofs, as in every proof by induction, the inductive hypothesis is "Assume there is no smaller counterexample."

Similar analysis implies that $L(A) = \{0^m 1^n \mid m > n\}$ and $L(B) = \{0^m 1^n \mid m < n\}$, and therefore $L(S) = \{0^m 1^n \mid m \neq n\}$.

5.3.1 Careful With Those Epsilons

There is an important subtlety in the proof of Lemma 5.2. The proof is written as induction on the length of the string w; unfortunately, this induction pattern does not work for all context-free grammars. Consider the following ambiguous grammar

$$S \to \varepsilon \mid SS \mid 0S1 \mid 1S0.$$

A bit of experimentation should convince you that $L(S)$ is the language of all binary strings with the same number of 0s and 1s. We cannot use the string-induction boilerplate for this grammar, because there are arbitrarily long[2] derivations of the form

$$S \rightsquigarrow SS \rightsquigarrow S \rightsquigarrow SS \rightsquigarrow S \rightsquigarrow SS \rightsquigarrow SS \rightsquigarrow \cdots \rightsquigarrow w,$$

which alternately apply the productions $S \to SS$ and $S \to \varepsilon$. Specifically, even if we knew that our arbitrary string w can be written as xy for some strings $x, y \in L(S)$, we cannot guarantee that $|x| < |w|$ and $|y| < |w|$, to we cannot apply the standard string-induction hypothesis.

However, we can still argue inductively about this grammar, by considering a *minimum-length* derivation of w, and basing the case analysis on the first production in this derivation. Here's an example of this induction boilerplate in action, with the modified boilerplate language highlighted.

[2] but not infinite; derivations are finite *by definition*!

Lemma 5.3. *For every string $w \in L(S)$, we have $\#(0, w) = \#(1, w)$.*

Proof: Let w be an arbitrary string in $L(S)$. Fix an minimum-length derivation of w.

Assume that for any string $x \in L(S)$ that is shorter than w, we have $x = 0^k 1^k$ for some non-negative integer k. There are four cases to consider, depending on the first production in our fixed derivation.

- Suppose the first production is $S \to \varepsilon$. Then $w = \varepsilon$ and therefore $\#(0, w) = \#(1, w) = 0$ by definition.
- Suppose the first production is $S \to SS$. Then $w = xy$ for some strings $x, y \in L(S)$. Both x and y must be non-empty; otherwise, we could shorten our derivation of w. Thus, both x and y are shorter than w. The inductive hypothesis implies $\#(0, x) = \#(1, x)$ and $\#(0, y) = \#(1, y)$, so $\#(0, w) = \#(0, x) + \#(0, y) = \#(1, x) + \#(1, y) = \#(1, w)$.
- Suppose the first production is $S \to 0S1$. Then $w = 0x1$ for some string $x \in L(S)$. The inductive hypothesis implies $\#(0, x) = \#(1, x)$ so $\#(0, w) = \#(0, x) + 1 = \#(1, x) + 1 = \#(1, w)$.
- Finally, suppose the first production is $S \to 1S0$. Then $w = 1x0$ for some string $x \in L(S)$. The inductive hypothesis implies $\#(0, x) = \#(1, x)$ so $\#(0, w) = \#(0, x) + 1 = \#(1, w) + 1 = \#(1, w)$.

In all cases, we conclude that $\#(0, w) = \#(1, w)$, as claimed. □

Another (more traditional) way to handle this issue is to fix an *arbitrary* derivation, and then induct on the length of the derivation, rather than the length of the string itself. The case analysis is still based on the first production in the chosen derivation.

In fact, this subtlety only matters for grammars that either contain a *nullable* non-terminal A such that $A \rightsquigarrow^* \varepsilon$ or *equivalent* nonterminals A and B such that $A \rightsquigarrow^* B$ and $B \rightsquigarrow^* A$. We describe algorithms to identify these pathologies and remove them from the grammar (without changing its language) in Section 5.9 below.

5.3.2 Mutual Induction

Another pitfall in induction proofs for context-free languages is that non-terminals may invoke each other. Consider, for example, the grammar

$$S \to 0A1 \mid \varepsilon \qquad A \to 1S0 \mid \varepsilon$$

Because each non-terminal appears on the right side of a production rule for the other, we must argue about $L(S)$ and $L(A)$ simultaneously.

Lemma 5.4. $L(S) = (01)^*$.

Proof: We actually prove simultaneously that $L(S) = (01)^*$ and $L(A) = (10)^*$.

First, we claim that for any non-negative integer n, we have $(01)^n \in L(S)$ and $(10)^n \in L(A)$. Let n be an arbitrary non-negative integer, and assume, for all non-negative integers $m < n$, that $(01)^m \in L(S)$ and $(10)^m \in L(A)$. There are two cases to consider.

- If $n = 0$, the production rules $S \to \varepsilon$ and $A \to \varepsilon$ immediately imply $S \rightsquigarrow \varepsilon = (01)^n$ and $A \rightsquigarrow \varepsilon = (10)^n$.

- Suppose $n > 0$. We easily observe that $(\mathtt{01})^n = \mathtt{0}(\mathtt{10})^{n-1}\mathtt{1}$, so the production rule $S \to \mathtt{0}A\mathtt{1}$ and inductive hypothesis imply $S \rightsquigarrow \mathtt{0}A\mathtt{1} \rightsquigarrow^* (\mathtt{01})^n$. Symmetrically, $(\mathtt{10})^n = \mathtt{1}(\mathtt{01})^{n-1}\mathtt{0}$, so the production rule $A \to \mathtt{1}S\mathtt{0}$ and the inductive hypothesis implies $A \rightsquigarrow \mathtt{1}S\mathtt{0} \rightsquigarrow^* (\mathtt{10})^n$.

Next we claim that for every string $w \in L(S)$, we have $w = (\mathtt{01})^n$ for some non-negative integer n, and for every string $w \in L(A)$, we have $w = (\mathtt{10})^n$ for some non-negative integer n. The proof requires two stages.

- Let w be an arbitrary string in $L(S)$, and assume for all $x \in L(A)$ such that $|x| < |w|$ that $x = (\mathtt{10})^n$ for some non-negative integer n. There are two cases to consider.
  - If $w = \varepsilon$, then $w = (\mathtt{01})^0$.
  - Suppose $w = \mathtt{0}x\mathtt{1}$ for some string $x \in L(A)$. The inductive hypothesis implies $x = (\mathtt{10})^n$ for some non-negative integer n. It follows that $w = \mathtt{0}(\mathtt{10})^n\mathtt{1} = (\mathtt{01})^{n+1}$.
- Let w be an arbitrary string in $L(A)$, and assume for all $x \in L(S)$ such that $|x| < |w|$ that $x = (\mathtt{01})^n$ for some non-negative integer n. There are two cases to consider.
  - If $w = \varepsilon$, then $w = (\mathtt{10})^0$.
  - Suppose $w = \mathtt{1}x\mathtt{0}$ for some string $x \in L(S)$. The inductive hypothesis implies $x = (\mathtt{01})^n$ for some non-negative integer n. It follows that $w = \mathtt{1}(\mathtt{01})^n\mathtt{0} = (\mathtt{10})^{n+1}$.

Together these two claims imply $L(S) = (\mathtt{01})^*$ and $L(A) = (\mathtt{10})^*$, as required. □

5.4 More Examples

Here are some more examples of context-free languages and grammars that generate them, along with brief sketches of correctness proofs.

- Palindromes in $\{\mathtt{0}, \mathtt{1}\}^*$:

$$S \to \mathtt{0}S\mathtt{0} \mid \mathtt{1}S\mathtt{1} \mid \mathtt{0} \mid \mathtt{1} \mid \varepsilon$$

  This grammar is a straightforward translation of the recursive definition of palindrome.

- Strings in $(\mathtt{0} + \mathtt{1})^*$ that are *not* palindromes.

$$S \to \mathtt{0}S\mathtt{0} \mid \mathtt{1}S\mathtt{1} \mid \mathtt{0}Z\mathtt{1} \mid \mathtt{1}Z\mathtt{0}$$
$$Z \to \varepsilon \mid \mathtt{0}Z \mid \mathtt{1}Z$$

  A string w is a non-palindrome if and only if $w = x\mathtt{0}z\mathtt{1}x^R$ or $w = x\mathtt{1}z\mathtt{0}x^R$ for some (possibly empty) strings x and y.

- Strings in $\{\mathtt{0}, \mathtt{1}\}^*$ with the same number of $\mathtt{0}$s and $\mathtt{1}$s:

$$S \to \mathtt{0}S\mathtt{1} \mid \mathtt{1}S\mathtt{0} \mid SS \mid \varepsilon$$

  A non-empty string w has the same number of $\mathtt{0}$s and $\mathtt{1}$s if and only one of the following conditions holds:

  - We can write $w = xy$ for some non-empty strings x and y such that $\#(\mathtt{0}, x) = \#(\mathtt{1}, x)$ and $\#(\mathtt{0}, y) = \#(\mathtt{1}, y)$.

- $\#(0,x) > \#(1,x)$ for every non-empty proper prefix x of w. In this case, $w = 0z1$ for some string z with $\#(0,z) = \#(1,z)$.
- $\#(0,x) < \#(1,x)$ for every non-empty proper prefix x of w. In this case, $w = 1z0$ for some string z with $\#(0,z) = \#(1,z)$.

- Strings in $\{0,1\}^*$ with the same number of 0s and 1s, again:

$$S \to 0S1S \mid 1S0S \mid \varepsilon$$

Let w be any non-empty string such that $\#(0,w) = \#(1,w)$, let x be the shortest non-empty prefix of w such that $\#(0,x) = \#(1,x)$, and let y be the complementary suffix of w, so $w = xy$. It is not hard to prove that x begins and ends with different symbols, so either $w = 0z1y$ or $w = 1z0y$, where $\#(0,y) = \#(1,y)$ and $\#(0,z) = \#(1,z)$.

- Strings in $\{0,1\}^*$ in which the number of 0s is greater than or equal to the number of 1s:

$$S \to 0S1 \mid 0S \mid 1S0 \mid S0 \mid SS \mid \varepsilon \qquad\qquad S \to 0S1S \mid 0SS \mid 1S0S \mid S0S \mid \varepsilon$$

We have to different grammars, each constructed from a grammar for strings with equal 0s and 1s by either dropping the 1 or keeping the 1 from the right side of each production rule containing a 1. For example, we split the production rule $S \to 0S1$ in the first grammar into two production rules $S \to 0S1$ and $S \to 0S$.

If we add the trivial production $S \to 0$ to the first grammar, we can remove two redundant productions to get the simpler grammar

$$S \to 0S1 \mid 1S0 \mid SS \mid 0 \mid \varepsilon$$

- Strings in $\{0,1\}^*$ with *different* numbers of 0s and 1s:

$$\begin{aligned} S &\to O \mid I && \text{(different)} \\ O &\to E0O \mid E0E && \text{(more 0s)} \\ I &\to E1I \mid E1E && \text{(more 1s)} \\ E &\to 0E1E \mid 1E0E \mid \varepsilon && \text{(equal)} \end{aligned}$$

We can argue correctness by considering each non-terminal in turn, in reverse order.

- E generates all strings with the same number of 0s and 1s, as in the previous example.
- I generates all strings with more 1s than 0s. Any such string can be decomposed into its longest prefix with the same number of 0s and 1s (E), followed by a 0, followed by a suffix with at least as many 0s as 1s (I or E).
- Symmetrically, O generates all strings with more 0s than 1s.
- Finally, S generates all strings with different numbers of 0s and 1s. Any such string either has more 0s (O) or more 1s (I)

- Balanced strings of parentheses:

$$S \to (S) \mid SS \mid \varepsilon \qquad \text{or} \qquad S \to (S)S \mid \varepsilon$$

Here we have two grammars for the same language. The first one uses simpler productions, and is a bit closer to the natural recursive definition. However, the first grammar is

ambiguous — consider the string `()()()` — while the second grammar is not. The second grammar decomposes any balanced string of parentheses into its shortest non-empty balanced prefix, which must start with `(` and end with `)`, and the remaining suffix, which must be balanced.

- Unbalanced strings of parentheses—the complement of the previous language:

$$
\begin{aligned}
S &\to L \mid RX && \text{(unbalanced)}\\
L &\to E\texttt{(}L \mid E\texttt{(}E && \text{(more left parens)}\\
R &\to E\texttt{)}R \mid E\texttt{)}E && \text{(more right parens)}\\
E &\to \varepsilon \mid \texttt{(}E\texttt{)}E \mid \texttt{)}E\texttt{(}E && \text{(equal left and right)}\\
X &\to \varepsilon \mid \texttt{(}X \mid \texttt{)}X && \text{(anything)}
\end{aligned}
$$

A string w of parens is balanced if and only if both (a) w has the same number of left and right parens and (b) no prefix of w has more right parens than left parens. (Proving this fact is a good homework exercise.) Thus, a string w of parens is *unbalanced* if and only if *either* w has more left parens than right parens *or* some prefix of w has more right parens than left parens.

- Arithmetic expressions, possibly with redundant parentheses, over the variables `X` and `Y`:

$$
\begin{aligned}
E &\to E\texttt{+}T \mid T && \text{(expressions)}\\
T &\to T\texttt{×}F \mid F && \text{(terms)}\\
F &\to \texttt{(}E\texttt{)} \mid \texttt{X} \mid \texttt{Y} && \text{(factors)}
\end{aligned}
$$

Every Eexpression is a sum of Terms, every Term is a product of Factors, and every Factor is either a variable or a parenthesized Eexpression.

- Regular expressions over the alphabet $\{\texttt{0}, \texttt{1}\}$ *without* redundant parentheses

$$
\begin{aligned}
S &\to T \mid T\texttt{+}S && \text{(Regular expressions)}\\
T &\to F \mid FT && \text{(Terms = summable expressions)}\\
F &\to \texttt{Ø} \mid W \mid \texttt{(}T\texttt{+}S\texttt{)} \mid X\texttt{*} \mid \texttt{(}Y\texttt{)*} && \text{(Factors = concatenable expressions)}\\
X &\to \texttt{Ø} \mid \texttt{Ɛ} \mid \texttt{0} \mid \texttt{1} && \text{(Directly starrable expressions)}\\
Y &\to T\texttt{+}S \mid F\texttt{•}T \mid X\texttt{*} \mid \texttt{(}Y\texttt{)*} \mid ZZ && \text{(Starrable expressions needing parens)}\\
W &\to \texttt{Ɛ} \mid Z && \text{(Words = strings)}\\
Z &\to \texttt{0} \mid \texttt{1} \mid ZZ && \text{(Non-empty strings)}
\end{aligned}
$$

Every regular expression is a sum of terms; every term is a concatenation of factors. Every factor is either the empty-set symbol, a string, a nontrivial sum of terms in parens, or a starred expression. The expressions `Ø*`, `Ɛ*`, `0*`, and `1*` require no parentheses; otherwise, the starred subexpression is either a nontrivial sum of terms, a nontrivial concatenation of factors, a starred expression, or a string of length 2 or more.

The "epsilon" symbol `Ɛ` in the production rules for W and Z does *not* represent the empty string *per se*, but rather an actual symbol that might appear in a regular expression. The empty string is not a regular expression, but the one-symbol string `Ɛ` is a regular expression that represents the set containing only the empty string!

The final grammar illustrates an important subtlety for certain applications of context-free grammars. This grammar is considerably more complicated than one might initially expect from the definition of regular languages. It's tempting to suggest a much simpler grammar like

$$S \to \emptyset \mid \varepsilon \mid \mathtt{0} \mid \mathtt{1} \mid S\mathtt{+}S \mid SS \mid S\mathtt{\star} \mid \mathtt{(}S\mathtt{)}$$

but this is incorrect! This grammar does correctly generate all regular expressions *as raw strings*, but it allows parse trees that do not respect the *meaning* of the regular expression. For example, this "simpler" grammar can parse the regular expression string `0⋆+1⋆` in two different ways:

The first tree correctly parses the regular expression string `1⋆+0⋆` as the expression $(\mathtt{1}^*) + (\mathtt{0}^*)$ but without the redundant parentheses. The second tree incorrectly parses the same string as $(\mathtt{1}^* + \mathtt{0})^*$, which describes a *very* different regular language!

5.5 Regular Languages are Context-Free

The following inductive argument proves that every regular language is also a context-free language. Let L be an arbitrary regular language, encoded by some regular expression R. Assume that any regular expression simpler than R represents a context-free language. ("Assume no smaller counterexample.") We construct a context-free grammar for L as follows. There are several cases to consider.

- Suppose L is empty. Then L is generated by the trivial grammar $S \to S$.
- Suppose $L = \{w\}$ for some string $w \in \Sigma^*$. Then L is generated by the grammar $S \to w$.
- Suppose L is the union of some regular languages L_1 and L_2. The inductive hypothesis implies that L_1 and L_2 are context-free. Let G_1 be a context-free language for L_1 with starting non-terminal S_1, and let G_2 be a context-free language for L_2 with starting non-terminal S_2, where the non-terminal sets in G_1 and G_2 are disjoint. Then $L = L_1 \cup L_2$ is generated by the production rule $S \to S_1 \mid S_2$.
- Suppose L is the concatenation of some regular languages L_1 and L_2. The inductive hypothesis implies that L_1 and L_2 are context-free. Let G_1 be a context-free language for L_1 with starting non-terminal S_1, and let G_2 be a context-free language for L_2 with starting non-terminal S_2, where the non-terminal sets in G_1 and G_2 are disjoint. Then $L = L_1 L_2$ is generated by the production rule $S \to S_1 S_2$.
- Suppose L is the Kleene closure of some regular language L_1. The inductive hypothesis implies that L_1 is context-free. Let G_1 be a context-free language for L_1 with starting non-terminal S_1. Then $L = L_1^*$ is generated by the production rule $S \to \varepsilon \mid S_1 S$.

In every case, we have found a context-free grammar that generates L, which means L is context-free.

In the previous lecture note, we proved that the context-free language $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$ is not regular. (In fact, this is the *canonical example* of a non-regular language.) Thus, context-free grammars are strictly more expressive than regular expressions.

5.6 Not Every Language is Context-Free

Again, you may be tempted to conjecture that *every* language is context-free, but a variant of our earlier cardinality argument implies that this is not the case.

Any context-free grammar over the alphabet Σ can be encoded as a string over the alphabet $\Sigma \cup \Gamma \cup \{\varepsilon, \rightarrow, |, \$\}$, where $\$$ indicates the end of the production rules for each non-terminal. For example, our example grammar

$$S \rightarrow A \mid B \qquad A \rightarrow 0A \mid 0C \qquad B \rightarrow B1 \mid C1 \qquad C \rightarrow \varepsilon \mid 0C1$$

can be encoded as the string

$$\texttt{S}\rightarrow\texttt{A}|\texttt{B}\$\texttt{A}\rightarrow\texttt{0A}|\texttt{0C}\$\texttt{B}\rightarrow\texttt{B1}|\texttt{C1}\$\texttt{C}\rightarrow\varepsilon|\texttt{0C1}\$$$

We can further encode any such string as a *binary* string by associating each symbol in the set $\Sigma \cup \Gamma \cup \{\varepsilon, \rightarrow, |, \$\}$ with a different binary substring. Specifically, if we encode each of the grammar symbols $\varepsilon, \rightarrow, |, \$$ as a string of the form $\texttt{11}^*\texttt{0}$, each terminal in Σ as a string of the form $\texttt{011}^*\texttt{0}$, and each non-terminal as a string of the form $\texttt{0011}^*\texttt{0}$, we can unambiguously recover the grammar from the encoding. For example, applying the code

$\varepsilon \mapsto \texttt{10}$	$\texttt{0} \mapsto \texttt{010}$	$\texttt{S} \mapsto \texttt{0010}$
$\rightarrow \mapsto \texttt{110}$	$\texttt{1} \mapsto \texttt{0110}$	$\texttt{A} \mapsto \texttt{00110}$
$| \mapsto \texttt{1110}$		$\texttt{B} \mapsto \texttt{001110}$
$\$ \mapsto \texttt{11110}$		$\texttt{C} \mapsto \texttt{0011110}$

transforms our example grammar into the 136-bit string

```
00101100011011100011101111000110
11001000110111001000111101111000
11101100011100110111000111100110
11110001111011010111001000111100
11011110.
```

Adding a $\texttt{1}$ to the start of this bit string gives us the binary encoding of the integer

$$102\,231\,235\,533\,163\,527\,515\,344\,124\,802\,467\,059\,875\,038.$$

Our construction guarantees that two different context-free grammars over the same alphabet (ignoring changing the names of the non-terminals) yield different positive integers. Thus, the set of context-free grammars over any alphabet is *at most* as large as the set of integers, and is therefore countably infinite. (Most integers are not encodings of context-free grammars, but that only helps us.) It follows that the set of all context-free *languages* over any fixed alphabet is also countably infinite. But we already showed that the set of *all* languages over any alphabet is uncountably infinite. So almost all languages are non-context-free!

There are techniques for proving that specific languages are not context-free, just as there are for proving certain languages are not regular; unfortunately, they are beyond the scope of this course. In particular, the $\{\texttt{0}^n\texttt{1}^n\texttt{0}^n \mid n \geq 0\}$ is not context-free. (In fact, this is the *canonical example* of a non-context-free language.)

*5.7 Recursive Automata

All the flavors of finite-state automata we have seen so far describe/encode/accept/compute *regular* languages; these are precisely the languages that can be constructed from individual strings by union, concatenation, and unbounded repetition. Just as context-free grammars are recursive generalizations of regular expressions, we can define a class of machines called *recursive automata*, which generalize (nondeterministic) finite-state automata. Recursive automata were introduced by Walter Woods in 1970 for natural language parsing; Wodds' terminology *recursive transition networks* is more common among computational linguists.

Formally, a ***recursive automaton*** consists of the following components:

- A non-empty finite set Σ, called the ***input alphabet***
- Another non-empty finite set N, disjoint from Σ, whose elements are called ***module names***
- A ***start name*** $S \in N$
- A set $M = \{M_A \mid A \in N\}$ of NFAs, called ***modules***, over the alphabet $\Sigma \cup N$. Each module M_A has the following components:
  - A finite set Q_A of ***states***, such that $Q_A \cap Q_B = \varnothing$ for all $A \neq B$
  - A ***start*** state $s_A \in Q_A$
  - A unique ***terminal*** or ***accepting*** state $t_A \in Q_A$
  - A nondeterministic ***transition function*** $\delta_A \colon Q_A \times (\Sigma \cup \{\varepsilon\} \cup N) \to 2^{Q_A}$.

Equivalently, we have a single global transition function $\delta \colon Q \times (\Sigma \cup \{\varepsilon\} \cup N) \to 2^Q$, where $Q = \bigcup_{A \in N} Q_A$, such that for any name A and any state $q \in Q_A$ we have $\delta(q) \subseteq Q_A$. Machine M_S is called the ***main module***.

A ***configuration*** of a recursive automaton is a triple (w, q, s), where w is a string in Σ^* called the ***input***, q is a state in Q called the ***local state***, and σ is a string in Q^* called the ***stack***. The module containing the local state q is called the ***active module***. A configuration can be changed by three types of transitions.

- A ***read*** transition consumes the first symbol in the input and changes the local state within the active module, just like a standard NFA.
- An ***epsilon*** transition changes the local state within the active module, without consuming any input characters, just like a standard NFA.
- A ***call*** transition chooses a module name A, pushes some state in $\delta(q, A)$ onto the stack, and then changes the local state to s_A (thereby changing the active module to M_A), without consuming any input characters.
- Finally, if the current state is the terminal state of the active module *and* the stack is non-empty, a ***return*** transition pops the top state off the stack and makes it the new local state (thereby possibly changing the active module), without consuming any input characters.

Symbolically, we can describe these transitions as follows:

read:	$(ax, q, \sigma) \longmapsto (x, q', \sigma)$	for some $q' \in \delta(q, a)$
epsilon:	$(w, q, \sigma) \longmapsto (w, q', \sigma)$	for some $q' \in \delta(q, \varepsilon)$
call:	$(w, q, \sigma) \longmapsto (w, s_A, q' \cdot \sigma)$	for some $A \in N$ and some $q' \in \delta(q, A)$
return:	$(w, t_A, q \cdot \sigma) \longmapsto (w, q, \sigma)$	

A recursive automaton ***accepts*** a string w if there is a *finite* sequence of transitions starting at the start configuration (w, s_S, ε) and ending at the terminal configuration $(\varepsilon, t_S, \varepsilon)$.

★★★

Reformulate recursive automata using recursion-as-magic, analogously to the nondeterminism-as-magic in string NFAs or Han and Wood's expression automata. A recursive automaton module M_A accepts a string w if any only if there is a finite sequence of transitions

$$s_A = q_0 \xrightarrow{\alpha_1} q_1 \xrightarrow{\alpha_2} q_2 \xrightarrow{\alpha_3} \cdots \xrightarrow{\alpha_\ell} q_\ell = t_A$$

among states in Q_A, and a decomposition of w into substrings $x_1 \bullet x_2 \bullet \cdots \bullet x_\ell$, where one of the following conditions holds for each index i:

- $\alpha_i = \varepsilon$ and $x_i = \varepsilon$
- $\alpha_i \in \Sigma$ and $x_i = \alpha_i$
- $\alpha_i \in N$ and module M_{α_i} accepts x_i.

This model is then easily extended to more general transition labels:

- Recursive *string*-automata allow transitions to be labeled by either strings in Σ^* or module names, and the first two bullets in the previous list become "$\alpha_i = x_i$".
- Recursive *expression*-automata allow transitions to be labeled by either regular expressions over Σ or module names, and the first two bullets in the previous list become "α_i is a regular expression and x_i matches α_i".
- We could even consider *recursive-expression* automata, which allow transitions to be labeled by arbitrary regular expressions over $\Sigma \cup N$.
- We could even even consider recursive-*grammar* automata, which allow transitions to be labeled by arbitrary *context-free grammars* over $\Sigma \cup N$. WEEEE/EEEEW!

For example, the following recursive automaton accepts the language $\{0^m 1^n \mid m \neq n\}$. The recursive automaton has two component modules; the start machine named S and a "subroutine" named E (for "equal") that accepts the language $\{0^n 1^n \mid n \geq 0\}$. White arrows indicate recursive transitions. The large arrow into each module indicates that module's start state; the large arrow leading out of each module indicates that modules terminal state.

A recursive automaton for the language $\{0^m 1^n \mid m \neq n\}$

Lemma 5.5. *Every context-free language is accepted by a recursive automaton.*

Proof:

★★★

Direct construction from the CFG, with one module per nonterminal.

□

For example, the context-free grammar

$$S \to 0A \mid B1$$
$$A \to 0A \mid E$$
$$B \to B1 \mid E$$
$$E \to \varepsilon \mid 0E0$$

leads to the following recursive automaton with four modules:

★★★ Figure!

Lemma 5.6. *Every recursive automaton accepts a context-free language.*

Proof (sketch): Let $R = (\Sigma, N, S, \delta, M)$ be an arbitrary recursive automaton. We define a context-free grammar G that describes the language accepted by R as follows.

The set of nonterminals in G is isomorphic the state set Q; that is, for each state $q \in Q$, the grammar contains a corresponding nonterminal $[q]$. The language of $[q]$ will be the set of strings w such that there is a finite sequence of transitions starting at the start configuration (w, q, ε) and ending at the terminal configuration $(\varepsilon, t, \varepsilon)$, where t is the terminal state of the module containing q.

The grammar has four types of production rules, corresponding to the four types of transitions:

- ***read:*** For each symbol a and each pair of states p and q such that $p \in \delta(q, a)$, the grammar contains the production rule $[q] \to a[p]$.
- ***epsilon:*** For any two states p and q such that $p \in \delta(q, \varepsilon)$, the grammar contains the production rule $[q] \to [p]$.
- ***call:*** Each name A and each pair of states states p and q such that $p \in \delta(q, A)$, the grammar contains the production rule $[q] \to [s_A][p]$.
- ***return:*** Each name A, the grammar contains the production rule $[t_A] \to \varepsilon$.

Finally, the starting nonterminal of G is $[s_S]$, which corresponds to the start state of the main module.

We can now argue inductively that the grammar G and the recursive automaton R describe the same language. Specifically, any sequence of transitions in R from (w, s_S, ε) to $(\varepsilon, t_S, \varepsilon)$ can be transformed mechanically into a derivation of w from the nonterminal $[s_S]$ in G. Symmetrically, the ***leftmost*** derivation of any string w in G can be mechanically transformed into an accepting sequence of transitions in R. We omit the straightforward but tedious details. □

For example, the recursive automaton on the previous page gives us the following context-free grammar. To make the grammar more readable, I've renamed the nonterminals corresponding to start and terminal states: $S = [s_S]$, $T = [t_S]$, and $E = [s_E] = [t_E]$:

$$S \to EA \mid \mathtt{0}B \qquad E \to \varepsilon \mid \mathtt{0}X$$
$$A \to \mathtt{1}A \mid \mathtt{1}T \qquad X \to EY$$
$$B \to \mathtt{0}B \mid ET \qquad Y \to \mathtt{1}Z$$
$$T \to \varepsilon \qquad Z \to E$$

Our earlier proofs imply that we can forbid ε-transitions or even allow regular-expression transitions in our recursive automata without changing the set of languages they accept.

5.8 Chomsky Normal Form

For many algorithmic problems involving context-free grammars, it is helpful to consider grammars with a particular special structure called ***Chomsky normal form***, abbreviated ***CNF***:

- The starting non-terminal S does not appear on the right side of any production rule.
- The starting non-terminal S *may* have the production rule $S \to \varepsilon$.
- The right side of every other production rule is either a single terminal symbol or a string of exactly two non-terminals—that is, every other production rule has the form $A \to BC$ or $A \to a$.

A particularly attractive feature of CNF grammars is that they yield *full binary* parse trees; in particular, every parse tree for a string of length $n > 0$ has exactly $2n - 1$ non-terminal nodes. Consequently, any string of length n in the language of a CNF grammar can be derived in exactly $2n - 1$ production steps. It follows that we can actually determine whether a string belongs to the language of a CNF grammar by brute-force consideration of all possible derivations of the appropriate length.

For arbitrary context-free grammars, there is no similar upper bound on the length of a derivation, and therefore no similar brute-force membership algorithm, because the grammar may contain additional ***ε-productions*** of the form $A \to \varepsilon$ and/or ***unit productions*** of the form $A \to B$, where both A and B are non-terminals. Unit productions introduce nodes of degree 1 into any parse tree, and ε-productions introduce leaves that do not contribute to the word being parsed.

Fortunately, it *is* possible to determine membership in the language of an arbitrary context-free grammar, thanks to the following theorem. Two context-free grammars are ***equivalent*** if they define the same language.

Every context-free grammar is equivalent to a grammar in Chomsky normal form.

Moreover, there are algorithms to automatically convert any context-free grammar into Chomsky normal form. Unfortunately, these conversion algorithms are quite complex, but for most applications of context-free grammars, the details of the conversion are unimportant—it's enough to know that the algorithms exist. For the sake of completeness, however, I will describe one such conversion algorithm in the next section.

*5.9 CNF Conversion Algorithm

I'll actually prove a stronger statement: Not only can we convert any context-free grammar into Chomsky normal form, but we can do so *quickly*. We analyze the running time of our conversion algorithm in terms of the ***total length*** of the input grammar, which is just the number of symbols needed to write down the grammar. Up to constant factors, the total length is the sum of the lengths of the production rules.

Theorem 5.7. *Given an arbitrary context-free grammar with total length L, we can compute an equivalent grammar in Chomsky normal form with total length $O(L^2)$ in $O(L^2)$ time.*

Our algorithm consists of several relatively straightforward stages. Efficient implementation of some of these stages requires standard graph-traversal algorithms, which are described in a different part of the course.

0. Add a new starting non-terminal. Add a new non-terminal S' and a production rule $S' \to S$, where S is the starting non-terminal for the given grammar. S' will be the starting non-terminal for the resulting CNF grammar. (In fact, this step is necessary only when $S \rightsquigarrow^* \varepsilon$, but at this point in the conversion process, we don't yet know whether that's true.)

1. Decompose long production rules. For each production rule $A \to \omega$ whose right side w has length greater than two, add new production rules of length two that still permit the derivation $A \rightsquigarrow^* \omega$. Specifically, suppose $\omega = \alpha\chi$ for some symbol $\alpha \in \Sigma \cup \Gamma$ and string $\chi \in (\Sigma \cup \Gamma)^*$. The algorithm replaces $A \to \omega$ with two new production rules $A \to \alpha B$ and $B \to \chi$, where B is a new non-terminal, and then (if necessary) recursively decomposes the production rule $B \to \chi$. For example, we would replace the long production rule $A \to \mathtt{0}BC\mathtt{1}CB$ with the following sequence of short production rules, where each A_i is a new non-terminal:

$$A \to \mathtt{0}A_1 \qquad A_1 \to BA_2 \qquad A_2 \to CA_3 \qquad A_3 \to \mathtt{1}A_4 \qquad A_4 \to CB$$

This stage can significantly increase the number of non-terminals and production rules, but it increases the *total length* of all production rules by at most a small constant factor.[3] Moreover, for the remainder of the conversion algorithm, every production rule has length at most two. The running time of this stage is $O(L)$.

2. Identify nullable non-terminals. A non-terminal A is ***nullable*** if and only if $A \rightsquigarrow^* \varepsilon$. The recursive definition of $\rightsquigarrow^*$ implies that A is nullable if and only if the grammar contains a production rule $A \to \omega$ where ω consists entirely of nullable non-terminals (in particular, if $\omega = \varepsilon$). You may be tempted to transform this recursive characterization directly into a recursive algorithm, but this is a bad idea; the resulting algorithm would fall into an infinite loop if (for example) the same non-terminal appeared on both sides of the same production rule. Instead, we apply the following ***fixed-point*** algorithm, which repeatedly scans through the entire grammar until a complete scan discovers no new nullable non-terminals.

```
NULLABLES(Σ, Γ, R, S):
  Γ_ε ← ∅          ⟨⟨known nullable non-terminals⟩⟩
  done ← FALSE
  while ¬done
      done ← TRUE
      for each non-terminal A ∈ Γ \ Γ_ε
          for each production rule A → ω
              if ω ∈ Γ_ε^*
                  add A to Γ_ε
                  done ← FALSE
  return Γ_ε
```

At this point in the conversion algorithm, if S' is ***not*** identified as nullable, then we can safely remove it from the grammar and use the original starting nonterminal S instead.

As written, NULLABLES runs in $O(nL) = O(L^2)$ time, where n is the number of non-terminals in Γ. Each iteration of the main loop except the last adds at least one non-terminal to Γ_ε, so the

[3] In most textbook descriptions of the CFG conversion algorithm, this stage is performed *last*, after removing ε-productions and unit productions. But with the stages in that traditional order, removing ε-productions could *exponentially* increase the length of the grammar in the worst case! Consider the production rule $A \to (BC)^k$, where B is nullable but C is not. Decomposing this rule first and then removing ε-productions introduces about $3k$ new rules; whereas, removing ε-productions first introduces 2^k new rules, most of which then must then be further decomposed!

algorithm halts after at most $n+1 \leq L$ iterations, and in each iteration, we examine at most L production rules. There is a faster implementation of NULLABLES that runs in $O(n+L) = O(L)$ time,[4] but since other parts of the conversion algorithm already require $O(L^2)$ time, we needn't bother.

3. Eliminate ε-productions. First, remove every production rule of the form $A \to \varepsilon$. Then for each production rule $A \to w$, add all possible new production rules of the form $A \to w'$, where w' is a ***non-empty*** string obtained from w by removing one nullable non-terminal. For example, if if the grammar contained the production rule $A \to BC$, where B and C are both nullable, we would add two new production rules $A \to B \mid C$. Finally, if the starting nonterminal S' was identified as nullable in the previous stage, add the production rule $S' \to \varepsilon$; this will be the *only* ε-production in the final grammar. This phase of the conversion runs in $O(L)$ time and at most triples the number of production rules.

4. Merge equivalent non-terminals. We say that two non-terminals A and B are ***equivalent*** if they can be derived from each other: $A \rightsquigarrow^* B$ and $B \rightsquigarrow^* A$. Because we have already removed ε-productions, any such derivation must consist entirely of unit productions. For example, in the grammar

$$S \to B \mid C, \qquad A \to B \mid D \mid CC \mid \mathtt{0}, \qquad B \to C \mid AD \mid \mathtt{1}, \qquad C \to A \mid DA, \qquad D \to BA \mid CS,$$

non-terminals A, B, C are all equivalent, but S is not in that equivalence class (because we cannot derive S from A) and neither is D (because we cannot derive A from D).

Construct a directed graph G whose vertices are the non-terminals and whose edges correspond to unit productions, in $O(L)$ time. Then two non-terminals are equivalent if and only if they are in the same strong component of G. Compute the strong components of G in $O(L)$ time using, for example, the algorithm of Kosaraju and Sharir. Then merge all the non-terminals in each equivalence class into a single non-terminal. Finally, remove any unit productions of the form $A \to A$. The total running time for this phase is $O(L)$. Starting with our example grammar above, merging B and C with A and removing the production $A \to A$ gives us the simpler grammar

$$S \to A, \qquad A \to AA \mid D \mid DA \mid \mathtt{0} \mid \mathtt{1}, \qquad D \to AA \mid AS.$$

We could further simplify the grammar by merging all non-terminals reachable from S using only unit productions (in this case, merging non-terminals S and S), but this further simplification is unnecessary.

5. Remove unit productions. Once again, we construct a directed graph G whose vertices are the non-terminals and whose edges correspond to unit productions, in $O(L)$ time. Because no two non-terminals are equivalent, G is acyclic. Thus, using topological sort, we can index the non-terminals $A_1, A_2, \ldots, A_n$ such that for every unit production $A_i \to A_j$ we have $i < j$, again in $O(L)$ time; moreover, we can assume that the starting non-terminal is A_1. (In fact, both the dag G and the linear ordering of non-terminals was already computed in the previous phase!!)

Then for each index j in decreasing order, for each unit production $A_i \to A_j$ and each production $A_j \to \omega$, we add a new production rule $A_i \to \omega$. At this point, all unit productions are

[4] Consider the bipartite graph whose vertices correspond to non-terminals and the right sides of production rules, with one edge per rule. The faster algorithm is a modified breadth-first search of this graph, starting at the vertex representing ε.

redundant and can be removed. Applying this algorithm to our example grammar above gives us the grammar

$$S \to AA \mid AS \mid DA \mid \mathtt{0} \mid \mathtt{1}, \qquad A \to AA \mid AS \mid DA \mid \mathtt{0} \mid \mathtt{1}, \qquad D \to AA \mid AS.$$

In the worst case, each production rule for A_n is copied to each of the other $n-1$ non-terminals. Thus, this phase runs in $\Theta(nL) = O(L^2)$ time and increases the length of the grammar to $\Theta(nL) = O(L^2)$ in the worst case.

This phase dominates the running time of the CNF conversion algorithm. Unlike previous phases, no faster algorithm for removing unit transformations is known! There are grammars of length L with unit productions such that any equivalent grammar without unit productions has length $\Omega(L^{1.499999})$ (for any desired number of 9s), but this lower bound does not rule out the possibility of an algorithm that runs in only $O(L^{3/2})$ time. Closing the gap between $\Omega(L^{3/2-\varepsilon})$ and $O(L^2)$ has been an open problem since the early 1980s!

6. Protect terminals. Finally, for each terminal $a \in \Sigma$, we introduce a new non-terminal A_a and a new production rule $A_a \to a$, and then replace a with A_a in every production rule of length two.

This completes the conversion to Chomsky normal form! As claimed, the total running time of the algorithm is $O(L^2)$, and the total length of the output grammar is also $O(L^2)$.

To see the conversion algorithm in action, let's apply these stages one at a time to our very first example grammar for the language $\{\mathtt{0}^m\mathtt{1}^n \mid m \neq n\}$:

$$S \to A \mid B \qquad A \to \mathtt{0}A \mid \mathtt{0}C \qquad B \to B\mathtt{1} \mid C\mathtt{1} \qquad C \to \varepsilon \mid \mathtt{0}C\mathtt{1}$$

0. Add a new starting non-terminal S'.

$$\underline{S' \to S} \qquad S \to A \mid B \qquad A \to \mathtt{0}A \mid \mathtt{0}C \qquad B \to B\mathtt{1} \mid C\mathtt{1} \qquad C \to \varepsilon \mid \mathtt{0}C\mathtt{1}$$

1. Decompose the long production rule $C \to \mathtt{0}C\mathtt{1}$.

$$S' \to S \qquad S \to A \mid B \qquad A \to \mathtt{0}A \mid \mathtt{0}C \qquad B \to B\mathtt{1} \mid C\mathtt{1} \qquad \underline{C \to \varepsilon \mid \mathtt{0}D \qquad D \to C\mathtt{1}}$$

2. Identify C as the only nullable non-terminal. Because S' is not nullable, remove the production rule $S' \to S$.

3. Eliminate the ε-production $C \to \varepsilon$.

$$S \to A \mid B \qquad A \to \mathtt{0}A \mid \mathtt{0}C \underline{\mid \mathtt{0}} \qquad B \to B\mathtt{1} \mid C\mathtt{1} \underline{\mid \mathtt{1}} \qquad C \to \mathtt{0}D \qquad D \to C\mathtt{1} \underline{\mid \mathtt{1}}$$

4. No two non-terminals are equivalent, so there's nothing to merge.

5. Remove the unit productions $S' \to S$, $S \to A$, and $S \to B$.

$$S \to \underline{\mathtt{0}A \mid \mathtt{0}C \mid B\mathtt{1} \mid C\mathtt{1} \mid \mathtt{0} \mid \mathtt{1}}$$
$$A \to \mathtt{0}A \mid \mathtt{0}C \mid \mathtt{0} \qquad B \to B\mathtt{1} \mid C\mathtt{1} \mid \mathtt{1} \qquad C \to \mathtt{0}D \qquad D \to C\mathtt{1} \mid \mathtt{1}.$$

6. Finally, protect the terminals $\mathtt{0}$ and $\mathtt{1}$ to obtain the final CNF grammar.

$$S \to \underline{EA \mid EC \mid BF \mid CF \mid \mathtt{0} \mid \mathtt{1}}$$
$$A \to \underline{EA \mid EC} \mid \mathtt{0} \qquad B \to \underline{BF \mid CF} \mid \mathtt{1}$$
$$C \to \underline{ED} \qquad D \to \underline{CF} \mid \mathtt{1}$$
$$\underline{E \to \mathtt{0}} \qquad \underline{F \to \mathtt{1}}$$

Exercises

1. Describe context-free grammars that generate each of the following languages. The function $\#(x, w)$ returns the number of occurrences of the ***substring*** x in the string w. For example, $\#(\mathtt{0}, \mathtt{101001}) = 3$ and $\#(\mathtt{010}, \mathtt{1010100011}) = 2$. These are *not* listed in order of increasing difficulty.

   (a) All strings in $\{\mathtt{0}, \mathtt{1}\}^*$ whose length is divisible by 5.

   (b) All strings in $\{\mathtt{0}, \mathtt{1}\}^*$ representing a non-negative multiple of 5 in binary.

   (c) $\{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid \#(\mathtt{0}, w) = 2 \cdot \#(\mathtt{1}, w)\}$

   (d) $\{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid \#(\mathtt{0}, w) \neq 2 \cdot \#(\mathtt{1}, w)\}$

   (e) $\{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid \#(\mathtt{00}, w) = \#(\mathtt{11}, w)\}$

   (f) $\{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid \#(\mathtt{01}, w) = \#(\mathtt{10}, w)\}$

   (g) $\{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid \#(\mathtt{0}, w) = \#(\mathtt{1}, w)$ and $|w|$ is a multiple of $3\}$

   (h) $\{\mathtt{0}, \mathtt{1}\}^* \setminus \{\mathtt{0}^n \mathtt{1}^n \mid n \geq 0\}$

   (i) $\{\mathtt{0}^n \mathtt{1}^{2n} \mid n \geq 0\}$

   (j) $\{\mathtt{0}, \mathtt{1}\}^* \setminus \{\mathtt{0}^n \mathtt{1}^{2n} \mid n \geq 0\}$

   (k) $\{\mathtt{0}^n \mathtt{1}^m \mid 0 \leq 2m \leq n < 3m\}$

   (l) $\{\mathtt{0}^i \mathtt{1}^j \mathtt{2}^{i+j} \mid i, j \geq 0\}$

   (m) $\{\mathtt{0}^i \mathtt{1}^j \mathtt{2}^k \mid i = j \text{ or } j = k\}$

   (n) $\{\mathtt{0}^i \mathtt{1}^j \mathtt{2}^k \mid i \neq j \text{ or } j \neq k\}$

   (o) $\{\mathtt{0}^i \mathtt{1}^j \mathtt{0}^j \mathtt{1}^i \mid i, j \geq 0\}$

   (p) $\left\{w\#\mathtt{0}^{\#(\mathtt{0},w)} \;\middle|\; w \in \{\mathtt{0}, \mathtt{1}\}^*\right\}$

   (q) $\{xy \mid x, y \in \{\mathtt{0}, \mathtt{1}\}^* \text{ and } x \neq y \text{ and } |x| = |y|\}$

   (r) $\left\{x\#y^R \;\middle|\; x, y \in \{\mathtt{0}, \mathtt{1}\}^* \text{ and } x \neq y\right\}$

   (s) $\{x\#y \mid x, y \in \{\mathtt{0}, \mathtt{1}\}^* \text{ and } \#(\mathtt{0}, x) = \#(\mathtt{1}, y)\}$

   (t) $\{\mathtt{0}, \mathtt{1}\}^* \setminus \{ww \mid w \in \{\mathtt{0}, \mathtt{1}\}^*\}$

   (u) All strings in $\{\mathtt{0}, \mathtt{1}\}^*$ that are *not* palindromes.

   (v) All strings in $\{\mathtt{(}, \mathtt{)}, \diamond\}^*$ in which the parentheses are balanced and the symbol $\diamond$ appears at most four times. For example, $\mathtt{()(())}$ and $\mathtt{(}\diamond\diamond\mathtt{(()()}\diamond\mathtt{)()())}\diamond$ and $\diamond\diamond\diamond$ are strings in this language, but $\mathtt{)(()}$ and $\mathtt{(}\diamond\diamond\diamond\mathtt{)}\diamond\diamond$ are not.

2. Describe recursive automata for each of the languages in problem 1. ("Describe" does not necessarily mean "draw"!)

3. Prove that if L is a context-free language, then L^R is also a context-free language. *[Hint: How do you reverse a context-free grammar?]*

4. Consider a generalization of context-free grammars that allows any *regular expression* over $\Sigma \cup \Gamma$ to appear on the right side of a production rule. Without loss of generality, for each non-terminal $A \in \Gamma$, the generalized grammar contains a single regular expression $R(A)$. To

apply a production rule to a string, we replace any non-terminal A with an arbitrary word in the language described by $R(A)$. As usual, the language of the generalized grammar is the set of all strings that can be derived from its start non-terminal.

For example:, the following generalized context-free grammar describes the language of all regular expressions over the alphabet $\{\mathtt{0}, \mathtt{1}\}$:

$S \to (T\mathtt{+})^* T + \emptyset$	(Regular expressions)
$T \to \varepsilon + F^* F$	(Terms = summable expressions)
$F \to (\mathtt{0} + \mathtt{1} + (S))(\mathtt{*} + \varepsilon)$	(Factors = concatenable expressions)

Here is a parse tree for the regular expression `0+1(10*1+01*0)*10*` (which represents the set of all binary numbers divisible by 3):

Prove that every *generalized* context-free grammar describes a context-free language. In other words, show that allowing regular expressions to appear in production rules does not increase the expressive power of context-free grammars.

Think globally, act locally.

— Attributed to Patrick Geddes (c.1915), among many others.

We can only see a short distance ahead,
but we can see plenty there that needs to be done.

— Alan Turing, "Computing Machinery and Intelligence" (1950)

Never worry about theory
as long as the machinery does what it's supposed to do.

— Robert Anson Heinlein, *Waldo & Magic, Inc.* (1950)

It is a sobering thought that when Mozart was my age,
he had been dead for two years.

— Tom Lehrer, introduction to "Alma", *That Was the Year That Was* (1965)

6 Turing Machines

In 1936, a few months before his 24th birthday, Alan Turing launched computer science as a modern intellectual discipline. In a single remarkable paper, Turing provided the following results:

- A simple formal model of mechanical computation now known as *Turing machines*.
- A description of a single *universal* machine that can be used to compute *any* function computable by *any* other Turing machine.
- A proof that no Turing machine can solve the *halting problem*—Given the formal description of an arbitrary Turing machine M, does M halt or run forever?
- A proof that no Turing machine can determine whether an arbitrary given proposition is provable from the axioms of first-order logic. This is Hilbert and Ackermann's famous *Entscheidungsproblem* ("decision problem").
- Compelling arguments[1] that his machines can execute *arbitrary* "calculation by finite means".

Although Turing did not know it at the time, he was not the first to prove that the *Entscheidungsproblem* had no algorithmic solution. The first such proof is implicit in the work of Kurt Gödel; in lectures on his incompleteness theorems[2] at Princeton in 1934, Gödel described a model of ***general recursive functions***, which he largely credited to Jacques Herbrand.[3] Gödel's incompleteness theorem can be viewed as a proof that some propositions cannot be proved using general recursive functions. However, Gödel viewed his definition as a "heuristic principle" rather than an accurate model of effective computation, or even a complete definition of recursion.

The first *published* proof was written by Alonzo Church and published just a new months before Turing's paper, using another different model of computation, now called the ***untyped λ-calculus***. Turing and Church developed their results independently; indeed, Turing rushed

[1] As Turing put it, "All arguments which can be given are bound to be, fundamentally, appeals to intuition, and for this reason rather unsatisfactory mathematically." The claim that anything that can be computed can be computing using Turing machines is now known as the ***Church-Turing thesis***.

[2] which he published at the ripe old age of 25

[3] Herbrand was a brilliant French mathematician who was killed in a mountain-climbing accident at the age of 23.

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

the submission of his own paper immediately after receiving a copy of Church's paper, pausing only long enough to prove that any function computable via λ-calculus can also be computed by a Turing machine and vice versa.[4] Church was the referee for Turing's paper; between the paper's submission and its acceptance, Turing was admitted to Princeton as a PhD student, where Church became his advisor. He finished his PhD two years later.

Informally, Turing described an abstract machine with a finite number of ***internal states*** that has access to "memory" in the form of a ***tape***. The tape consists of a semi-infinite sequence of ***cells***, each containing a single symbol from some arbitrary finite alphabet. The Turing machine can access the tape only through its ***head***, which is positioned over a single cell. Initially, the tape contains an arbitrary finite ***input string*** followed by an infinite sequence of ***blanks***, and the head is positioned over the first cell on the tape. In a single iteration, the machine reads the symbol in that cell, possibly write a new symbol into that cell, possibly changes its internal state, possibly moves the head to a neighboring cell, and possibly halts. The precise behavior of the machine at each iteration is entirely determined by its internal state and the symbol that it reads. When the machine halts, it indicates whether it has ***accepted*** or ***rejected*** the original input string.

A few iterations of a six-state Turing machine.

6.1 Why Bother?

Students used to thinking of computation in terms of higher-level operations like random memory accesses, function calls, and recursion may wonder why we should even consider a model as simple and constrained as Turing machines. Admittedly, Turing machines are a terrible model for thinking about ***fast*** computation; simple operations that take constant time in the standard

[4] At roughly the same time as Turing's paper, Church and his more recent PhD graduate Steven Kleene independently proved that all general recursive functions are definable in the λ-calculus and vice versa. At the time, Kleene was 27, and Church was 33.

random-access model can require *arbitrarily* many steps on a Turing machine. Worse, seemingly minor variations in the precise definition of "Turing machine" can have significant impact on problem complexity. As a simple example (which will make more sense later), we can reverse a string of n bits in $O(n)$ time using a two-tape Turing machine, but the same task provably requires $\Omega(n^2)$ time on a single-tape machine.

But here we are not interested in finding *fast* algorithms, or indeed in finding algorithms at all, but rather in proving that some problems cannot be solved by *any* computational means. Such a bold claim requires a formal definition of "computation" that is simple enough to support formal argument, but still powerful enough to describe arbitrary algorithms. Turing machines are ideal for this purpose. In particular, Turing machines are powerful enough to *simulate other Turing machines*, while still simple enough to let us build up this self-simulation from scratch, unlike more complex but efficient models like the standard random-access machine

(Arguably, self-simulation is even simpler in Church's λ-calculus, or in Schönfinkel and Curry's combinator calculus, which is one of many reasons those models are more common in the design and analysis of programming languages than Turing machines. Those models are much more abstract; in particular, they are harder to show equivalent to standard iterative models of computation.)

6.2 Formal Definitions

Formally, a Turing machine consists of the following components. (Hang on; it's a long list.)

- An arbitrary finite set Γ with at least two elements, called the ***tape alphabet***.
- An arbitrary symbol $\square \in \Gamma$, called the ***blank symbol*** or just the ***blank***.
- An arbitrary nonempty subset $\Sigma \subseteq (\Gamma \setminus \{\square\})$, called the ***input alphabet***.
- Another arbitrary finite set Q whose elements are called ***states***.
- Three distinct special states $\mathsf{start}, \mathsf{accept}, \mathsf{reject} \in Q$.
- A ***transition*** function $\delta : (Q \setminus \{\mathsf{accept}, \mathsf{reject}\}) \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$.

A ***configuration*** or ***global state*** of a Turing machine is represented by a triple $(q, x, i) \in Q \times \Gamma^* \times \mathbb{N}$, indicating that the machine's internal state is q, the tape contains the string x followed by an infinite sequence of blanks, and the head is located at position i. Trailing blanks in the tape string are ignored; the triples (q, x, i) and $(q, x\square, i)$ describe exactly the same configuration.

The transition function δ describes the evolution of the machine. For example, $\delta(q, a) = (p, b, -1)$ means that when the machine reads symbol a in state q, it changes its internal state to p, writes symbol b onto the tape at its current location (replacing a), and then decreases its position by 1 (or more intuitively, moves one step to the left). If the position of the head becomes negative, no further transitions are possible, and the machine ***crashes***.

We write $\boldsymbol{(p, x, i) \Rightarrow_M (q, y, j)}$ to indicate that Turing machine M transitions from the first configuration to the second in one step. (The symbol $\Rightarrow$ is often pronounced "yields"; I will omit the subscript M if the machine is clear from context.) For example, $\delta(p, a) = (q, b, \pm 1)$ means that

$$(p, xay, i) \;\Rightarrow\; (q, xby, i \pm 1)$$

for any non-negative integer i, any string x of length i, and any string y. The evolution of any Turing machine is ***deterministic***; each configuration C yields a *unique* configuration C'. We write $\boldsymbol{C \Rightarrow^* C'}$ to indicate that there is a (possibly empty) sequence of transitions from configuration C to configuration C'. (The symbol $\Rightarrow^*$ can be pronounced "eventually yields".)

The initial configuration is $(\mathsf{start}, w, 0)$ for some arbitrary (and possibly empty) ***input string*** $w \in \Sigma^*$. If M eventually reaches the accept state—more formally, if $(\mathsf{start}, w, 0) \Rightarrow^* (\mathsf{accept}, x, i)$ for some string $x \in \Gamma^*$ and some integer i—we say that M ***accepts*** the original input string w. Similarly, if M eventually reaches the reject state, we say that M ***rejects*** w. We must emphasize that "rejects" and "does not accept" are *not* synonyms; if M crashes or runs forever, then M neither accepts nor rejects w.

We distinguish between two different senses in which a Turing machine can "accept" a language. Let M be a Turing machine with input alphabet Σ, and let $L \subseteq \Sigma^*$ be an arbitrary language over Σ.

- M ***recognizes*** or ***accepts*** L if and only if M accepts every string in L but nothing else. A language is ***recognizable*** (or *semi-computable* or *recursively enumerable*) if it is recognized by some Turing machine.

- M ***decides*** L if and only if M accepts every string in L and rejects every string in $\Sigma^* \setminus L$. Equivalently, M decides L if and only if M recognizes L and halts (without crashing) on all inputs. A language is ***decidable*** (or *computable* or *recursive*) if it is decided by some Turing machine.

Trivially, every decidable language is recognizable, but (as we will see later), not every recognizable language is decidable.

6.3 A First Example

Consider the language $L = \{0^n 1^n 0^n \mid n \geq 0\}$. This language is neither regular nor context-free, but it can be decided by the following six-state Turing machine. The alphabets and states of the machine are defined as follows:

$$\Gamma = \{0, 1, \$, x, \square\}$$
$$\Sigma = \{0, 1\}$$
$$Q = \{\mathsf{start}, \mathsf{seek1}, \mathsf{seek0}, \mathsf{reset}, \mathsf{verify}, \mathsf{accept}, \mathsf{reject}\}$$

The transition function is described in the table on the next page; all unspecified transitions lead to the reject state. The figure below shows a graphical representation of the same machine, which resembles a drawing of a DFA, but with output symbols and actions specified on each edge. For example, we indicate the transition $\delta(p, 0) = (q, 1, +1)$ by writing $0/1, +1$ next to the arrow from state p to state q.

A graphical representation of the example Turing machine

$\delta(\ p\ , a) = (\ q\ , b, \Delta)$	explanation
$\delta(\texttt{start}, \texttt{0}) = (\texttt{seek1}, \texttt{\$}, +1)$	mark first 0 and scan right
$\delta(\texttt{start}, \texttt{x}) = (\texttt{verify}, \texttt{\$}, +1)$	looks like we're done, but let's make sure
$\delta(\texttt{seek1}, \texttt{0}) = (\texttt{seek1}, \texttt{0}, +1)$	scan rightward for 1
$\delta(\texttt{seek1}, \texttt{x}) = (\texttt{seek1}, \texttt{x}, +1)$	
$\delta(\texttt{seek1}, \texttt{1}) = (\texttt{seek0}, \texttt{x}, +1)$	mark 1 and continue right
$\delta(\texttt{seek0}, \texttt{1}) = (\texttt{seek0}, \texttt{1}, +1)$	scan rightward for 0
$\delta(\texttt{seek0}, \texttt{x}) = (\texttt{seek0}, \texttt{x}, +1)$	
$\delta(\texttt{seek0}, \texttt{0}) = (\texttt{reset}, \texttt{x}, +1)$	mark 0 and scan left
$\delta(\texttt{reset}, \texttt{0}) = (\texttt{reset}, \texttt{0}, -1)$	scan leftward for \$
$\delta(\texttt{reset}, \texttt{1}) = (\texttt{reset}, \texttt{1}, -1)$	
$\delta(\texttt{reset}, \texttt{x}) = (\texttt{reset}, \texttt{x}, -1)$	
$\delta(\texttt{reset}, \texttt{\$}) = (\texttt{start}, \texttt{\$}, +1)$	step right and start over
$\delta(\texttt{verify}, \texttt{x}) = (\texttt{verify}, \texttt{\$}, +1)$	scan right for any unmarked symbol
$\delta(\texttt{verify}, \square) = (\texttt{accept}, \square, -1)$	success!

The transition function for a Turing machine that decides the language $\{\texttt{0}^n\texttt{1}^n\texttt{0}^n \mid n \geq 0\}$.

Finally, we trace the execution of this machine on two input strings: $\texttt{001100} \in L$ and $\texttt{00100} \notin L$. In each configuration, we indicate the position of the head using a small triangle instead of listing the position explicitly. Notice that we automatically add blanks to the tape string as necessary. Proving that this machine actually decides L—and in particular, that it never crashes or infinite-loops—is a straightforward but tedious exercise in induction.

6.4 Variations

There are actually several formal models that all fall under the name "Turing machine", each with small variations on the definition we've given. Although we do need to be explicit about *which* variant we want to use for any particular problem, the differences between the variants are relatively unimportant. For any machine defined in one model, there is an equivalent machine in each of the other models; in particular, all of these variants recognize the same languages and decide the same languages. For example:

- **Halting conditions.** Some models allow multiple accept and reject states, which (depending on the precise model) trigger acceptance or rejection either when the machine enters the state, or when the machine has no valid transitions out of such a state. Others include only explicit accept states, and either equate crashing with rejection or do not define a rejection mechanism at all. Still other models include halting as one of the possible *actions* of the machine, in addition to moving left or moving right; in these models, the machine accepts/rejects its input if and only if it halts in an accepting/non-accepting state.

- **Actions.** Some Turing machine models allow transitions that do not move the head, or that move the head by more than one cell in a single step. Others insist that a single step of the machine *either* writes a new symbol onto the tape *or* moves the head one step. Finally, as mentioned above, some models include halting as one of the available actions.

- **Transition function.** Some models of Turing machines, including Turing's original definition, allow the transition function to be undefined on some state-symbol pairs. In this

$$
\begin{aligned}
&(\text{start}, \underset{\blacktriangle}{0}01100)\\
\Rightarrow\ &(\text{seek1}, \$\underset{\blacktriangle}{0}1100)\\
\Rightarrow\ &(\text{seek1}, \$0\underset{\blacktriangle}{1}100)\\
\Rightarrow\ &(\text{seek0}, \$0x\underset{\blacktriangle}{1}00)\\
\Rightarrow\ &(\text{seek0}, \$0x1\underset{\blacktriangle}{0}0)\\
\Rightarrow\ &(\text{reset}, \$0x\underset{\blacktriangle}{1}x0)\\
\Rightarrow\ &(\text{reset}, \$0\underset{\blacktriangle}{x}1x0)\\
\Rightarrow\ &(\text{reset}, \$\underset{\blacktriangle}{0}x1x0)\\
\Rightarrow\ &(\text{reset}, \underset{\blacktriangle}{\$}0x1x0)\\
\Rightarrow\ &(\text{start}, \$\underset{\blacktriangle}{0}x1x0)\\
\Rightarrow\ &(\text{seek1}, \$\$\underset{\blacktriangle}{x}1x0)\\
\Rightarrow\ &(\text{seek1}, \$\$x\underset{\blacktriangle}{1}x0)\\
\Rightarrow\ &(\text{seek0}, \$\$xx\underset{\blacktriangle}{x}0)\\
\Rightarrow\ &(\text{seek0}, \$\$xxx\underset{\blacktriangle}{0})\\
\Rightarrow\ &(\text{reset}, \$\$xx\underset{\blacktriangle}{x}x)\\
\Rightarrow\ &(\text{reset}, \$\$x\underset{\blacktriangle}{x}xx)\\
\Rightarrow\ &(\text{reset}, \$\$\underset{\blacktriangle}{x}xxx)\\
\Rightarrow\ &(\text{reset}, \$\underset{\blacktriangle}{\$}xxxx)\\
\Rightarrow\ &(\text{start}, \$\$\underset{\blacktriangle}{x}xxx)\\
\Rightarrow\ &(\text{verify}, \$\$\$\underset{\blacktriangle}{x}xx)\\
\Rightarrow\ &(\text{verify}, \$\$\$\$\underset{\blacktriangle}{x}x)\\
\Rightarrow\ &(\text{verify}, \$\$\$\$\$\underset{\blacktriangle}{x})\\
\Rightarrow\ &(\text{verify}, \$\$\$\$\$\$\underset{\blacktriangle}{\square})\\
\Rightarrow\ &(\text{accept}, \$\$\$\$\$\underset{\blacktriangle}{\$}) \Rightarrow \textbf{accept!}
\end{aligned}
$$

The evolution of the example Turing machine on the input string $001100 \in L$

$$
\begin{aligned}
&(\text{start}, \underset{\blacktriangle}{0}0100)\\
\Rightarrow\ &(\text{seek1}, \$\underset{\blacktriangle}{0}100)\\
\Rightarrow\ &(\text{seek1}, \$0\underset{\blacktriangle}{1}00)\\
\Rightarrow\ &(\text{seek0}, \$0x\underset{\blacktriangle}{0}0)\\
\Rightarrow\ &(\text{reset}, \$0\underset{\blacktriangle}{x}x0)\\
\Rightarrow\ &(\text{reset}, \$\underset{\blacktriangle}{0}xx0)\\
\Rightarrow\ &(\text{reset}, \underset{\blacktriangle}{\$}0xx0)\\
\Rightarrow\ &(\text{start}, \$\underset{\blacktriangle}{0}xx0)\\
\Rightarrow\ &(\text{seek1}, \$\$\underset{\blacktriangle}{x}x0)\\
\Rightarrow\ &(\text{seek1}, \$\$x\underset{\blacktriangle}{x}0)\\
\Rightarrow\ &(\text{seek1}, \$\$xx\underset{\blacktriangle}{0}) \Rightarrow \textbf{reject!}
\end{aligned}
$$

The evolution of the example Turing machine on the input string $00100 \notin L$

formulation, the transition function is given by a set $\delta \subset Q \times \Gamma \times Q \times \Gamma \times \{+1, -1\}$, such that for each state q and symbol a, there is at most one transition $(q, a, \cdot, \cdot, \cdot) \in \delta$. If the machine enters a configuration from which there is no transition, it halts and (depending on the precise model) either crashes or rejects. Others define the transition function as $\delta : Q \times \Gamma \to Q \times (\Gamma \cup \{-1, +1\})$, allowing the machine to *either* write a symbol to the tape *or* move the head in each step.

- **Beginning of the tape.** Some models forbid the head to move past the beginning of the tape, either by starting the tape with a special symbol that cannot be overwritten and that forces a rightward transition, or by declaring that a leftward transition at position 0 leaves the head in position 0, or even by pure fiat—declaring any machine that performs a leftward move at position 0 to be invalid.

To prove that any two of these variant "species" of Turing machine are equivalent, we must show how to transform a machine of one species into a machine of the other species that accepts and rejects the same strings. For example, let $M = (\Gamma, \square, \Sigma, Q, s, \mathsf{accept}, \mathsf{reject}, \delta)$ be a Turing machine with explicit accept and reject states. We can define an equivalent Turing machine M' that halts only when it moves left from position 0, and accepts only by halting while in an accepting state, as follows. We define the set of accepting states for M' as $A = \{\mathsf{accept}\}$ and define a new transition function

$$\delta'(q, a) := \begin{cases} (\mathsf{accept}, a, -1) & \text{if } q = \mathsf{accept} \\ (\mathsf{reject}, a, -1) & \text{if } q = \mathsf{reject} \\ \delta(q, a) & \text{otherwise} \end{cases}$$

Similarly, suppose someone gives us a Turing machine $M = (\Gamma, \square, \Sigma, Q, s, \mathsf{accept}, \mathsf{reject}, \delta)$ whose transition function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$ allows the machine to transition without moving its head. Then we can construct an equivalent Turing machine $M' = (\Gamma, \square, \Sigma, Q', s, \mathsf{accept}, \mathsf{reject}, \delta')$ that moves its head at every transition by defining $Q' := Q \times \{0, 1\}$ and

$$\delta'((p, 0), a) := \begin{cases} ((q, 1), b, +1) & \text{if } \delta(p, a) = (q, b, 0), \\ ((q, 0), b, \Delta) & \text{if } \delta(p, a) = (q, b, \Delta) \text{ and } \Delta \neq 0, \end{cases}$$

$$\delta'((p, 1), a) := ((p, 0), a, -1).$$

6.5 Computing Functions

Turing machines can also be used to compute functions from strings to strings, instead of just accepting or rejecting strings. Since we don't care about acceptance or rejection, we replace the explicit accept and reject states with a single halt state, and we define the ***output*** of the Turing machine to be the contents of the tape when the machine halts, after removing the infinite sequence of trailing blanks. More formally, for any Turing machine M, any string $w \in \Sigma^*$, and any string $x \in \Gamma^*$ that does not end with a blank, we write $\boldsymbol{M(w) = x}$ if and only if $(w, s, 0) \Rightarrow_M^* (x, \mathsf{halt}, i)$ for some integer i. If M does not halt on input w, then we write $\boldsymbol{M(w)} \nearrow$, which can be read either "M diverges on w" or "$M(w)$ is undefined." We say that M ***computes*** the function $f : \Sigma^* \to \Sigma^*$ if and only if $M(w) = f(w)$ for every string w.

6.5.1 Shifting

One basic operation that is used in many Turing machine constructions is ***shifting*** the input string a constant number of steps to the right or to the left. For example, given any input

string $w \in \{0, 1\}^*$, we can compute the string $0w$ using a Turing machine with tape alphabet $\Gamma = \{0, 1, \square\}$, state set $Q = \{0, 1, \mathsf{halt}\}$, start state 0, and the following transition function:

$$
\begin{array}{rcl}
\delta(p,\ a) & = & (\ q\ ,\ b,\ \Delta\ ) \\
\hline
\delta(0,\ 0) & = & (\ 0\ ,\ 0,\ +1) \\
\delta(0,\ 1) & = & (\ 1\ ,\ 0,\ +1) \\
\delta(0,\ \square) & = & (\mathsf{halt},\ 0,\ +1) \\
\hline
\delta(1,\ 0) & = & (\ 0\ ,\ 1,\ +1) \\
\delta(1,\ 1) & = & (\ 1\ ,\ 1,\ +1) \\
\delta(1,\ \square) & = & (\mathsf{halt},\ 1,\ +1)
\end{array}
$$

By increasing the number of states, we can build a Turing machine that shifts the input string any fixed number of steps in either direction. For example, a machine that shifts its input to the left by five steps might read the string from right to left, storing the five most recently read symbols in its internal state. A typical transition for such a machine would be $\delta(12345, 0) = (01234, 5, -1)$.

6.5.2 Binary Addition

With a more complex Turing machine, we can implement binary addition. The input is a string of the form $w{+}x$, where $w, x \in \{0, 1\}^n$, representing two numbers in binary; the output is the binary representation of $w+x$. To simplify our presentation, we assume that $|w| = |x| > 0$; however, this restrictions can be removed with the addition of a few more states. The following figure shows the entire Turing machine at a glance. The machine uses the tape alphabet $\Gamma = \{\square, 0, 1, +, \underline{0}, \underline{1}\}$; the start state is $\mathsf{shift0}$. All missing transitions go to a fail state, indicating that the input was badly formed.

Execution of this Turing machine proceeds in several phases, each with its own subset of states, as indicated in the figure. The initialization phase scans the entire input, shifting it to the right to make room for the output string, marking the rightmost bit of w, and reading and erasing the last bit of x.

$$
\begin{array}{rcl}
\delta(\ \ p\ \ ,\ a) & = & (\ \ q\ \ ,\ b,\ \Delta\ ) \\
\hline
\delta(\mathsf{shift0},\ 0) & = & (\mathsf{shift0},\ 0,\ +1) \\
\delta(\mathsf{shift0},\ 1) & = & (\mathsf{shift1},\ 0,\ +1) \\
\delta(\mathsf{shift0},\ +) & = & (\mathsf{shift+},\ \underline{0},\ +1) \\
\delta(\mathsf{shift0},\ \square) & = & (\ \mathsf{add0}\ ,\ \square,\ -1) \\
\hline
\delta(\mathsf{shift1},\ 0) & = & (\mathsf{shift0},\ 1,\ +1) \\
\delta(\mathsf{shift1},\ 1) & = & (\mathsf{shift1},\ 1,\ +1) \\
\delta(\mathsf{shift1},\ +) & = & (\mathsf{shift+},\ \underline{1},\ +1) \\
\delta(\mathsf{shift1},\ \square) & = & (\ \mathsf{add1}\ ,\ \square,\ -1) \\
\hline
\delta(\mathsf{shift+},\ 0) & = & (\mathsf{shift0},\ +,\ +1) \\
\delta(\mathsf{shift+},\ 1) & = & (\mathsf{shift1},\ +,\ +1)
\end{array}
$$

The first part of the main loop scans left to the marked bit of w, adds the bit of x that was just erased plus the carry bit from the previous iteration, and records the carry bit for the next iteration in the machines internal state.

A Turing machine that adds two binary numbers of the same length.

$\delta(\ p\ ,\ a) = (\ q\ ,\ b,\ \Delta)$	$\delta(\ p\ ,\ a) = (\ q\ ,\ b,\ \Delta)$	$\delta(\ p\ ,\ a) = (\ q\ ,\ b,\ \Delta)$
$\delta(\texttt{add0},\ \texttt{0}) = (\texttt{add0},\ \texttt{0},\ -1)$	$\delta(\texttt{add1},\ \texttt{0}) = (\texttt{add1},\ \texttt{0},\ -1)$	$\delta(\texttt{add2},\ \texttt{0}) = (\texttt{add2},\ \texttt{0},\ -1)$
$\delta(\texttt{add0},\ \texttt{1}) = (\texttt{add0},\ \texttt{0},\ -1)$	$\delta(\texttt{add1},\ \texttt{1}) = (\texttt{add1},\ \texttt{0},\ -1)$	$\delta(\texttt{add2},\ \texttt{1}) = (\texttt{add2},\ \texttt{0},\ -1)$
$\delta(\texttt{add0},\ \texttt{+}) = (\texttt{add0},\ \texttt{0},\ -1)$	$\delta(\texttt{add1},\ \texttt{+}) = (\texttt{add1},\ \texttt{0},\ -1)$	$\delta(\texttt{add2},\ \texttt{+}) = (\texttt{add2},\ \texttt{0},\ -1)$
$\delta(\texttt{add0},\ \underline{\texttt{0}}) = (\texttt{back0},\ \texttt{0},\ -1)$	$\delta(\texttt{add1},\ \underline{\texttt{0}}) = (\texttt{back0},\ \texttt{1},\ -1)$	$\delta(\texttt{add2},\ \underline{\texttt{0}}) = (\texttt{back1},\ \texttt{0},\ -1)$
$\delta(\texttt{add0},\ \underline{\texttt{1}}) = (\texttt{back0},\ \texttt{1},\ -1)$	$\delta(\texttt{add1},\ \underline{\texttt{1}}) = (\texttt{back1},\ \texttt{0},\ -1)$	$\delta(\texttt{add2},\ \underline{\texttt{1}}) = (\texttt{back1},\ \texttt{1},\ -1)$

The second part of the main loop marks the previous bit of w, scans right to the end of x, and then reads and erases the last bit of x, all while maintaining the carry bit.

$\delta(\ p\ ,\ a) = (\ q\ ,\ b,\ \Delta)$	$\delta(\ p\ ,\ a) = (\ q\ ,\ b,\ \Delta)$
$\delta(\texttt{back0},\ \texttt{0}) = (\texttt{next0},\ \underline{\texttt{0}},\ +1)$	$\delta(\texttt{back1},\ \texttt{0}) = (\texttt{next1},\ \underline{\texttt{0}},\ +1)$
$\delta(\texttt{back0},\ \texttt{1}) = (\texttt{next0},\ \underline{\texttt{1}},\ +1)$	$\delta(\texttt{back1},\ \texttt{1}) = (\texttt{next1},\ \underline{\texttt{1}},\ +1)$
$\delta(\texttt{next0},\ \texttt{0}) = (\texttt{next0},\ \texttt{0},\ +1)$	$\delta(\texttt{next1},\ \texttt{0}) = (\texttt{next1},\ \texttt{0},\ +1)$
$\delta(\texttt{next0},\ \texttt{1}) = (\texttt{next0},\ \texttt{0},\ +1)$	$\delta(\texttt{next1},\ \texttt{1}) = (\texttt{next1},\ \texttt{0},\ +1)$
$\delta(\texttt{next0},\ \texttt{+}) = (\texttt{next0},\ \texttt{0},\ +1)$	$\delta(\texttt{next1},\ \texttt{+}) = (\texttt{next1},\ \texttt{0},\ +1)$
$\delta(\texttt{next0},\ \square) = (\texttt{get0},\ \square,\ -1)$	$\delta(\texttt{next1},\ \square) = (\texttt{get1},\ \square,\ -1)$
$\delta(\texttt{get0},\ \texttt{0}) = (\texttt{add0},\ \square,\ -1)$	$\delta(\texttt{get1},\ \texttt{0}) = (\texttt{add1},\ \square,\ -1)$
$\delta(\texttt{get0},\ \texttt{1}) = (\texttt{add1},\ \square,\ -1)$	$\delta(\texttt{get1},\ \texttt{1}) = (\texttt{add2},\ \square,\ -1)$
$\delta(\texttt{get0},\ \texttt{+}) = (\texttt{last0},\ \square,\ -1)$	$\delta(\texttt{get1},\ \texttt{+}) = (\texttt{last1},\ \square,\ -1)$

Finally, after erasing the + in the last iteration of the main loop, the termination phase adds the last carry bit to the leftmost output bit and halts.

$$\begin{array}{rcl}
\delta(\ p\ ,\ a) & = & (\ q\ ,\ b,\ \Delta\ ) \\
\hline
\delta(\mathsf{last0},\ \mathtt{0}) & = & (\mathsf{last0},\ \mathtt{0},\ -1) \\
\delta(\mathsf{last0},\ \mathtt{1}) & = & (\mathsf{last0},\ \mathtt{0},\ -1) \\
\delta(\mathsf{last0},\ \underline{\mathtt{0}}) & = & (\ \mathsf{halt}\ ,\ \mathtt{0},\ \quad\ ) \\
\hline
\delta(\mathsf{last1},\ \mathtt{0}) & = & (\mathsf{last1},\ \mathtt{0},\ -1) \\
\delta(\mathsf{last1},\ \mathtt{1}) & = & (\mathsf{last1},\ \mathtt{0},\ -1) \\
\delta(\mathsf{last1},\ \underline{\mathtt{0}}) & = & (\ \mathsf{halt}\ ,\ \mathtt{1},\ \quad\ )
\end{array}$$

6.6 Variations on Tracks, Heads, and Tapes

Multiple Tracks

It is sometimes convenient to endow the Turing machine tape with multiple ***tracks***, each with its own tape alphabet, and allow the machine to read from and write to the same position on all tracks simultaneously. For example, to define a Turing machine with three tracks, we need three tape alphabets Γ_1, Γ_2, and Γ_3, each with its own blank symbol, where (say) Γ_1 contains the input alphabet Σ as a subset; we also need a transition function of the form

$$\delta : Q \times \Gamma_1 \times \Gamma_2 \times \Gamma_3 \to Q \times \Gamma_1 \times \Gamma_2 \times \Gamma_3 \times \{-1, +1\}$$

Describing a configuration of this machine requires a quintuple (q, x_1, x_2, x_3, i), indicating that each track i contains the string x_i followed by an infinite sequence of blanks. The initial configuration is $(\mathsf{start}, w, \varepsilon, \varepsilon, 0)$, with the input string written on the first track, and the other two tracks completely blank.

But any such machine is equivalent (if not *identical*) to a single-track Turing machine with the (still finite!) tape alphabet $\Gamma := \Gamma_1 \times \Gamma_2 \times \Gamma_3$. Instead of thinking of the tape as three infinite sequences of symbols, we think of it as a single infinite sequence of "records", each containing three symbols. Moreover, there's nothing special about the number 3 in this construction; a Turing machine with *any* constant number of tracks is equivalent to a single-track machine.

Doubly-Infinite Tape

It is also sometimes convenient to allow the tape to be infinite in both directions, for example, to avoid boundary conditions. There are several ways to simulate a doubly-infinite tape on a machine with only a semi-infinite tape. Perhaps the simplest method is to use a semi-infinite tape with two tracks, one containing the cells with positive index and the other containing the cells with negative index in reverse order, with a special marker symbol at position zero to indicate the transition.

0	+1	+2	+3	+4	$\cdots$
►	−1	−2	−3	−4	$\cdots$

Another method is to shuffle the positive-index and negative-index cells onto a single track, and add additional states to allow the Turing machine to move two steps in a single transition. Again, we need a special symbol at the left end of the tape to indicate the transition:

►	0	−1	+1	−2	+2	−3	+3	$\cdots$

A third method maintains two sentinel symbols ► and ◄ that surround all other non-blank symbols on the tape. Whenever the machine reads the right sentinel ◄, we write a blank, move right, write ◄, move left, and then proceed as if we had just read a blank. On the other hand, when the machine reads the left sentinel ►, we shift the entire contents of the tape (up to and

including the right sentinel) one step to the right, then move back to the left sentinel, move right, write a blank, and finally proceed as if we had just read a blank. Since the Turing machine does not actually have access to the position of the head *as an integer*, shifting the head and the tape contents one step right has no effect on its future evolution.

►	−3	−2	−1	0	+1	+2	+3	+4	+5	◄	···

Using either of the first two methods, we can simulate t steps of an arbitrary Turing machine with a doubly-infinite tape using only $O(t)$ steps on a standard Turing machine. The third method, unfortunately, requires $\Theta(t^2)$ steps in the worst case.

Insertion and Deletion

We can also allow Turing machines to insert and delete cells on the tape, in addition to simply overwriting existing symbols. We've already seen how to insert a new cell: Leave a special mark on the tape (perhaps in a second track), shift everything to the right of this mark one cell to the right, scan left to the mark, erase the mark, and finally write the correct character into the new cell. Deletion is similar: Mark the cell to be deleted, shift everything to the right of the mark one step to the left, scan left to the mark, and erase the mark. We may also need to maintain a mark in some cell to the right every non-blank symbol, indicating that all cells further to the right are blank, so that we know when to stop shifting left or right.

Multiple Heads

Another convenient extension is to allow machines simultaneous access to more than one position on the tape. For example, to define a Turing machine with *three* heads, we need a transition function of the form

$$\delta : Q \times \Gamma^3 \to Q \times \Gamma^3 \times \{-1, +1\}^3.$$

Describing a configuration of such a machine requires a quintuple (q, x, i, j, k), indicating that the machine is in state q, the tape contains string x, and the three heads are at positions i, j, k. The transition function tells us, given q and the three symbols $x[i], x[j], x[k]$, which three symbols to write on the tape and which direction to move each of the heads.

We can simulate this behavior with a single head by adding additional tracks to the tape that record the positions of each head. To simulate a machine M with three heads, we use a tape with four tracks: track 0 is the actual work tape; each of the remaining tracks has a single non-blank symbol recording the position of one of the heads. We also insert a special marker symbols at the left end of the tape.

►	M	Y	W	O	R	K	T	A	P	E	···
►									▲		···
►		▲									···
►						▲					···

We can simulate any single transition of M, starting with our single head at the left end of the tape, as follows. Throughout the simulation, we maintain the internal state of M as one of the components of our current state. First, for each i, we read the symbol under the ith head of M as follows:

> Scan to the right to find the mark on track i, read the corresponding symbol from track 0 into our internal state, and then return to the left end of the tape.

At this point, our internal state records M's current internal state and the three symbols under M's heads. After one more transition (using M's transition function), our internal state records M's *next* state, the symbol to be written by each head, and the direction to move each head. Then, for each i, we write with and move the ith head of M as follows:

> Scan to the right to find the mark on track i, write the correct symbol onto on track 0, move the mark on track i one step left or right, and then return to the left end of the tape.

Again, there is nothing special about the number 3 here; we can simulate machines with *any* fixed number of heads.

Careful analysis of this technique implies that for any integer k, we can simulate t steps of an arbitrary Turing machine with k independent heads in $\Theta(t^2)$ time on a standard Turing machine with only one head. Unfortunately, this quadratic blowup is unavoidable. It is relatively easy to recognize the language of *marked palindromes* $\{w \bullet w^R \mid w \in \{0, 1\}^*\}$ in $O(n)$ time using a Turing machine with two heads, but recognizing this language provably requires $\Omega(n^2)$ time on a standard machine with only one head. On the other hand, with much more sophisticated techniques, it is possible to simulate t steps of a Turing machine with k head, for any fixed integer k, using only $O(t \log t)$ steps on a Turing machine with just *two* heads.

Multiple Tapes

We can also allow machines with multiple independent tapes, each with its own head. To simulate such a machine with a single tape, we simply maintain each tape as an independent track with its own head. Equivalently, we can simulate a machine with k tapes using a single tape with $2k$ tracks, half storing the contents of the k tapes and half storing the positions of the k heads.

►	T	A	P	E	#	O	N	E			···
►									▲		···
►	T	A	P	E	#	T	W	O			···
►			▲								···
►	T	A	P	E	#	T	H	R	E	E	···
►						▲					···

Just as for multiple tracks, for any constant k, we can simulate t steps of an arbitrary Turing machine with k independent tapes in $\Theta(t^2)$ steps on a standard Turing machine with one tape, and this quadratic blowup is unavoidable. Moreover, it is possible to simulate t steps on a k-tape Turing machine using only $O(t \log t)$ steps on a *two*-tape Turing machine using more sophisticated techniques. (This faster simulation is easier to obtain for multiple independent tapes than for multiple heads on the same tape.)

By combining these tricks, we can simulate a Turing machine with any fixed number of tapes, each of which may be infinite in one or both directions, each with any fixed number of heads and any fixed number of tracks, with at most a quadratic blowup in the running time.

6.7 Simulating a Real Computer

6.7.1 Subroutines and Recursion

★★★

Use a second tape/track as a "call stack". Add save and restore actions. In the simplest formulation, subroutines do not have local memory. To call a subroutine, save the current state onto the call stack and jump to the first state of the subroutine. To return, restore (and remove) the return state from the call stack. We can simulate t steps of any recursive Turing machine with $O(t)$ steps on a multitape standard Turing machine, or in $O(t^2)$ steps on a standard Turing machine.

More complex versions of this simulation can adapt to

6.7.2 Random-Access Memory

★★★

Keep [address•data] pairs on a separate "memory" tape. Write address to an "address" tape; read data from or write data to a "data" tape. Add new or changed [address•data] pairs at the end of the memory tape. (Semantics of reading from an address that has never been written to?)

Suppose all memory accesses require at most ℓ address and data bits. Then we can simulate the kth memory access in $O(k\ell)$ steps on a multitape Turing machine or in $O(k^2\ell^2)$ steps on a single-tape machine. Thus, simulating t memory accesses in a random-access machine with ℓ-bit words requires $O(t^2\ell)$ time on a multitape Turing machine, or $O(t^3\ell^2)$ time on a single-tape machine.

6.8 Universal Turing Machines

With all these tools in hand, we can now describe the pinnacle of Turing machine constructions: the ***universal*** Turing machine. For modern computer scientists, it's useful to think of a universal Turing machine as a "Turing machine *interpreter* written in Turing machine". Just as the input to a Python interpreter is a string of Python source code, the input to our universal Turing machine U is a string $\langle M, w\rangle$ that encodes both an arbitrary Turing machine M and a string w in the input alphabet of M. Given these encodings, U simulates the execution of M on input w; in particular,

- U accepts $\langle M, w\rangle$ if and only if M accepts w.
- U rejects $\langle M, w\rangle$ if and only if M rejects w.

The next few pages, I will sketch a universal Turing machine U that uses the input alphabet $\{0, 1, [,], \bullet, |\}$ and a somewhat larger tape alphabet (via marks on additional tracks). However, I will *not* require that the Turing machines that U simulates have similarly small alphabets, so we first need a method to encode *arbitrary* input and tape alphabets.

Encodings

Let $M = (\Gamma, \square, \Sigma, Q, \mathit{start}, \mathit{accept}, \mathit{reject}, \delta)$ be an arbitrary Turing machine, with a single half-infinite tape and a single read-write head. (I will consistently indicate the states and tape symbols of M in *slanted green* to distinguish them from the upright red states and tape symbols of U.)

We encode each symbol $a \in \Gamma$ as a unique string $|a|$ of $\lceil \lg(|\Gamma|)\rceil$ bits. Thus, if $\Gamma = \{0, 1, \$, x, \square\}$, we might use the following encoding:

$$\langle 0\rangle = 001, \qquad \langle 1\rangle = 010, \qquad \langle \$\rangle = 011, \qquad \langle x\rangle = 100, \qquad \langle \square\rangle = 000.$$

The input string w is encoded by its sequence of symbol encodings, with separators • between every pair of symbols and with brackets [and] around the whole string. For example, with this encoding, the input string *001100* would be encoded on the input tape as

$$\langle 001100 \rangle = [001 \bullet 001 \bullet 010 \bullet 010 \bullet 001 \bullet 001]$$

Similarly, we encode each state $q \in Q$ as a distinct string $\langle q \rangle$ of $\lceil \lg|Q| \rceil$ bits. Without loss of generality, we encode the start state with all 1s and the reject state with all 0s. For example, if $Q = \{start, seek1, seek0, reset, verify, accept, reject\}$, we might use the following encoding:

$$\langle start \rangle = 111 \qquad \langle seek1 \rangle = 010 \qquad \langle seek0 \rangle = 011 \qquad \langle reset \rangle = 100$$
$$\langle verify \rangle = 101 \qquad \langle accept \rangle = 110 \qquad \langle reject \rangle = 000$$

We encode the machine M itself as the string $\langle M \rangle = [\langle reject \rangle \bullet \langle \square \rangle]\langle \delta \rangle$, where $\langle \delta \rangle$ is the concatenation of substrings $[\langle p \rangle \bullet \langle a \rangle \,|\, \langle q \rangle \bullet \langle b \rangle \bullet \langle \Delta \rangle]$ encoding each transition $\delta(p, a) = (q, b, \Delta)$ such that $q \neq$ reject. We encode the actions $\Delta = \pm 1$ by defining $\langle -1 \rangle := 0$ and $\langle +1 \rangle := 1$. Conveniently, every transition string has exactly the same length. For example, with the symbol and state encodings described above, the transition $\delta(reset, \$) = (start, \$, +1)$ would be encoded as

$$[100 \bullet 011 | 001 \bullet 011 \bullet 1].$$

Our first example Turing machine for recognizing $\{0^n 1^n 0^n \mid n \geq 0\}$ would be represented by the following string (here broken into multiple lines for readability):

```
[000•000][[001•001|010•011•1][001•100|101•011•1]
          [010•001|010•001•1][010•100|010•100•1]
          [010•010|011•100•1][011•010|011•010•1]
          [011•100|011•100•1][011•001|100•100•1]
          [100•001|100•001•0][100•010|100•010•0]
          [100•100|100•100•0][100•011|001•011•1]
          [101•100|101•011•1][101•000|110•000•0]]
```

Finally, we encode any *configuration* of M on U's work tape by alternating between encodings of states and encodings of tape symbols. Thus, each tape cell is represented by the string $[\langle q \rangle \bullet \langle a \rangle]$ indicating that (1) the cell contains symbol a; (2) if $q \neq reject$, then M's head is located at this cell, and M is in state q; and (3) if $q = reject$, then M's head is located somewhere else. Conveniently, each cell encoding uses exactly the same number of bits. We also surround the entire tape encoding with brackets [and].

For example, with the encodings described above, the initial configuration $(start, \underset{\blacktriangle}{0}01100, 0)$ for our first example Turing machine would be encoded on U's tape as follows.

$$[\underbrace{[111 \bullet 001]}_{\textbf{start } 0}\underbrace{[000 \bullet 001]}_{reject\ 0}\underbrace{[000 \bullet 010]}_{reject\ 1}\underbrace{[000 \bullet 010]}_{reject\ 1}\underbrace{[000 \bullet 001]}_{reject\ 0}\underbrace{[000 \bullet 001]}_{reject\ 0}]$$

Similarly, the intermediate configuration $(reset, \$0x\underset{\blacktriangle}{1}x0, 3)$ would be encoded as follows:

$$[\underbrace{[000 \bullet 011]}_{reject\ \$}\underbrace{[000 \bullet 011]}_{reject\ 0}\underbrace{[000 \bullet 100]}_{reject\ x}\underbrace{[010 \bullet 010]}_{\textbf{reset } 1}\underbrace{[000 \bullet 100]}_{reject\ x}\underbrace{[000 \bullet 001]}_{reject\ 0}]$$

Input and Execution

Without loss of generality, we assume that the input to our universal Turing machine U is given on a separate read-only ***input tape***, as the encoding of an arbitrary Turing machine M followed by an encoding of its input string x. Notice the substrings `[[` and `]]` each appear only only once on the input tape, immediately before and after the encoded transition table, respectively. U also has a read-write ***work tape***, which is initially blank.

We start by initializing the work tape with the encoding $\langle \mathit{start}, x, 0\rangle$ of the initial configuration of M with input x. First, we write `[[`$\langle \mathit{start}\rangle$`•`. Then we copy the encoded input string $\langle x\rangle$ onto the work tape, but we change the punctuation as follows:

- Instead of copying the left bracket `[`, write `[[`$\langle \mathit{start}\rangle$`•`.
- Instead of copying each separator `•`, write `][`$\langle \mathit{reject}\rangle$`•`
- Instead of copying the right bracket `]`, write two right brackets `]]`.

The state encodings $\langle \mathit{start}\rangle$ and $\langle \mathit{reject}\rangle$ can be copied directly from the beginning of $\langle M\rangle$ (replacing `0`s for `1`s for $\langle \mathit{start}\rangle$). Finally, we move the head back to the start of U's tape.

At the start of each step of the simulation, U's head is located at the start of the work tape. We scan through the work tape to the unique encoded cell `[`$\langle p\rangle$`•`$\langle a\rangle$`]` such that $p \neq \mathit{reject}$. Then we scan through the encoded transition function $\langle \delta\rangle$ to find the unique encoded tuple `[`$\langle p\rangle$`•`$\langle a\rangle$`|`$\langle q\rangle$`•`$\langle b\rangle$`•`$\langle \Delta\rangle$`]` whose left half matches our the encoded tape cell. If there is no such tuple, then U immediately halts and rejects. Otherwise, we copy the right half $\langle q\rangle$`•`$\langle b\rangle$ of the tuple to the work tape. Now if $q = \mathit{accept}$, then U immediately halts and accepts. (We don't bother to encode *reject* transformations, so we know that $q \neq \mathit{reject}$.) Otherwise, we transfer the state encoding to either the next or previous encoded cell, as indicated by M's transition function, and then continue with the next step of the simulation.

During the final state-copying phase, we ever read two right brackets `]]`, indicating that we have reached the right end of the tape encoding, we replace the second right bracket with `[`$\langle \mathit{reject}\rangle$`•`$\langle \square\rangle$`]]` (mostly copied from the beginning of the machine encoding $\langle M\rangle$) and then scan back to the left bracket we just wrote. This trick allows our universal machine to *pretend* that its tape contains an infinite sequence of *encoded* blanks `[`$\langle \mathit{reject}\rangle$`•`$\langle \square\rangle$`]` instead of *actual* blanks $\square$.

Example

As an illustrative example, suppose U is simulating our first example Turing machine M on the input string `001100`. The execution of M on input w eventually reaches the configuration (*seek1*, `$$x1x0`, 3). At the start of the corresponding step in U's simulation, U is in the following configuration:

```
[[000•011][000•011][000•100][010•010][000•100][000•001]]
```

First U scans for the first encoded tape cell whose state is not *reject*. That is, U repeatedly compares the first half of each encoded state cell on the work tape with the prefix `[`$\langle \mathit{reject}\rangle$`•` of the machine encoding $\langle M\rangle$ on the input tape. U finds a match in the fourth encoded cell.

```
[[000•011][000•011][000•100][010•010][000•100][000•001]]
```

Next, U scans the machine encoding $\langle M\rangle$ for the substring `[010•010` matching the current encoded cell. U eventually finds a match in the left size of the the encoded transition

`[010•010|011•100•1]`. U copies the state-symbol pair `011•100` from the right half of this encoded transition into the current encoded cell. (The underline indicates which symbols are changed.)

`[[000•011][000•011][000•100][011•100]▲[000•100][000•001]]`

The encoded transition instructs U to move the current state encoding one cell to the right. (The underline indicates which symbols are changed.)

`[[000•011][000•011][000•100][000•100][011•▲100][000•001]]`

Finally, U scans left until it reads two left brackets `[[`; this returns the head to the left end of the work tape to start the next step in the simulation. U's tape now holds the encoding of M's configuration (*seek0*, `$$xxx▲0`, 4), as required.

`▲[[000•011][000•011][000•100][000•100][011•100][000•001]]`

Exercises

In the following problems, a *standard* Turing machine has a single semi-infinite tape, one read-write head, and the input alphabet $\Sigma = \{0, 1\}$. For problems that ask you to *construct* a standard Turing machine, you may assume without loss of generality that the initial tape contains a special symbol ► just to the left of the input string, indicating the left end of the tape; the read-write head starts just to the right of this symbol. For problems that ask you to *simulate* a standard Turing machine, you may assume without loss of generality that the tape alphabet is $\{0, 1, \square\}$.

Turing Machine Programming

1. Describe standard Turing machines that decide each of the following languages:

   (a) Palindromes over the alphabet $\{0, 1\}$

   (b) $\{ww \mid w \in \{0, 1\}^*\}$

   (c) $\left\{0^a 1^b 0^{ab} \mid a, b \in \mathbb{N}\right\}$

2. Let $\langle n \rangle_2$ denote the binary representation of the non-negative integer n. For example, $\langle 17 \rangle_2 = 10001$ and $\langle 42 \rangle_2 = 101010$. Describe standard Turing machines that compute the following functions from $\{0, 1\}^*$ to $\{0, 1\}^*$:

   (a) $w \mapsto www$

   (b) $1^n 0 1^m \mapsto 1^{mn}$

   (c) $1^n \mapsto 1^{2^n}$

   (d) $1^n \mapsto \langle n \rangle_2$

   (e) $0^* \langle n \rangle_2 \mapsto 1^n$

   (f) $\langle n \rangle_2 \mapsto \langle n^2 \rangle_2$

3. Describe standard Turing machines that write each of the following infinite streams of bits onto their tape. Specifically, for each integer n, there must be a finite time after which the first n symbols on the tape always match the first n symbols in the target stream.

   (a) An infinite stream of `1`s

   (b) `0101101110111101111101111110`..., where the nth block of `1`s has length n.

   (c) The stream of bits whose nth bit is `1` if and only if n is prime.

   (d) The ***Thue-Morse sequence*** $T_0 \bullet T_1 \bullet T_2 \bullet T_3 \cdots$, where

$$T_n := \begin{cases} \texttt{0} & \text{if } n = 0 \\ \texttt{1} & \text{if } n = 1 \\ T_{n-1} \bullet \overline{T_{n-1}} & \text{otherwise} \end{cases}$$

   where $\overline{w}$ indicates the binary string obtained from w by flipping every bit. Equivalently, the nth bit of the Thue Morse sequence if `0` if the binary representation of n has an even number of `1`s and `1` otherwise.

`0110100110010110100101100110100110010110011010010110`...

   (e) The ***Fibonacci*** sequence $F_0 \bullet F_1 \bullet F_2 \bullet F_3 \cdots$, where

$$F_n := \begin{cases} \texttt{0} & \text{if } n = 0 \\ \texttt{1} & \text{if } n = 1 \\ F_{n-2} \bullet F_{n-1} & \text{otherwise} \end{cases}$$

`0101101011011010110101101101011011010110101101101010101`...

Simulation by "Weaker" Machines

4. A ***two-stack machine*** is a Turing machine with two tapes with the following restricted behavior. At all times, on each tape, every cell to the right of the head is blank, and every cell at or to the left of the head is non-blank. Thus, a head can only move right by writing a non-blank symbol into a blank cell; symmetrically, a head can only move left by erasing the rightmost non-blank cell. Thus, each tape behaves like a stack. To avoid underflow, there is a special symbol at the start of each tape that cannot be overwritten. Initially, one tape contains the input string, with the head at its *last* symbol, and the other tape is empty (except for the start-of-tape symbol).

   Prove that any standard Turing machine can be simulated by a two-stack machine. That is, given any standard Turing machine M, describe a two-stack machine M' that accepts and rejects exactly the same input strings as M.

5. A ***k-register machine*** is a finite-state automaton with k non-negative integer registers. Formally, a k-register machine consists of a finite set Q of states (which include `start`, `accept`, and `reject`) and a transition function

$$\delta : Q \times \{0,1\}^k \to Q \times \{\texttt{halve}, \texttt{nop}, \texttt{double}, \texttt{double+1}\}^k$$

that takes the internal state and the *signs* of the registers as input, and produces a new internal state and *changes* to the registers as output. The instructions halve, nop, double, and double+1 change any register with value n to $n/2$, n, $2n$, and $2n+1$, respectively.

For example, if $\delta(p,0,1,0,1) = (q, \mathsf{halve}, \mathsf{nop}, \mathsf{halve}, \mathsf{double{+}1})$, then from the configuration $(p,0,2,3,1)$, the machine would transition to $(q,1,2,1,3)$.

Prove that any standard Turing machine (with suitably encoded input and output) can be simulated by a two-register machine. The input to the register machine is encoded in reversed binary in one of the registers, so the parity of the register value is the first input bit; the other register is initially zero.

6. A ***k-counter machine*** (also known as a ***Minksy machine***) is a finite-state automaton with k non-negative integer registers. Formally, a k-counter machine consists of a finite set Q of states (which include start, accept, and reject) and a transition function

$$\delta : Q \times \{0,+\}^k \to Q \times \{\mathsf{inc}, \mathsf{nop}, \mathsf{dec}\}^k$$

that takes the internal state and the *signs* of the registers as input, and produces a new internal state and changes to the registers as output. The instructions inc, nop, and dec change any register with value n to $n+1$, n, and $n-1$, respectively. The transition function must forbid decrementing a register whose value is already zero.

For example, if $\delta(p,0,+,+,+) = (q, \mathsf{inc}, \mathsf{dec}, \mathsf{nop}, \mathsf{dec})$, then from the configuration $(p,0,2,3,1)$, the machine would transition to $(q,1,1,3,0)$.

(a) Prove that any standard Turing machine (with suitably encoded input and output) can be simulated by a three-counter machine. *[Hint: Simulate a two-**register** machine, using the third counter for scratch work.]*

(b) Prove that any three-counter machine (with suitably encoded input and output) can be simulated by a two-counter machine. *[Hint: Store all three counters in a single integer of the form $2^a 3^b 5^c$, and use the other counter for scratch work.]*

⋆(c) Prove that a three-counter machine can compute *a suitable encoding of* any computable function. Specifically, for any computable function $f : \mathbb{N} \to \mathbb{N}$, prove there is a three-counter machine M that transforms any input $(n,0,0)$ into $(f(n),0,0)$. *[Hint: First transform $(n,0,0)$ to $(2^n,0,0)$ using all three counters; then run a two- (or three-)counter TM simulation to obtain $(2^{f(n)},0,0)$; and finally transform $(2^{f(n)},0,0)$ to $(f(n),0,0)$ using all three counters.]*

★(d) Prove that not two-counter machine can transform $(n,0)$ to $(2^n,0)$. This impossiblity result was independently proved by Bārzdiņš in 1963, Yao in 1971, and Schroeppel in 1972. [5]

7. A ***hole-punch*** Turing machine is a standard Turing machine with two restrictions. First, the tape alphabet has only two symbols □ and ■, and thus the input alphabet is the singleton

[5] Ja. M. Barzdin′ [Jānis Bārzdiņš]. Ob odnom klasse mašin T′ûringa (mašiny Minskogo) [On a class of Turing machines (Minsky machines)]. *Algebra i Logika* 1(6):42–51, 1963. In Russian. Sorry.

Oscar H. Ibarra, Nicholas Q. Trân. A note on simple programs with two variables. *Theoretical Computer Science* 112(2): 391–397, 1993.

Rich Schroeppel. A two counter machine cannot calculate 2^N. Artificial Intelligence Memo 257, MIT AI Lab, May 1972. [Schroeppel claims that the same result was independently proved by Frances Yao in 1971.

set {■}. Second, the machine can never write a blank (□) over a non-blank (■); intuitively, the machine can punch new holes (■s) into the tape, but it cannot erase holes.

Prove that any standard Turing machine (with a unary input alphabet) can be simulated by a hole-punch Turing machine.

8. A ***tag***-Turing machine has two heads: one can only read, the other can only write. Initially, the read head is located at the left end of the tape, and the write head is located at the first blank after the input string. At each transition, the read head can either move one cell to the right or stay put, but the write head *must* write a symbol to its current cell and move one cell to the right. Neither head can ever move to the left.

   Prove that any standard Turing machine can be simulated by a tag-Turing machine. That is, given any standard Turing machine M, describe a tag-Turing machine M' that accepts and rejects exactly the same input strings as M.

9. ⋆(a) Prove that any standard Turing machine can be simulated by a standard Turing machine with only three states. *[Hint: Keep an encoding of the state of the simulat**ed** machine on the tape of the simulat**ing** machine.]*

   ★(b) Prove that any standard Turing machine can be simulated by a standard Turing machine with only *two* states.

Simulating "Stronger" Machines

10. A ***two-dimensional*** Turing machine uses an infinite two-dimensional grid of cells as the tape; at each transition, the head can move from its current cell to any of its four neighbors on the grid. The transition function of such a machine has the form $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\uparrow, \leftarrow, \downarrow, \rightarrow\}$, where the arrows indicate which direction the head should move.

    (a) Prove that any two-dimensional Turing machine can be simulated by a standard Turing machine.

    (b) Suppose further that we endow our two-dimensional Turing machine with the following additional actions, in addition to moving the head:

    - Insert row: Move all symbols on or above the row containing the head up one row, leaving the head's row blank.
    - Insert column: Move all symbols on or to the right of the column containing the head one column to the right, leaving the head's column blank.
    - Delete row: Move all symbols above the row containing the head down one row, deleting the head's row of symbols.
    - Delete column: Move all symbols the right of the column containing the head one column to the right, deleting the head's column of symbols.

    Show that any two-dimensional Turing machine that can add an delete rows can be simulated by a standard Turing machine.

11. A ***binary-tree*** Turing machine uses an infinite binary tree as its tape; that is, *every* cell in the tape has a left child and a right child. At each step, the head moves from its current

cell to its Parent, its Left child, or to its Right child. Thus, the transition function of such a machine has the form $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathsf{P}, \mathsf{L}, \mathsf{R}\}$. The input string is initially given along the left spine of the tape.

Show that any binary-tree Turing machine can be simulated by a standard Turing machine.

12. A ***stack-tape*** Turing machine uses an semi-infinite tape, where every cell is actually the top of an independent stack. The behavior of the machine at each iteration is governed by its internal state and the symbol *at the top* of the current cell's stack. At each transition, the head can optionally push a new symbol onto the stack, or pop the top symbol off the stack. (If a stack is empty, its "top symbol" is a blank and popping has no effect.)

    Show that any stack-tape Turing machine can be simulated by a standard Turing machine. (Compare with Problem 4!)

13. A ***tape-stack*** Turing machine has two actions that modify its work tape, in addition to simply writing individual cells: it can ***save*** the entire tape by pushing in onto a stack, and it can ***restore*** the entire tape by popping it off the stack. Restoring a tape returns the content of every cell to its content when the tape was saved. Saving and restoring the tape do not change the machine's state or the position of its head. If the machine attempts to "restore" the tape when the stack is empty, the machine crashes.

    Show that any tape-stack Turing machine can be simulated by a standard Turing machine.

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

I said in my haste, All men are liars.

— Psalms 116:11 (King James Version)

Some problems are so complex that you have to be highly intelligent and well informed just to be undecided about them.

— Laurence Johnston Peter, *Peter's Almanac* (September 24, 1982)

"Proving or disproving a formula—once you've encrypted the formula into numbers, that is—is just a calculation on that number. So it means that the answer to the question is, no! Some formulas cannot be proved or disproved by any mechanical process! So I guess there's some point in being human after all!"

Alan looked pleased until Lawrence said this last thing, and then his face collapsed. "Now there you go making unwarranted assumptions."

— Neal Stephenson, *Cryptonomicon* (1999)

No matter how P might perform, Q will scoop it:
Q uses P's output to make P look stupid.
Whatever P says, it cannot predict Q:
P is right when it's wrong, and is false when it's true!

— Geoffrey S. Pullum, "Scooping the Loop Sniffer" (2000)

★★★ Rewrite in the language of algorithms instead of the language of Turing machines, using "source code" instead of "encoding" everywhere. Formulation in terms of TMs makes almost everything much more complicated than it needs to be. (The dovetail/product construction in the proof of Lemma 4 may be an exception.)

7 Undecidability

Perhaps the single most important result in Turing's remarkable 1936 paper that introduces Turing machines is his observation that there are problems that cannot be solved by *any* algorithm. Turing's canonical example of an undecidable problem was the ***halting problem***, which asks whether a given Turing machine halts when given a particular input string. Among other consequences, Turing's undecidability result provided an elegant negative solution to Hilbert's *Entscheidungsproblem*, which asked for an algorithm to decide whether a given statement of first-order logic is true—no such algorithm exists.

7.1 Acceptable versus Decidable

Recall that there are three possible outcomes for a Turing machine M running on any particular input string w: acceptance, rejection, and divergence. Every Turing machine M immediately defines four different languages (over the input alphabet Σ of M):

- The *accepting* language $\textsc{Accept}(M) := \{w \in \Sigma^* \mid M \text{ accepts } w\}$
- The *rejecting* language $\textsc{Reject}(M) := \{w \in \Sigma^* \mid M \text{ rejects } w\}$
- The *halting* language $\textsc{Halt}(M) := \textsc{Accept}(M) \cup \textsc{Reject}(M)$
- The *diverging* language $\textsc{Diverge}(M) := \Sigma^* \setminus \textsc{Halt}(M)$

For any language L, the sentence "***M accepts L***" means $\textsc{Accept}(M) = L$, and the sentence "***M decides L***" means $\textsc{Accept}(M) = L$ and $\textsc{Diverge}(M) = \varnothing$.

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

Now let L be an arbitrary language. We say that L is ***acceptable*** (or *semi-computable*, or *semi-decidable*, or *recognizable*, or *listable*, or *recursively enumerable*) if some Turing machine accepts L, and ***unacceptable*** otherwise. Similarly, L is ***decidable*** (or *computable*, or *recursive*) if some Turing machine decides L, and ***undecidable*** otherwise.

7.2 Lo, I Have Become Death, Stealer of Pie

There is a subtlety in the definitions of "acceptable" and "decidable" that many beginners miss: A language can be decidable even if we can't exhibit a specific Turing machine decides it. As a canonical example, consider the language $\Pi = \{w \mid 1^{|w|} \text{ appears in the binary expansion of } \pi\}$. Despite appearances, this language *is* decidable! There are only two cases to consider:

- Suppose there is an integer N such that the binary expansion of π contains the substring 1^N but does not contain the substring 1^{N+1}. Let M_N be the Turing machine with $N+3$ states $\{0, 1, \ldots, N, \text{accept}, \text{reject}\}$, start state 0, and the following transition function:

$$\delta(q, a) = \begin{cases} \text{accept} & \text{if } a = \square \\ \text{reject} & \text{if } a \neq \square \text{ and } q = n \\ (q+1, a, +1) & \text{otherwise} \end{cases}$$

  This machine correctly decides Π.

- Suppose the binary expansion of π contains arbitrarily long substrings of 1s. Then any Turing machine that accepts all inputs correctly decides Π.

We have no idea which of these machines correctly decides Π, but one of them does, and that's enough!

7.3 Useful Properties

This subsection lists several simple but useful properties of (un)decidable and (un)acceptable languages. Almost all of these properties follow from routine definition-chasing; readers are strongly encouraged to try to prove each lemma themselves before reading ahead.

One might reasonably ask why we don't also define "rejectable" and "haltable" languages. The following lemma, whose proof is an easy exercise (hint, hint), implies that these sets are both identical to the acceptable languages.

Lemma 1. *Let M be an arbitrary Turing machine.*

(a) There is a Turing machine M^R such that $\textsc{Accept}(M^R) = \textsc{Reject}(M)$ and $\textsc{Reject}(M^R) = \textsc{Accept}(M)$.

(b) There is a Turing machine M^A such that $\textsc{Accept}(M^A) = \textsc{Accept}(M)$ and $\textsc{Reject}(M^A) = \varnothing$.

(c) There is a Turing machine M^H such that $\textsc{Accept}(M^H) = \textsc{Halt}(M)$ and $\textsc{Reject}(M^H) = \varnothing$.

The decidable languages have several fairly obvious useful closure properties.

Lemma 2. *If L and L' are decidable, then $L \cup L'$, $L \cap L'$, $L \setminus L'$, and $L' \setminus L$ are also decidable.*

Proof: Let M and M' be Turing machines that decide L and L', respectively. We can build a Turing machine $M_\cup$ that decides $L \cup L'$ as follows. First, $M_\cup$ copies its input string w onto a second tape. Then $M_\cup$ runs M on input w (on the first tape), and then runs M' on input w (on the second tape). If either M or M' accepts, then $M_\cup$ accepts; if both M and M' reject, then $M_\cup$ rejects.

The other three languages are similar. $\square$

Corollary 3. *The following hold for all languages L and L'.*
(a) If $L \cap L'$ is undecidable and L' is decidable, then L is undecidable.
(b) If $L \cup L'$ is undecidable and L' is decidable, then L is undecidable.
(c) If $L \setminus L'$ is undecidable and L' is decidable, then L is undecidable.
(d) If $L' \setminus L$ is undecidable and L' is decidable, then L is undecidable.

Unfortunately, acceptable languages are not quite as well-behaved as decidable languages, thanks to the subtle distinction between *rejecting* a string and *not accepting* a string.

Lemma 4. *For all acceptable languages L and L', the languages $L \cup L'$ and $L \cap L'$ are also acceptable.*

Proof: Let M and M' be Turing machines that decide L and L', respectively. We can build a Turing machine $M_\cap$ that decides $L \cap L'$ as follows. First, $M_\cap$ copies its input string w onto a second tape. Then $M_\cap$ runs M on input w using the first tape, and then runs M' on input w using the second tape. If both M and M' accept, then $M_\cap$ accepts; if either M or M' reject, then $M_\cap$ rejects; if either M or M' diverge, then $M_\cap$ diverges (automatically).

The construction for $L \cup L'$ is more subtle; instead of running M and M' in series, we must run them in parallel. Like $M_\cap$, the new machine $M_\cup$ starts by copying its input string w onto a second tape. But then $M_\cup$ runs M and M' simultaneously; with each step of $M_\cup$ simulating both one step of M on the first tape and one step of M' on the second. Ignoring the states and transitions needed for initialization, the state set of $M_\cup$ is the product of the state sets of M and M', and the transition function is

$$\delta_\cup(q, a, q', a') = \begin{cases} \mathsf{accept}_\cup & \text{if } q = \mathsf{accept} \text{ or } q' = \mathsf{accept}' \\ \mathsf{reject}_\cup & \text{if } q = \mathsf{reject} \text{ and } q' = \mathsf{reject}' \\ (\delta(q, a), \delta'(q', a')) & \text{otherwise} \end{cases}$$

Thus, $M_\cup$ accepts as soon as either M or M' accepts, and rejects only after both M or M' reject. □

Lemma 5. *An acceptable language L is decidable if and only if $\Sigma^* \setminus L$ is also acceptable.*

Proof: Let M and $\overline{M}$ be Turing machines that accept L and $\Sigma^* \setminus L$, respectively. Following the previous proof, we construct a new Turing machine M^* that copies its input onto a second tape, and then simulates M and M' in parallel on the two tapes. If M accepts, then M^* accepts; if $\overline{M}$ accepts, then M^* rejects. Since every string is accepted by either M or $\overline{M}$, we conclude that M^* decides L.

The other direction follows immediately from Lemma 1. □

7.4 Code is Data; Data is Code

Perhaps the single most important observation in developing these undecidability results—and one of the most important observations in computer science more broadly—is that Turing machines can be encoded as strings. At one level, this observation is completely trivial: Any written description of a Turing machine is a string, and modern code *is* just a sequence of bytes, stored in a file like any other data. But this apparently trivial observation is actually incredibly powerful.

Most natural encodings of Turing machines have three important properties.

- **Unique:** Different Turing machines are encoded as different strings.
- **Modifiable:** We can algorithmically modify any Turing machine M, given the encoding of M as input. For example, there are algorithms to swap the accept and reject states of any Turing machine, or to add new states and transitions representing pre- and post-processing phases, or to build a new machine that calls M as a subroutine, or to build a new machine that runs several copies of M in parallel.
- **Executable:** There is a fixed ***universal*** Turing machine U that can simulate the behavior of an arbitrary Turing machine M, given the encodings of M and w as input. For example, if we decided to encode Turing machines as Python programs, then U would be a Python interpreter.

The precise details of the encoding are unimportant, but for the sake of concreteness, let me describe a natural encoding of Turing machines as strings over the six-character alphabet $\{0, 1, \{, \bullet, \}\}$. Let $M = (\Gamma, \square, \Sigma, Q, \mathit{start}, \mathit{accept}, \mathit{reject}, \delta)$ be an arbitrary Turing machine, with a single half-infinite tape and a single read-write head. (I will consistently indicate the states and tape symbols of M in *slanted green* to distinguish them from the upright red symbols in the encoding alphabet.)

- We encode each symbol $a \in \Gamma$ as a unique string $|a|$ of $\lceil \lg(|\Gamma|) \rceil$ bits. For example, if $\Gamma = \{0, 1, \$, x, \square\}$, we might use the following encoding:

$$\langle 0 \rangle = 001, \qquad \langle 1 \rangle = 010, \qquad \langle \$ \rangle = 011, \qquad \langle x \rangle = 100, \qquad \langle \square \rangle = 000.$$

- Similarly, we encode each state $q \in Q$ as a distinct string $\langle q \rangle$ of $\lceil \lg|Q| \rceil$ bits. Without loss of generality, we encode the start state with all 1s and the reject state with all 0s. For example, if $Q = \{\mathit{start}, \mathit{seek1}, \mathit{seek0}, \mathit{reset}, \mathit{verify}, \mathit{accept}, \mathit{reject}\}$, we might use the following encoding:

$$\langle \mathit{start} \rangle = 111 \qquad \langle \mathit{seek1} \rangle = 010 \qquad \langle \mathit{seek0} \rangle = 011 \qquad \langle \mathit{reset} \rangle = 100$$
$$\langle \mathit{verify} \rangle = 101 \qquad \langle \mathit{accept} \rangle = 110 \qquad \langle \mathit{reject} \rangle = 000$$

- Finally, we encode the machine M itself as the string $\langle M \rangle = \{\langle \mathit{reject} \rangle \bullet \langle \square \rangle\}\langle \delta \rangle$, where $\langle \delta \rangle$ is the concatenation of substrings $\{\langle p \rangle \bullet \langle a \rangle \bullet \langle q \rangle \bullet \langle b \rangle \bullet \langle \Delta \rangle\}$ encoding each transition $\delta(p, a) = (q, b, \Delta)$ such that $q \neq \mathrm{reject}$. We encode the actions $\Delta = \pm 1$ by defining $\langle -1 \rangle := 0$ and $\langle +1 \rangle := 1$. Conveniently, every transition string has exactly the same length. For example, with the symbol and state encodings described above, the transition $\delta(\mathit{reset}, \$) = (\mathit{start}, \$, +1)$ would be encoded as the string

$$\{100 \bullet 011 \bullet 001 \bullet 011 \bullet 1\}.$$

  Our first example Turing machine for recognizing $\{0^n 1^n 0^n \mid n \geq 0\}$ would be represented by the following string (broken into multiple lines for readability):

```
{000•000}{{001•001•010•011•1}{001•100•101•011•1}{010•001•010•001•1}
          {010•100•010•100•1}{010•010•011•100•1}{011•010•011•010•1}
          {011•100•011•100•1}{011•001•100•100•1}{100•001•100•001•0}
          {100•010•100•010•0}{100•100•100•100•0}{100•011•001•011•1}
          {101•100•101•011•1}{101•000•110•000•0}}
```

Building a universal Turing machine U that uses this encoding is more a matter of careful bookkeeping than real insight. We can encode any *configuration* of M on U's work tape by encoding each cell of M's tape as a string $\{\langle q\rangle \bullet \langle a\rangle\}$ indicating that (1) the cell contains symbol a; (2) if $q \neq \mathit{reject}$, then M's head is located at this cell, and M is in state q; and (3) if $q = \mathit{reject}$, then M's head is located somewhere else. We also surround the entire tape encoding with brackets $\{$ and $\}$. For example, the initial configuration $(\mathit{start}, \underset{\blacktriangle}{0}0110, 0)$ for our example Turing machine would be encoded as follows.

$$\langle \mathit{start}, \underset{\blacktriangle}{0}0110, 0\rangle = \{\underbrace{\{111\bullet 001\}}_{\textbf{start } 0}\underbrace{\{000\bullet 001\}}_{\text{reject } 0}\underbrace{\{000\bullet 010\}}_{\text{reject } 1}\underbrace{\{000\bullet 010\}}_{\text{reject } 1}\underbrace{\{000\bullet 001\}}_{\text{reject } 0}\}$$

Similarly, the intermediate configuration $(\mathit{reset}, \$0x\underset{\blacktriangle}{1}x, 3)$ would be encoded as follows:

$$\langle \mathit{reset}, \$\$x\underset{\blacktriangle}{1}x, 3\rangle = \{\underbrace{\{000\bullet 011\}}_{\text{reject } \$}\underbrace{\{000\bullet 011\}}_{\text{reject } 0}\underbrace{\{000\bullet 100\}}_{\text{reject } x}\underbrace{\{010\bullet 010\}}_{\textbf{reset } 1}\underbrace{\{000\bullet 100\}}_{\text{reject } x}\}$$

To simulate one step of M's execution, we (1) find the location of the head (or reject if the head has vanished), (2) look up the transition for the state-symbol pair at the head, and (3) update the current cell and one of its neighbors to reflect the transition. The remaining grungy details are left as an exercise.

7.5 Self-Haters Gonna Self-Hate

A Turing machine encoding $\langle M\rangle$ is just a string, and any string (over the correct alphabet) can be used as the input to a Turing machine. Thus, a suitable encoding of *any* Turing machine can be used as the input to *any* Turing machine. In particular:

The encoding $\langle M\rangle$ of Turing machine M can be used as input to *the same* Turing machine M.

Turing used this observation about self-reference to derive his first undecidable language as follows. Let's say that a Turing machine M is ***self-rejecting*** if it rejects its own encoding $\langle M\rangle$. Let SelfReject be the set of all encodings of self-rejecting Turing machines:

$$\textsc{SelfReject} := \{\langle M\rangle \mid M \text{ rejects } \langle M\rangle\}$$

Theorem 6. *SelfReject is undecidable.*

Proof: Suppose to the contrary that there is a Turing machine SR that decides SelfReject. Then by definition, $\textsc{Accept}(SR) = \textsc{SelfReject}$ and $\textsc{Diverge}(SR) = \varnothing$. More explicitly, for ***any*** Turing machine M,

- SR accepts $\langle M\rangle \iff M$ rejects $\langle M\rangle$, and
- SR rejects $\langle M\rangle \iff M$ does not reject $\langle M\rangle$.

In particular, these equivalences must hold when M is the machine SR. Thus,

- SR accepts $\langle SR\rangle \iff SR$ rejects $\langle SR\rangle$, and
- SR rejects $\langle SR\rangle \iff SR$ does not reject $\langle SR\rangle$.

In short, SR accepts $\langle SR\rangle$ if and only if SR rejects $\langle SR\rangle$, which is impossible! The only logical conclusion is that the Turing machine SR does not exist. □

7.6 Aside: Uncountable Barbers

Turing's proof by contradiction is an avatar of the famous ***diagonalization argument*** that uncountable sets exist, published by Georg Cantor in 1891. Indeed, SelfReject is sometimes called "the diagonal language". Recall that a function $f : A \to B$ is a ***surjection***[1] if $f(A) = \{f(a) \mid a \in A\} = B$.

Cantor's Theorem. *Let $f : X \to 2^X$ be an **arbitrary** function from an **arbitrary** set X to its power set. This function f is not a surjection.*

Proof: Fix an arbitrary function $f : X \to 2^X$. Call an element $x \in X$ ***happy*** if $x \in f(x)$ and ***sad*** if $x \notin f(x)$. Let Y be the set of all sad elements of X; that is, for *every* element $x \in X$, we have

$$x \in Y \iff x \notin f(x).$$

For the sake of argument, suppose f is a surjection. Then (by definition of surjection) there must be an element $y \in X$ such that $f(y) = Y$. Then for *every* element $x \in X$, we have

$$x \in f(y) \iff x \notin f(x).$$

In particular, the previous equivalence must hold when $x = y$:

$$y \in f(y) \iff y \notin f(y).$$

We have a contradiction! We conclude that f is not a surjection after all. □

Now let $X = \Sigma^*$, and define the function $f : X \to 2^X$ as follows:

$$f(w) := \begin{cases} \textsc{Accept}(M) & \text{if } w = \langle M \rangle \text{ for some Turing machine } M \\ \varnothing & \text{if } w \text{ is not the encoding of a Turing machine} \end{cases}$$

Cantor's theorem immediately implies that not all languages are acceptable.

Alternatively, let X be the set of all Turing machines that halt on all inputs. For any Turing machine $M \in X$, let $f(M)$ be the set of all Turing machines $N \in X$ such that M accepts the encoding $\langle N \rangle$. Then a Turing machine M is *sad* if it rejects its own encoding $\langle M \rangle$; thus, Y is essentially the set SelfReject. Cantor's argument now immediately implies that no Turing machine decides the language SelfReject.

The core of Cantor's diagonalization argument also appears in the "barber paradox" popularized by Bertrand Russell in the 1910s. In a certain small town, every resident has a haircut on Haircut Day. Some residents cut their own hair; others have their hair cut by another resident of the same town. To obtain an official barber's license, a resident must cut the hair of all residents who don't cut their own hair, and no one else. Given these assumptions, we can immediately conclude that there are no licensed barbers. After all, who would cut the barber's hair?

To map Russell's barber paradox back to Cantor's theorem, let X be the set of residents, and let $f(x)$ be the set of residents who have their hair cut by x; then a resident is *sad* if they do not cut their own hair. To prove that SelfReject is undecidable, replace "resident" with "a Turing machine that halts on all inputs", and replace "A cuts B's hair" with "A accepts $\langle B \rangle$".

[1] more commonly, flouting all reasonable standards of grammatical English, "an onto function"

7.7 Just Don't Know What to Do with Myself

Similar diagonal arguments imply that three other languages are also undecidable:

$$\begin{aligned}
\textsc{SelfAccept} &:= \big\{\langle M\rangle \;\big|\; M \text{ accepts } \langle M\rangle\big\} \\
\textsc{SelfHalt} &:= \big\{\langle M\rangle \;\big|\; M \text{ halts on } \langle M\rangle\big\} \\
\textsc{SelfDiverge} &:= \big\{\langle M\rangle \;\big|\; M \text{ diverges on } \langle M\rangle\big\}
\end{aligned}$$

The proofs for these three languages are not quite as direct as the proof for SelfReject; each fictional deciding machine requires a small modification to create the contradiction.

Theorem 7. *SelfAccept is undecidable.*

Proof: For the sake of argument, suppose there is a Turing machine SA such that $\textsc{Accept}(SA) = \textsc{SelfAccept}$ and $\textsc{Diverge}(M) = \varnothing$. Let SA^R be the Turing machine obtained from SA by swapping its accept and reject states (as in the proof of Lemma 1). Then $\textsc{Reject}(SA^R) = \textsc{SelfAccept}$ and $\textsc{Diverge}(SA^R) = \varnothing$. It follows that SA^R rejects $\langle SA^R\rangle$ if and only if SA^R accepts $\langle SA^R\rangle$, which is impossible. □

Theorem 8. *SelfHalt is undecidable.*

Proof: Suppose to the contrary that there is a Turing machine SH such that $\textsc{Accept}(SH) = \textsc{SelfHalt}$ and $\textsc{Diverge}(SH) = \varnothing$. Let SH^X be the Turing machine obtained from SH by redirecting every transition to accept to a new hanging state hang, and then redirecting every transition to reject to accept. Then $\textsc{Accept}(SH^X) = \Sigma^* \setminus \textsc{SelfHalt}$ and $\textsc{Reject}(SH^X) = \varnothing$. It follows that SH^X accepts $\langle SH^X\rangle$ if and only if SH^X does not halt on $\langle SH^X\rangle$, and we have a contradiction. □

Theorem 9. *SelfDiverge is* ***unacceptable and therefore*** *undecidable.*

Proof: Suppose to the contrary that there is a Turing machine SD such that $\textsc{Accept}(M) = \textsc{SelfDiverge}$. Let SD^A be the Turing machine obtained from M by redirecting every transition to reject to a new hanging state hang such that $\delta(\text{hang}, a) = (\text{hang}, a, +1)$ for every symbol a. Then $\textsc{Accept}(SD^A) = \textsc{SelfDiverge}$ and $\textsc{Reject}(SD^A) = \varnothing$. It follows that SD^A accepts $\langle SD^A\rangle$ if and only if SD^A does not halt on $\langle SD^A\rangle$, which is impossible. □

*7.8 Nevertheless, Acceptable

Our undecidability argument for SelfDiverge actually implies the stronger result that SelfDiverge is unacceptable; we never assumed that the hypothetical accepting machine SD halts on all inputs. However, we can use or modify our universal Turing machine U to *accept* the other three self-referential languages.

Theorem 10. *SelfAccept is acceptable.*

Proof: We describe a Turing machine SA that accepts the language SelfAccept. Given any string w as input, SA first verifies that w is the encoding of a Turing machine. If w is not the encoding of a Turing machine, then SA diverges. Otherwise, $w = \langle M\rangle$ for some Turing machine M; in this case, SA writes the string $ww = \langle M\rangle\langle M\rangle$ onto its tape and passes control to the universal Turing machine U. U then simulates M (the machine encoded by the first half of

its input) on the string $\langle M \rangle$ (the second half of its input).[2] In particular, U accepts $\langle M, M \rangle$ if and only if M accepts $\langle M \rangle$. We conclude that SR accepts $\langle M \rangle$ if and only if M accepts $\langle M \rangle$. □

Theorem 11. *SelfReject is acceptable.*

Proof: Let U^R be the Turing machine obtained from our universal machine U by swapping the accept and reject states. We describe a Turing machine SR that accepts the language SelfReject as follows. SR first verifies that its input string w is the encoding of a Turing machine and diverges if not. Otherwise, SR writes the string $ww = \langle M, M \rangle$ onto its tape and passes control to the reversed universal Turing machine U^R. Then U^R accepts $\langle M, M \rangle$ if and only if M rejects $\langle M \rangle$. We conclude that SR accepts $\langle M \rangle$ if and only if M rejects $\langle M \rangle$. □

Finally, because SelfHalt is the union of two acceptable languages, SelfHalt is also acceptable.

7.9 The Halting Problem via Reduction

Now consider the following related languages:[3]

$$\begin{aligned}
\textsc{Accept} &:= \left\{ \langle M, w \rangle \;\middle|\; M \text{ accepts } w \right\} \\
\textsc{Reject} &:= \left\{ \langle M, w \rangle \;\middle|\; M \text{ rejects } w \right\} \\
\textsc{Halt} &:= \left\{ \langle M, w \rangle \;\middle|\; M \text{ halts on } w \right\} \\
\textsc{Diverge} &:= \left\{ \langle M, w \rangle \;\middle|\; M \text{ diverges on } w \right\}
\end{aligned}$$

Deciding the language Halt is usually called the ***halting problem***: Given a program M and an input w to that program, does the program halt? This problem may seem trivial; why not just run the program and see? More formally, why not just pass the input string $\langle M, x \rangle$ to our universal Turing machine U? That strategy works perfectly if we just want to *accept* Halt, but we actually want to *decide* Halt; if M is not going to halt on w, we still want an answer in a finite amount of time. Sadly, we can't always get what we want.

Theorem 12. *Halt is undecidable.*

Proof: Suppose to the contrary that there is a Turing machine H that decides Halt. Then we can use H to build another Turing machine SH that decides the language SelfHalt. Given any string w, the machine SH first verifies that $w = \langle M \rangle$ for some Turing machine M (rejecting if not), then writes the string $ww = \langle M, M \rangle$ onto the tape, and finally passes control to H. But SelfHalt is undecidable, so no such machine SH exists. We conclude that H does not exist either. □

Nearly identical arguments imply that the languages Accept, Reject, and Diverge are undecidable.

[2] To simplify the presentation, I am implicitly assuming here that $\langle M \rangle = \langle \langle M \rangle \rangle$. Without this assumption, we need a Turing machine that transforms an arbitrary string $w \in \Sigma_M^*$ into its encoding $\langle w \rangle \in \Sigma_U^*$; building such a Turing machine is straightforward.

[3] Many sources including Sipser and Wikipedia uses the shorter name A_{TM} instead of Accept, but uses $HALT_{TM}$ instead of Halt. I have no idea why Sipser thought four-letter names are okay, but six-letter names are not. The subscript TM is just a reminder that these are languages of *Turing machine* encodings, as opposed to encodings of DFAs or some other machine model.

Here we have our first example of an undecidability proof by ***reduction***. Specifically, we ***reduced*** the language SelfHalt to the language Halt. More generally, to reduce one language X to another language Y, we assume (for the sake of argument) that there is a program P_Y that decides Y, and we write another program that decides X, using P_Y as a black-box subroutine. If later we discover that Y is decidable, we can immediately conclude that X is decidable. Equivalently, if we later discover that X is undecidable, we can immediately conclude that Y is undecidable.

To prove that a language L is undecidable, reduce a known undecidable language to L.

Perhaps the most confusing aspect of reduction arguments is that the *languages* we want to prove undecidable nearly (but not quite) always involve encodings of Turing machines, while at the same time, the *programs* that we build to prove them undecidable are also Turing machines. Our proof that Halt is undecidable involved three different machines:

- The hypothetical Turing machine H that decides Halt.
- The new Turing machine SH that decides SelfHalt, using H as a subroutine.
- The Turing machine M whose encoding is the input to H.

It is *incredibly* easy to get confused about which machines are playing each in the proof. Therefore, it is absolutely vital that we give each machine in a reduction proof a unique and mnemonic name, and then ***always*** refer to each machine ***by name***. Never write, say, or even *think* "the Turing machine" or "the state" or "the tape" or "the input" or (gods forbid) "it". You also may find it useful to think of the working ***programs*** we are trying to construct (H and SH in this proof) as being written in a different language than the arbitrary ***source code*** that we want those programs to analyze ($\langle M \rangle$ in this proof).

7.10 One Million Years Dungeon!

As a more complex set of examples, consider the following languages:

$$\begin{aligned}
\textsc{NeverAccept} &:= \big\{\langle M \rangle \mid \textsc{Accept}(M) = \varnothing\big\} \\
\textsc{NeverReject} &:= \big\{\langle M \rangle \mid \textsc{Reject}(M) = \varnothing\big\} \\
\textsc{NeverHalt} &:= \big\{\langle M \rangle \mid \textsc{Halt}(M) = \varnothing\big\} \\
\textsc{NeverDiverge} &:= \big\{\langle M \rangle \mid \textsc{Diverge}(M) = \varnothing\big\}
\end{aligned}$$

Theorem 13. *NeverAccept is undecidable.*

Proof: Suppose to the contrary that there is a Turing machine NA that decides NeverAccept. Then by swapping the accept and reject states, we obtain a Turing machine NA^R that decides the complementary language $\Sigma^* \setminus \textsc{NeverAccept}$.

To reach a contradiction, we construct a Turing machine A that decides Accept as follows. Given the encoding $\langle M, w \rangle$ of an arbitrary machine M and an arbitrary string w as input, A writes the encoding $\langle M_w \rangle$ of a new Turing machine M_w that ignores its input, writes w onto the tape, and then passes control to M. Finally, A passes the new encoding $\langle M_w \rangle$ as input to NA^R. The following cartoon tries to illustrate the overall construction.

Before going any further, it may be helpful to list the various Turing machines that appear in this construction.

A reduction from from ACCEPT to NEVERACCEPT, which proves NEVERACCEPT undecidable.

- The hypothetical Turing machine NA that decides NEVERACCEPT.
- The Turing machine NA^R that decides $\Sigma^* \setminus$ NEVERACCEPT, which we constructed by modifying NA.
- The Turing machine A that we are building, which decides ACCEPT using NA^R as a black-box subroutine.
- The Turing machine M, whose encoding is part of the input to A.
- The Turing machine M_w whose encoding A constructs from $\langle M, w\rangle$ and then passes to NA^R as input.

Now let M be an arbitrary Turing machine and w be an arbitrary string, and suppose we run our new Turing machine A on the encoding $\langle M, w\rangle$. To complete the proof, we need to consider two cases: Either M accepts w or M does not accept w.

- First, suppose M accepts w.
  - Then for all strings x, the machine M_w accepts x.
  - So ACCEPT$(M_w) = \Sigma^*$, by the definition of ACCEPT(M_w).
  - So $\langle M_w\rangle \notin$ NEVERACCEPT, by definition of NEVERACCEPT.
  - So NA rejects $\langle M_w\rangle$, because NA decides NEVERACCEPT.
  - So NA^R accepts $\langle M_w\rangle$, buy construction of NA^R.
  - We conclude that A accepts $\langle M, w\rangle$, by construction of A.
- On the other hand, suppose M does not accept w, either rejecting or diverging instead.
  - Then for all strings x, the machine M_w does not accept x.
  - So ACCEPT$(M_w) = \varnothing$, by the definition of ACCEPT(M_w).
  - So $\langle M_w\rangle \in$ NEVERACCEPT, by definition of NEVERACCEPT.
  - So NA accepts $\langle M_w\rangle$, because NA decides NEVERACCEPT.
  - So NA^R rejects $\langle M_w\rangle$, buy construction of NA^R.
  - We conclude that A rejects $\langle M, w\rangle$, by construction of A.

In short, A decides the language ACCEPT, which is impossible. We conclude that NA does not exist. □

Again, similar arguments imply that the languages NEVERREJECT, NEVERHALT, and NEVERDIVERGE are undecidable. In each case, the core of the argument is describing how to transform the incoming machine-and-input encoding $\langle M, w\rangle$ into the encoding of an appropriate new Turing machine $\langle M_w\rangle$.

Now that we know that NEVERACCEPT and its relatives are undecidable, we can use them as the basis of further reduction proofs. Here is a typical example:

Theorem 14. *The language* $\textsc{DivergeSame} := \left\{\langle M_1\rangle\langle M_2\rangle \;\middle|\; \textsc{Diverge}(M_1) = \textsc{Diverge}(M_2)\right\}$ *is undecidable.*

Proof: Suppose for the sake of argument that there is a Turing machine DS that decides DIVERGESAME. Then we can build a Turing machine ND that decides NEVERDIVERGE as follows. Fix a Turing machine Y that accepts Σ^* (for example, by defining $\delta(\mathsf{start}, a) = (\mathsf{accept}, \cdot, \cdot)$ for all $a \in \Gamma$). Given an arbitrary Turing machine encoding $\langle M\rangle$ as input, ND writes the string $\langle M\rangle\langle Y\rangle$ onto the tape and then passes control to DS. There are two cases to consider:

- If DS accepts $\langle M\rangle\langle Y\rangle$, then $\textsc{Diverge}(M) = \textsc{Diverge}(Y) = \varnothing$, so $\langle M\rangle \in$ NEVERDIVERGE.
- If DS rejects $\langle M\rangle\langle Y\rangle$, then $\textsc{Diverge}(M) \neq \textsc{Diverge}(Y) = \varnothing$, so $\langle M\rangle \notin$ NEVERDIVERGE.

In short, ND accepts $\langle M\rangle$ if and only if $\langle M\rangle \in$ NEVERDIVERGE, which is impossible. We conclude that DS does not exist. □

7.11 Rice's Theorem

In 1953, Henry Rice proved the following extremely powerful theorem, which essentially states that ***every*** interesting question about the language accepted by a Turing machine is undecidable.

★★★

The following formulation is closer to the proof and may be (slightly) easier to use:

> **Rice's Theorem.** *For any set $\mathcal{L}$ of languages, if $\varnothing \notin \mathcal{L}$ and there is a Turing machine M such that $\textsc{Accept}(M) \in \mathcal{L}$, then the language $\textsc{AcceptIn}(\mathcal{L}) := \{\langle M\rangle \mid \textsc{Accept}(M) \in \mathcal{L}\}$ is undecidable.*

The only downside of this formulation is that when $\varnothing \in \mathcal{L}$, we need to consider either the complementary property $\overline{\mathcal{L}} = 2^{\Sigma^*} \setminus \mathcal{L}$ or the complementary language $\{\langle M\rangle \mid \textsc{Accept}(M) \notin \mathcal{L}\}$.

Rice's Theorem. *Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:*

- *There is a Turing machine Y such that $\textsc{Accept}(Y) \in \mathcal{L}$.*
- *There is a Turing machine N such that $\textsc{Accept}(N) \notin \mathcal{L}$.*

The language $\textsc{AcceptIn}(\mathcal{L}) := \left\{\langle M\rangle \;\middle|\; \textsc{Accept}(M) \in \mathcal{L}\right\}$ *is undecidable.*

Proof: Without loss of generality, suppose $\varnothing \notin \mathcal{L}$. (A symmetric argument establishes the theorem in the opposite case $\varnothing \in \mathcal{L}$.) Fix an arbitrary Turing machine Y such that $\textsc{Accept}(Y) \in \mathcal{L}$.

Suppose to the contrary that there is a Turing machine $A_{\mathcal{L}}$ that decides $\textsc{AcceptIn}(\mathcal{L})$. To derive a contradiction, we describe a Turing machine H that decides the halting language HALT, using $A_{\mathcal{L}}$ as a black-box subroutine. Given the encoding $\langle M, w\rangle$ of an arbitrary Turing machine M and an arbitrary string w as input, H writes the encoding $\langle WTF\rangle$ of a new Turing machine WTF that executes the following algorithm:

```
WTF(x):
  run M on input w (and discard the result)
  run Y on input x
```

H then passes the new encoding $\langle WTF \rangle$ to $A_{\mathcal{L}}$.

Now let M be an arbitrary Turing machine and w be an arbitrary string, and suppose we run our new Turing machine H on the encoding $\langle M, w \rangle$. There are two cases to consider.

- Suppose M halts on input w.
  - Then for all strings x, the machine WTF accepts x if and only if Y accepts x.
  - So $\textsc{Accept}(WTF) = \textsc{Accept}(Y)$, by definition of $\textsc{Accept}(\cdot)$.
  - So $\textsc{Accept}(WTF) \in \mathcal{L}$, by definition of Y.
  - So $A_{\mathcal{L}}$ accepts $\langle WTF \rangle$, because $A_{\mathcal{L}}$ decides $\textsc{AcceptIn}(\mathcal{L})$.
  - So H accepts $\langle M, w \rangle$, by definition of H.
- Suppose M does not halt on input w.
  - Then for all strings x, the machine WTF does not halt on input x, and therefore does not accept x.
  - So $\textsc{Accept}(WTF) = \varnothing$, by definition of $\textsc{Accept}(WTF)$.
  - So $\textsc{Accept}(WTF) \notin \mathcal{L}$, by our assumption that $\varnothing \notin \mathcal{L}$.
  - So $A_{\mathcal{L}}$ rejects $\langle WTF \rangle$, because $A_{\mathcal{L}}$ decides $\textsc{AcceptIn}(\mathcal{L})$.
  - So H rejects $\langle M, w \rangle$, by definition of H.

In short, H decides the language $\textsc{Halt}$, which is impossible. We conclude that $A_{\mathcal{L}}$ does not exist. □

The set $\mathcal{L}$ in the statement of Rice's Theorem is often called a ***property*** of languages, rather than a *set*, to avoid the inevitable confusion about sets of sets of finite sequences of characters. We can also think of $\mathcal{L}$ as a ***decision problem*** about languages, where the languages are represented by Turing machines that accept or decide them. Rice's theorem states that the ***only*** properties of languages that are decidable are the trivial properties "Does this Turing machine accept an acceptable language?" (Answer: Yes, by definition.) and "Does this Turing machine accept Discover?" (Answer: No, because Discover is a credit card, not a language.)

Rice's Theorem makes it incredibly easy to prove that language properties are undecidable; we only need to exhibit one acceptable language that has the property and another acceptable language that does not. In fact, ***every*** proof using Rice's theorem can use at least one of the following Turing machines:

- $M_{\textsc{Accept}}$ accepts every string, by defining $\delta(\mathsf{start}, a) = \mathsf{accept}$ for every tape symbol a.
- $M_{\textsc{Reject}}$ rejects every string, by defining $\delta(\mathsf{start}, a) = \mathsf{reject}$ for every tape symbol a.
- $M_{\textsc{Diverge}}$ diverges on every string, by defining $\delta(\mathsf{start}, a) = (\mathsf{start}, a, +1)$ for every tape symbol a.

Corollary 15. *Each of the following languages is undecidable.*
(a) $\{\langle M \rangle \mid M$ *accepts given an empty initial tape*$\}$
(b) $\{\langle M \rangle \mid M$ *accepts the string* `UIUC`$\}$
(c) $\{\langle M \rangle \mid M$ *accepts exactly three strings*$\}$

(d) $\{\langle M\rangle \mid M$ *accepts all palindromes*$\}$
(e) $\{\langle M\rangle \mid \textsc{Accept}(M)$ *is regular*$\}$
(f) $\{\langle M\rangle \mid \textsc{Accept}(M)$ *is not regular*$\}$
(g) $\{\langle M\rangle \mid \textsc{Accept}(M)$ *is undecidable*$\}$
(h) $\{\langle M\rangle \mid \textsc{Accept}(M) = \textsc{Accept}(N)\}$, *for some arbitrary fixed Turing machine* N.

Proof: In all cases, undecidability follows from Rice's theorem.

(a) Let $\mathcal{L}$ be the set of all languages that contain the empty string. Then $\textsc{AcceptIn}(\mathcal{L}) = \{\langle M\rangle \mid M$ accepts given an empty initial tape$\}$.

- Given an empty initial tape, $M_{\textsc{Accept}}$ accepts, so $\textsc{Accept}(M_{\textsc{Accept}}) \in \mathcal{L}$.
- Given an empty initial tape, $M_{\textsc{Diverge}}$ does not accept, so $\textsc{Accept}(M_{\textsc{Diverge}}) \notin \mathcal{L}$.

Therefore, Rice's Theorem implies that $\textsc{AcceptIn}(\mathcal{L})$ is undecidable.

(b) Let $\mathcal{L}$ be the set of all languages that contain the string `UIUC`.

- $M_{\textsc{Accept}}$ accepts `UIUC`, so $\textsc{Accept}(M_{\textsc{Accept}}) \in \mathcal{L}$.
- $M_{\textsc{Diverge}}$ does not accept `UIUC`, so $\textsc{Accept}(M_{\textsc{Diverge}}) \notin \mathcal{L}$.

Therefore, $\textsc{AcceptIn}(\mathcal{L}) = \{\langle M\rangle \mid M$ accepts the string `UIUC`$\}$ is undecidable by Rice's Theorem.

(c) There is a Turing machine that accepts the language $\{$`larry`, `curly`, `moe`$\}$. On the other hand, $M_{\textsc{Reject}}$ does not accept exactly three strings.

(d) $M_{\textsc{Accept}}$ accepts all palindromes, and $M_{\textsc{Reject}}$ does not accept all palindromes.

(e) $M_{\textsc{Reject}}$ accepts the regular language $\varnothing$, and there is a Turing machine $M_{0^n1^n}$ that accepts the non-regular language $\{0^n1^n \mid n \geq 0\}$.

(f) $M_{\textsc{Reject}}$ accepts the regular language $\varnothing$, and there is a Turing machine $M_{0^n1^n}$ that accepts the non-regular language $\{0^n1^n \mid n \geq 0\}$.[4]

(g) $M_{\textsc{Reject}}$ accepts the decidable language $\varnothing$, and there is a Turing machine that *accepts* the undecidable language $\textsc{SelfReject}$.

(h) The Turing machine N accepts $\textsc{Accept}(N)$ by definition. For the negative Turing machine $M_{\textsc{Accept}}$ accepts Σ^* and the Turing machine $M_{\textsc{Reject}}$ accepts $\varnothing$, so at least one of those two machines does not accept $\textsc{Accept}(N)$. □

We can also use Rice's theorem as a component in more complex undecidability proofs, where the target language consists of more than just a single Turing machine encoding.

Theorem 16. *The language* $L := \left\{\langle M, w\rangle \;\middle|\; M \text{ accepts } w^k \text{ for every integer } k \geq 0\right\}$ *is undecidable.*

Proof: Fix an arbitrary string w, and let $\mathcal{L}$ be the set of all languages that contain w^k for all k. Then $\textsc{Accept}(M_{\textsc{Accept}}) = \Sigma^* \in \mathcal{L}$ and $\textsc{Accept}(M_{\textsc{Reject}}) = \varnothing \notin \mathcal{L}$. Thus, *even if the string* w *is fixed in advance*, no Turing machine can decide L. □

Nearly identical reduction arguments imply the following variants of Rice's theorem. (The names of these theorems are not standard.)

[4] Yes, parts (e) and (f) have exactly the same proof.

Rice's Rejection Theorem. *Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:*

- *There is a Turing machine Y such that $\textsc{Reject}(Y) \in \mathcal{L}$*
- *There is a Turing machine N such that $\textsc{Reject}(N) \notin \mathcal{L}$.*

The language $\textsc{RejectIn}(\mathcal{L}) := \big\{\langle M\rangle \;\big|\; \textsc{Reject}(M) \in \mathcal{L}\big\}$ is undecidable.

Rice's Halting Theorem. *Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:*

- *There is a Turing machine Y such that $\textsc{Halt}(Y) \in \mathcal{L}$*
- *There is a Turing machine N such that $\textsc{Halt}(N) \notin \mathcal{L}$.*

The language $\textsc{HaltIn}(\mathcal{L}) := \big\{\langle M\rangle \;\big|\; \textsc{Halt}(M) \in \mathcal{L}\big\}$ is undecidable.

Rice's Divergence Theorem. *Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:*

- *There is a Turing machine Y such that $\textsc{Diverge}(Y) \in \mathcal{L}$*
- *There is a Turing machine N such that $\textsc{Diverge}(N) \notin \mathcal{L}$.*

The language $\textsc{DivergeIn}(\mathcal{L}) := \big\{\langle M\rangle \;\big|\; \textsc{Diverge}(M) \in \mathcal{L}\big\}$ is undecidable.

Rice's Decision Theorem. *Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:*

- *There is a Turing machine Y that **decides** an language in $\mathcal{L}$.*
- *There is a Turing machine N that **decides** an language not in $\mathcal{L}$.*

*The language $\textsc{DecideIn}(\mathcal{L}) := \big\{\langle M\rangle \;\big|\; M$ **decides** a language in $\mathcal{L}\big\}$ is undecidable.*

As easy as it is to use Rice's theorem and its variants, they cannot be used for all undecidability proofs; these theorems only apply to properties of *languages*. For example, the language $\textsc{ThisIsSparta} := \{\langle M\rangle \mid M$ accepts the string SPARTA after exactly 300 steps$\}$ is decidable, even though there are Turing machines that accept the string SPARTA after exactly 300 steps and there are other Turing machines that do not.

More subtly, Rice's theorem cannot be applied to self-referential languages like $\textsc{RevAccept} := \{\langle M\rangle \mid M$ accepts $\langle M\rangle^R\}$, because membership depends on details of the encoded machine and not just the language that the encoded machine accepts. To be clear: $\textsc{RevAccept}$ ***is*** undecidable; you just can't use Rice's theorem to prove that fact.

*7.12 The Rice-McNaughton-Myhill-Shapiro Theorem

The following subtle generalization of Rice's theorem precisely characterizes which properties of acceptable languages are *acceptable*. This result was partially proved by Henry Rice in 1953, in the same paper that proved Rice's Theorem; Robert McNaughton, John Myhill, and Norman Shapiro completed the proof a few years later, each independently from the other two.[5]

The Rice-McNaughton-Myhill-Shapiro Theorem. *Let $\mathcal{L}$ be an arbitrary set of acceptable languages. The language $\textsc{AcceptIn}(\mathcal{L}) := \{\langle M\rangle \mid \textsc{Accept}(M) \in \mathcal{L}\}$ is **acceptable** if and only if $\mathcal{L}$ satisfies the following conditions:*

*(a) $\mathcal{L}$ is **monotone**: For any language $L \in \mathcal{L}$, every superset of L is also in $\mathcal{L}$.*

*(b) $\mathcal{L}$ is **compact**: Every language in $\mathcal{L}$ has a finite subset that is also in $\mathcal{L}$.*

[5] McNaughton never published his proof (although he did announce the result); consequently, this theorem is sometimes called "The Rice-Myhill-Shapiro Theorem". Even more confusingly, Myhill published his proof twice, once in a paper with John Shepherdson and again in a later paper with Jacob Dekker. So maybe it should be called the Rice–Dekker-Myhill–(McNaughton–)Myhill-Shepherdson–Shapiro Theorem.

*(c) $\mathcal{L}$ is **finitely acceptable**: The language $\{\langle L\rangle \mid L \in \mathcal{L}$ and L is finite$\}$ is acceptable.*[6]

I won't give a complete proof of this theorem (in part because it requires techniques I haven't introduced), but the following lemma is arguably the most interesting component:

Lemma 17. *Let $\mathcal{L}$ be a set of acceptable languages. If $\mathcal{L}$ is not monotone, then* AcceptIn$(\mathcal{L})$ *is unacceptable.*

Proof: Suppose to the contrary that there is a Turing machine $AI_{\mathcal{L}}$ that accepts AcceptIn$(\mathcal{L})$. Using this Turing machine as a black box, we describe a Turing machine SD that accepts the unacceptable language SelfDiverge. Fix two Turing machines Y and N such that

$$\begin{aligned} &\textsc{Accept}(Y) \in \mathcal{L}, \\ &\textsc{Accept}(N) \notin \mathcal{L}, \\ \text{and}\quad &\textsc{Accept}(Y) \subseteq \textsc{Accept}(N). \end{aligned}$$

Let w be the input to SD. After verifying that $w = \langle M\rangle$ for some Turing machine M (and rejecting otherwise), SD writes the encoding $\langle WTF\rangle$ or a new Turing machine WTF that implements the following algorithm:

```
WTF(x):
  write x to second tape
  write ⟨M⟩ to third tape
  in parallel:
      run Y on the first tape
      run N on the second tape
      run M on the third tape
  if Y accepts x
      accept
  if N accepts x and M halts on ⟨M⟩
      accept
```

Finally, SD passes the new encoding $\langle WTF\rangle$ to $AI_{\mathcal{L}}$. There are two cases to consider:

- If M halts on $\langle M\rangle$, then $\textsc{Accept}(WTF) = \textsc{Accept}(N) \notin \mathcal{L}$, and therefore $AI_{\mathcal{L}}$ does not accept $\langle WTF\rangle$.
- If M does not halt on $\langle M\rangle$, then $\textsc{Accept}(WTF) = \textsc{Accept}(Y) \in \mathcal{L}$, and therefore $AI_{\mathcal{L}}$ accepts $\langle WTF\rangle$.

In short, SD accepts SelfDiverge, which is impossible. We conclude that SD does not exist. □

Corollary 18. *Each of the following languages is unacceptable.*
(a) $\{\langle M\rangle \mid$ *Accept*(M) *is finite*$\}$
(b) $\{\langle M\rangle \mid$ *Accept*(M) *is infinite*$\}$
(c) $\{\langle M\rangle \mid$ *Accept*(M) *is regular*$\}$
(d) $\{\langle M\rangle \mid$ *Accept*(M) *is not regular*$\}$
(e) $\{\langle M\rangle \mid$ *Accept*(M) *is decidable*$\}$

[6] Here the encoding $\langle L\rangle$ of a finite language $L \subseteq \Sigma^*$ is exactly the string that you would write down to explicitly describe L. Formally, $\langle L\rangle$ is the unique string over the alphabet $\Sigma \cup \{\{, \bullet, \}, \varepsilon\}$ that contains the strings in L in lexicographic order, separated by commas $\bullet$ and surrounded by braces $\{\}$, with ε representing the empty string. For example, $\big\langle\{\varepsilon, 0, 01, 0110, 01101001\}\big\rangle = \{\varepsilon\bullet 0\bullet 01\bullet 0110\bullet 01101001\}$.

(f) $\{\langle M\rangle \mid \textit{Accept}(M) \textit{ is undecidable}\}$
(g) $\{\langle M\rangle \mid M \textit{ accepts at least one string in } \textit{SelfDiverge}\}$
(h) $\{\langle M\rangle \mid \textit{Accept}(M) = \textit{Accept}(N)\}$, *for some arbitrary fixed Turing machine* N.

Proof: (a) The set of finite languages is not monotone: $\varnothing$ is finite; Σ^* is not finite; both $\varnothing$ and Σ^* are acceptable (in fact decidable); and $\varnothing \subset \Sigma^*$.

(b) The set of infinite acceptable languages is not compact: No finite subset of the infinite acceptable language Σ^* is infinite!

(c) The set of regular languages is not monotone: Consider the languages $\varnothing$ and $\{0^n 1^n \mid n \geq 0\}$.

(d) The set of non-regular acceptable languages is not monotone: Consider the languages $\{0^n 1^n \mid n \geq 0\}$ and Σ^*.

(e) The set of decidable languages is not monotone: Consider the languages $\varnothing$ and SelfReject.

(f) The set of undecidable acceptable languages is not monotone: Consider the languages SelfReject and Σ^*.

(g) The set $\mathcal{L} = \{L \mid L \cap \textsc{SelfDiverge} \neq \varnothing\}$ is not finitely acceptable. For any string w, deciding whether $\{w\} \in \mathcal{L}$ is equivalent to deciding whether $w \in \textsc{SelfDiverge}$, which is impossible.

(h) If $\textsc{Accept}(N) \neq \Sigma^*$, then the set $\{\textsc{Accept}(N)\}$ is not monotone. On the other hand, if $\textsc{Accept}(N) = \Sigma^*$, then the set $\{\textsc{Accept}(N)\}$ is not compact: No finite subset of Σ^* is equal to Σ^*!

□

7.13 Turing Machine Behavior: It's Complicated

Rice's theorems imply that every interesting question about the language that a Turing machine accepts—or more generally, the function that a program computes—is undecidable. A more subtle question is whether we can recognize Turing machines that exhibit certain *internal behavior*. Some behaviors we can recognize; others we can't.

Theorem 19. *The language* $\textsc{NeverLeft} := \{\langle M, w\rangle \mid \textit{Given } w \textit{ as input, } M \textit{ never moves left}\}$ *is decidable.*

Proof: Given the encoding $\langle M, w\rangle$, we simulate M with input w using our universal Turing machine U, but with the following termination conditions. If M ever moves its head to the left, then we reject. If M halts without moving its head to the left, then we accept. Finally, if M reads more than $|Q|$ blanks, where Q is the state set of M, then we accept. If the first two cases do not apply, M only moves to the right; moreover, after reading the entire input string, M only reads blanks. Thus, after reading $|Q|$ blanks, it must repeat some state, and therefore loop forever without moving to the left. The three cases are exhaustive. □

Theorem 20. *The language* $\textsc{LeftThree} := \{\langle M, w\rangle \mid \textit{Given } w \textit{ as input, } M \textit{ eventually moves left three times in a row}\}$ *is undecidable.*

Proof: Given $\langle M\rangle$, we build a new Turing machine M' that accepts the same language as M and moves left three times in a row if and only if it accepts, as follows. For each non-accepting state p

of M, the new machine M' has three states p_1, p_2, p_3, with the following transitions:

$$\begin{aligned}
\delta'(p_1, a) &= (q_2, b, \Delta), && \text{where } (q, b, \Delta) = \delta(p, a) \text{ and } q \neq \mathsf{accept} \\
\delta'(p_2, a) &= (p_3, a, +1) \\
\delta'(p_3, a) &= (p_1, a, -1)
\end{aligned}$$

In other words, after each non-accepting transition, M' moves once to the right and then once to the left. For each transition to accept, M' has a sequence of seven transitions: three steps to the right, then three steps to the left, and then finally accept$'$, all without modifying the tape. (The three steps to the right ensure that M' does not fall off the left end of the tape.)

Finally, M' moves left three times in a row if and only if M accepts w. Thus, if we could decide LeftThree, we could also decide Accept, which is impossible. □

There is no hard and fast rule like Rice's theorem to distinguish decidable behaviors from undecidable behaviors, but I can offer two rules of thumb.

- If it is possible to simulate an arbitrary Turing machine while avoiding the target behavior, then the behavior is not decidable. For example, there is no algorithm to determine whether a given Turing machine reenters its start state, or revisits the left end of the tape, or writes a blank.
- If a Turing machine with the target behavior is limited to a finite number of configurations, or is guaranteed to force an infinite loop after a finite number of transitions, then the behavior is likely to be decidable. For example, there *are* algorithms to determine whether a given Turing machine ever leaves its start state, or reads its entire input string, or writes a non-blank symbol over a blank.

Exercises

1. Let M be an arbitrary Turing machine.
   (a) Describe a Turing machine M^R such that
   $$\textsc{Accept}(M^R) = \textsc{Reject}(M) \quad \text{and} \quad \textsc{Reject}(M^R) = \textsc{Accept}(M).$$
   (b) Describe a Turing machine M^A such that
   $$\textsc{Accept}(M^A) = \textsc{Accept}(M) \quad \text{and} \quad \textsc{Reject}(M^A) = \varnothing.$$
   (c) Describe a Turing machine M^H such that
   $$\textsc{Accept}(M^H) = \textsc{Halt}(M) \quad \text{and} \quad \textsc{Reject}(M^H) = \varnothing.$$

2. (a) Prove that Accept is undecidable.
   (b) Prove that Reject is undecidable.
   (c) Prove that Diverge is undecidable.

3. (a) Prove that NeverReject is undecidable.

(b) Prove that NeverHalt is undecidable.

(c) Prove that NeverDiverge is undecidable.

4. Prove that each of the following languages is undecidable.

   (a) $\textsc{AlwaysAccept} := \{\langle M\rangle \mid \textsc{Accept}(M) = \Sigma^*\}$

   (b) $\textsc{AlwaysReject} := \{\langle M\rangle \mid \textsc{Reject}(M) = \Sigma^*\}$

   (c) $\textsc{AlwaysHalt} := \{\langle M\rangle \mid \textsc{Halt}(M) = \Sigma^*\}$

   (d) $\textsc{AlwaysDiverge} := \{\langle M\rangle \mid \textsc{Diverge}(M) = \Sigma^*\}$

5. Let $\mathcal{L}$ be a non-empty proper subset of the set of acceptable languages. Prove that the following languages are undecidable:

   (a) $\textsc{RejectIn}(\mathcal{L}) := \left\{\langle M\rangle \;\middle|\; \textsc{Reject}(M) \in \mathcal{L}\right\}$

   (b) $\textsc{HaltIn}(\mathcal{L}) := \left\{\langle M\rangle \;\middle|\; \textsc{Halt}(M) \in \mathcal{L}\right\}$

   (c) $\textsc{DivergeIn}(\mathcal{L}) := \left\{\langle M\rangle \;\middle|\; \textsc{Diverge}(M) \in \mathcal{L}\right\}$

6. For each of the following decision problems, either *sketch* an algorithm or prove that the problem is undecidable. Recall that w^R denotes the reversal of string w. For each problem, the input is the encoding $\langle M\rangle$ of a Turing machine M.

   (a) Does M reject the empty string?

   (b) Does M accept $\langle M\rangle^R$?

   (c) Does M accept $\langle M\rangle\langle M\rangle$?

   (d) Does M accept $\langle M\rangle^k$ for any integer k?

   (e) Does M accept the encoding of any Turing machine?

   (f) Is there a Turing machine that accepts $\langle M\rangle$?

   (g) Is $\langle M\rangle$ a palindrome?

   (h) Does M reject any palindrome?

   (i) Does M accept all palindromes?

   (j) Does M diverge only on palindromes?

   (k) Is there an input string that forces M to move left?

   (l) Is there an input string that forces M to move left three times in a row?

   (m) Does M accept the encoding of any Turing machine N such that $\textsc{Accept}(N) = \textsc{SelfDiverge}$?

7. For each of the following decision problems, either *sketch* an algorithm or prove that the problem is undecidable. Recall that w^R denotes the reversal of string w. For each problem, the input is an encoding $\langle M, w\rangle$ of a Turing machine M and its input string w.

   (a) Does M accept the string ww^R?

(b) Does M accept either w or w^R?

(c) Does M either accept w or reject w^R?

(d) Does M accept the string w^k for some integer k?

(e) Does M accept w in at most $2^{|w|}$ steps?

(f) If we run M on input w, does M ever change a symbol on its tape?

(g) If we run M on input w, does M ever move to the right?

(h) If we run M on input w, does M ever move to the right twice in a row?

(i) If we run M on input w, does M move its head to the right more than $2^{|w|}$ times (not necessarily consecutively)?

(j) If we run M with input w, does M ever change a $\square$ on the tape to any other symbol?

(k) If we run M with input w, does M ever change a $\square$ on the tape to 1?

(l) If we run M with input w, does M ever write a $\square$?

(m) If we run M with input w, does M ever leave its start state?

(n) If we run M with input w, does M ever reenter its start state?

(o) If we run M with input w, does M ever reenter a state that it previously left? That is, are there states $p \neq q$ such that M moves from state p to state q and then later moves back to state p?

8. Let M be a Turing machine, let w be an arbitrary input string, and let s and t be positive integers integer. We say that M accepts w ***in space s*** if M accepts w after accessing at most the first s cells on the tape, and M accepts w ***in time t*** if M accepts w after at most t transitions.

   (a) Prove that the following languages are decidable:

      i. $\{\langle M, w\rangle \mid M \text{ accepts } w \text{ in time } |w|^2\}$

      ii. $\{\langle M, w\rangle \mid M \text{ accepts } w \text{ in space } |w|^2\}$

   (b) Prove that the following languages are undecidable:

      i. $\{\langle M\rangle \mid M \text{ accepts at least one string } w \text{ in time } |w|^2\}$

      ii. $\{\langle M\rangle \mid M \text{ accepts at least one string } w \text{ in space } |w|^2\}$

9. Let L_0 be an arbitrary language. For any integer $i > 0$, define the language

$$L_i := \{\langle M\rangle \mid M \text{ decides } L_{i-1}\}.$$

   For which integers $i > 0$ is L_i decidable? Obviously the answer depends on the initial language L_0; give a complete characterization of all possible cases. Prove your answer is correct. *[Hint: This question is a lot easier than it looks!]*

10. Argue that each of the following decision problems about programs in your favorite programming language are undecidable.

   (a) Does this program correctly compute Fibonacci numbers?

(b) Can this program fall into an infinite loop?

(c) Will the value of this variable ever change?

(d) Will this program every attempt to deference a null pointer?

(e) Does this program free every block of memory that it dynamically allocates?

(f) Is any statement in this program unreachable?

(g) Do these two programs compute the same function?

*11. Call a Turing machine ***conservative*** if it never writes over its input string. More formally, a Turing machine is conservative if for every transition $\delta(p, a) = (q, b, \Delta)$ where $a \in \Sigma$, we have $b = a$; and for every transition $\delta(p, a) = (q, b, \Delta)$ where $a \notin \Sigma$, we have $b \neq \Sigma$.

(a) Prove that if M is a conservative Turing machine, then $\textsc{Accept}(M)$ is a regular language.

(b) Prove that the language $\{\langle M \rangle \mid M \text{ is conservative and } M \text{ accepts } \varepsilon\}$ is undecidable.

Together, these two results imply that every conservative Turing machine accepts the same language as some DFA, but it is impossible to determine *which* DFA.

★12. (a) Prove that it is undecidable whether a given C++ program is syntactically correct. *[Hint: Use templates!]*

(b) Prove that it is undecidable whether a given ANSI C program is syntactically correct. *[Hint: Use the preprocessor!]*

(c) Prove that it is undecidable whether a given Perl program is syntactically correct. *[Hint: Does that slash character / delimit a regular expression or represent division?]*

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

Caveat lector: This note is not even a first draft, but more of a rough sketch, with many topics still to be written and/or unwritten. But the semester is over, so it's time to put it down. Please send bug reports and suggestions to jeffe@illinois.edu.

Any sufficiently advanced technology is indistinguishable from magic.
— Arthur C. Clarke, "Hazards of Prophecy: The Failure of Imagination" (1962)

Any technology that is distinguishable from magic is insufficiently advanced.
— Barry Gehm, quoted by Stan Schmidt in *ANALOG* magazine (1991)

8 Universal Models of Computation

★★★

Remind about the Church-Turing thesis.
There is some confusion here between **universal models of computation** and the somewhat wider class of **undecidable problems/languages**.

8.1 Universal Turing Machines

The pinnacle of Turing machine constructions is the ***universal*** Turing machine. For modern computer scientists, it's useful to think of a universal Turing machine as a "Turing machine *interpreter* written in Turing machine". Just as the input to a Python interpreter is a string of Python source code, the input to our universal Turing machine U is a string $\langle M, w\rangle$ that encodes both an arbitrary Turing machine M and a string w in the input alphabet of M. Given these encodings, U simulates the execution of M on input w; in particular,

- U accepts $\langle M, w\rangle$ if and only if M accepts w.
- U rejects $\langle M, w\rangle$ if and only if M rejects w.

In the next few pages, I will sketch a universal Turing machine U that uses the input alphabet $\{\mathtt{0}, \mathtt{1}, \mathtt{[}, \mathtt{]}, \bullet, \mathtt{|}\}$ and a somewhat larger tape alphabet. However, I do *not* require that the Turing machines that U simulates have similarly small alphabets, so we first need a method to encode *arbitrary* input and tape alphabets.

Encodings

Let $M = (\Gamma, \square, \Sigma, Q, \mathit{start}, \mathit{accept}, \mathit{reject}, \delta)$ be an arbitrary Turing machine, with a single half-infinite tape and a single read-write head. (I will consistently indicate the states and tape symbols of M in *slanted green* to distinguish them from the upright red states and tape symbols of U.)

We encode each symbol $a \in \Gamma$ as a unique string $|a|$ of $\lceil \lg(|\Gamma|)\rceil$ bits. Thus, if $\Gamma = \{\mathit{0}, \mathit{1}, \mathit{\$}, \mathit{x}, \square\}$, we might use the following encoding:

$$\langle \mathit{0}\rangle = \mathtt{001}, \qquad \langle \mathit{1}\rangle = \mathtt{010}, \qquad \langle \mathit{\$}\rangle = \mathtt{011}, \qquad \langle \mathit{x}\rangle = \mathtt{100}, \qquad \langle \square\rangle = \mathtt{000}.$$

The input string w is encoded by its sequence of symbol encodings, with separators $\bullet$ between every pair of symbols and with brackets $\mathtt{[}$ and $\mathtt{]}$ around the whole string. For example, with this encoding, the input string $\mathit{001100}$ would be encoded on the input tape as

$$\langle \mathit{001100}\rangle = \mathtt{[001\bullet001\bullet010\bullet010\bullet001\bullet001]}$$

© Copyright 2017 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/ for the most recent revision.

Similarly, we encode each state $q \in Q$ as a distinct string $\langle q \rangle$ of $\lceil \lg|Q| \rceil$ bits. Without loss of generality, we encode the start state with all 1s and the reject state with all 0s. For example, if $Q = \{\mathit{start}, \mathit{seek1}, \mathit{seek0}, \mathit{reset}, \mathit{verify}, \mathit{accept}, \mathit{reject}\}$, we might use the following encoding:

$$\langle \mathit{start} \rangle = \texttt{111} \qquad \langle \mathit{seek1} \rangle = \texttt{010} \qquad \langle \mathit{seek0} \rangle = \texttt{011} \qquad \langle \mathit{reset} \rangle = \texttt{100}$$
$$\langle \mathit{verify} \rangle = \texttt{101} \qquad \langle \mathit{accept} \rangle = \texttt{110} \qquad \langle \mathit{reject} \rangle = \texttt{000}$$

We encode the machine M itself as the string $\langle M \rangle = \texttt{[}\langle \mathit{reject} \rangle \bullet \langle \square \rangle\texttt{]}\langle \delta \rangle$, where $\langle \delta \rangle$ is the concatenation of substrings $\texttt{[}\langle p \rangle \bullet \langle a \rangle \texttt{|} \langle q \rangle \bullet \langle b \rangle \bullet \langle \Delta \rangle\texttt{]}$ encoding each transition $\delta(p, a) = (q, b, \Delta)$ such that $q \neq \mathit{reject}$. We encode the actions $\Delta = \pm 1$ by defining $\langle -1 \rangle := \texttt{0}$ and $\langle +1 \rangle := \texttt{1}$. Conveniently, every transition string has exactly the same length. For example, with the symbol and state encodings described above, the transition $\delta(\mathit{reset}, \$) = (\mathit{start}, \$, +1)$ would be encoded as

$$\texttt{[100•011|001•011•1]}.$$

Our first example Turing machine for recognizing $\{0^n 1^n 0^n \mid n \geq 0\}$ would be represented by the following string (here broken into multiple lines for readability):

```
[000•000][[001•001|010•011•1][001•100|101•011•1]
          [010•001|010•001•1][010•100|010•100•1]
          [010•010|011•100•1][011•010|011•010•1]
          [011•100|011•100•1][011•001|100•100•1]
          [100•001|100•001•0][100•010|100•010•0]
          [100•100|100•100•0][100•011|001•011•1]
          [101•100|101•011•1][101•000|110•000•0]]
```

Finally, we encode any *configuration* of M on U's work tape by alternating between encodings of states and encodings of tape symbols. Thus, each tape cell is represented by the string $\texttt{[}\langle q \rangle \bullet \langle a \rangle\texttt{]}$ indicating that (1) the cell contains symbol a; (2) if $q \neq \mathit{reject}$, then M's head is located at this cell, and M is in state q; and (3) if $q = \mathit{reject}$, then M's head is located somewhere else. Conveniently, each cell encoding uses exactly the same number of bits. We also surround the entire tape encoding with brackets [and].

For example, with the encodings described above, the initial configuration $(\mathit{start}, \underset{\blacktriangle}{0}01100, 0)$ for our first example Turing machine would be encoded on U's tape as follows.

$$\langle \mathit{start}, \underset{\blacktriangle}{0}01100, 0 \rangle = \texttt{[}\underbrace{\texttt{[111•001]}}_{\textbf{start } 0}\underbrace{\texttt{[000•001]}}_{\text{reject } 0}\underbrace{\texttt{[000•010]}}_{\text{reject } 1}\underbrace{\texttt{[000•010]}}_{\text{reject } 1}\underbrace{\texttt{[000•001]}}_{\text{reject } 0}\underbrace{\texttt{[000•001]}}_{\text{reject } 0}\texttt{]}$$

Similarly, the intermediate configuration $(\mathit{reset}, \$0x\underset{\blacktriangle}{1}x0, 3)$ would be encoded as follows:

$$\langle \mathit{reset}, \$\$x\underset{\blacktriangle}{1}x0, 3 \rangle = \texttt{[}\underbrace{\texttt{[000•011]}}_{\text{reject } \$}\underbrace{\texttt{[000•011]}}_{\text{reject } 0}\underbrace{\texttt{[000•100]}}_{\text{reject } x}\underbrace{\texttt{[010•010]}}_{\textbf{reset } 1}\underbrace{\texttt{[000•100]}}_{\text{reject } x}\underbrace{\texttt{[000•001]}}_{\text{reject } 0}\texttt{]}$$

Input and Execution

Without loss of generality, we assume that the input to our universal Turing machine U is given on a separate read-only ***input tape***, as the encoding of an arbitrary Turing machine M followed by an encoding of its input string x. Notice the substrings [[and]] each appear only only once on the input tape, immediately before and after the encoded transition table, respectively. U also has a read-write ***work tape***, which is initially blank.

We start by initializing the work tape with the encoding $\langle \mathit{start}, x, 0\rangle$ of the initial configuration of M with input x. First, we write `[[`$\langle \mathit{start}\rangle$`•`. Then we copy the encoded input string $\langle x\rangle$ onto the work tape, but we change the punctuation as follows:

- Instead of copying the left bracket `[`, write `[[`$\langle \mathit{start}\rangle$`•`.
- Instead of copying each separator `•`, write `][`$\langle \mathit{reject}\rangle$`•`
- Instead of copying the right bracket `]`, write two right brackets `]]`.

The state encodings $\langle \mathit{start}\rangle$ and $\langle \mathit{reject}\rangle$ can be copied directly from the beginning of $\langle M\rangle$ (replacing `0`s for `1`s for $\langle \mathit{start}\rangle$). Finally, we move the head back to the start of U's tape.

At the start of each step of the simulation, U's head is located at the start of the work tape. We scan through the work tape to the unique encoded cell `[`$\langle p\rangle$`•`$\langle a\rangle$`]` such that $p \neq \mathit{reject}$. Then we scan through the encoded transition function $\langle \delta\rangle$ to find the unique encoded tuple `[`$\langle p\rangle$`•`$\langle a\rangle$`|`$\langle q\rangle$`•`$\langle b\rangle$`•`$\langle \Delta\rangle$`]` whose left half matches our the encoded tape cell. If there is no such tuple, then U immediately halts and rejects. Otherwise, we copy the right half $\langle q\rangle$`•`$\langle b\rangle$ of the tuple to the work tape. Now if $q = \mathit{accept}$, then U immediately halts and accepts. (We don't bother to encode *reject* transformations, so we know that $q \neq \mathit{reject}$.) Otherwise, we transfer the state encoding to either the next or previous encoded cell, as indicated by M's transition function, and then continue with the next step of the simulation.

During the final state-copying phase, we ever read two right brackets `]]`, indicating that we have reached the right end of the tape encoding, we replace the second right bracket with `[`$\langle \mathit{reject}\rangle$`•`$\langle \square\rangle$`]]` (mostly copied from the beginning of the machine encoding $\langle M\rangle$) and then scan back to the left bracket we just wrote. This trick allows our universal machine to *pretend* that its tape contains an infinite sequence of *encoded* blanks `[`$\langle \mathit{reject}\rangle$`•`$\langle \square\rangle$`]` instead of *actual* blanks $\square$.

Example

As an illustrative example, suppose U is simulating our first example Turing machine M on the input string `001100`. The execution of M on input w eventually reaches the configuration (*seek1*, `$$x1x0`, 3). At the start of the corresponding step in U's simulation, U is in the following configuration:

```
[[000•011][000•011][000•100][010•010][000•100][000•001]]
```

First U scans for the first encoded tape cell whose state is not *reject*. That is, U repeatedly compares the first half of each encoded state cell on the work tape with the prefix `[`$\langle \mathit{reject}\rangle$`•` of the machine encoding $\langle M\rangle$ on the input tape. U finds a match in the fourth encoded cell.

```
[[000•011][000•011][000•100][010•010][000•100][000•001]]
```

Next, U scans the machine encoding $\langle M\rangle$ for the substring `[010•010` matching the current encoded cell. U eventually finds a match in the left size of the the encoded transition `[010•010|011•100•1]`. U copies the state-symbol pair `011•100` from the right half of this encoded transition into the current encoded cell. (The underline indicates which symbols are changed.)

```
[[000•011][000•011][000•100][011•100][000•100][000•001]]
```

The encoded transition instructs U to move the current state encoding one cell to the right. (The underline indicates which symbols are changed.)

```
[[000•011][000•011][000•100][000•100][011•100][000•001]]
```

Finally, U scans left until it reads two left brackets `[[`; this returns the head to the left end of the work tape to start the next step in the simulation. U's tape now holds the encoding of M's configuration (*seek0*, `$$xxx0`, 4), as required.

```
[[000•011][000•011][000•100][000•100][011•100][000•001]]
```

8.2 Two-Stack Machines

A ***two-stack machine*** is a Turing machine with two tapes with the following restricted behavior. At all times, on each tape, every cell to the right of the head is blank, and every cell at or to the left of the head is non-blank. Thus, a head can only move right by writing a non-blank symbol into a blank cell; symmetrically, a head can only move left by erasing the rightmost non-blank cell. Thus, each tape behaves like a stack. To avoid underflow, there is a special symbol at the start of each tape that cannot be overwritten. Initially, one tape contains the input string, with the head at its *last* symbol, and the other tape is empty (except for the start-of-tape symbol).

★★★

Simulate a doubly-infinite tape with two stacks, one holding the tape contents to the left of the head, the other holding the tape contents to the right of the head. For each transition of a standard Turing machine M, the stack machine pops the top symbol off the (say) left stack, changes its internal state according to the transition δ, and then either pushes a new symbol onto the right stack, or pushes a new symbol onto the left stack and then moves the top symbol from the right stack to the left stack.

8.3 Counter Machines

★★★

A configuration of a k-counter machine consists of k non-negative integers and an internal state from some finite set Q. The transition function $\delta : Q \times \{0, +1\}^k \to Q \times \{-1, 0, +1\}^k$ takes an internal state and the signs of the counters as input, and produces a new internal state and changes to counters as output.

- Prove that any Turing machine can be simulated by a three-counter machine. One counter holds the binary representation of the tape after the head; another counter holds the reversed binary representation of the tape before the head. Implement transitions via halving, doubling, and parity, using the third counter for scratch work.
- Prove that two counters can simulate three. Store $2^a 3^b 5^c$ in one counter, use the other for scratch work.
- Prove that a three-counter machine can compute any computable function: Given input $(n, 0, 0)$, we can compute $(f(n), 0, 0)$ for *any* computable function f. First transform $(n, 0, 0)$ to $(2^n, 0, 0)$ using all three counters; then run two- (or three-)counter TM simulation to obtain $(2^{f(n)}, 0, 0)$; and finally transform $(2^{f(n)}, 0, 0)$ to $(f(n), 0, 0)$ using all three counters.
- **HARD:** Prove that a two-counter machine cannot transform $(n, 0)$ to $(2^n, 0)$. [Barzhdin 1963, Yao 1971, Schröpel 1972]

8.4 FRACTRAN

★★★

FRACTRAN [Conway 1987]: A one-counter machine whose "program" is a sequence of rational numbers. The counter is initially 1. At each iteration, multiply the counter by the first rational number that yields an integer; if there is no such number, halt.

- Prove that for any computable function $f : \mathbb{N} \to \mathbb{N}$, there is a FRACTRAN program that transforms 2^{n+1} into $3^{f(n)+1}$, for all natural numbers n.
- Prove that every FRACTRAN program, given the integer 1 as input, either outputs 1 or loops forever. It follows that there is no FRACTRAN program for the increment function $n \mapsto n+1$.

8.5 Post Correspondence Problem

Given n of pairs of strings $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, is there a finite sequence of integers $(i_1, i_2, \ldots, i_k)$ such that $x_{i_1} x_{i_2} \cdots x_{i_k} = y_{i_1} y_{i_2} \cdots y_{i_k}$? For notation convenience, we write each pair vertically as $\left[\begin{smallmatrix} x \\ y \end{smallmatrix}\right]$ instead of horizontally as (x, y). For example, given the string pairs

$$a = \begin{bmatrix} \texttt{0} \\ \texttt{100} \end{bmatrix},\ b = \begin{bmatrix} \texttt{01} \\ \texttt{00} \end{bmatrix},\ c = \begin{bmatrix} \texttt{110} \\ \texttt{11} \end{bmatrix},$$

we should answer TRUE, because

$$cbca = \begin{bmatrix} \texttt{110} \\ \texttt{11} \end{bmatrix}\begin{bmatrix} \texttt{01} \\ \texttt{00} \end{bmatrix}\begin{bmatrix} \texttt{110} \\ \texttt{11} \end{bmatrix}\begin{bmatrix} \texttt{0} \\ \texttt{100} \end{bmatrix}$$

gives us `110110100` for both concatenations. As more extreme examples, the shortest solutions for the input

$$a = \begin{bmatrix} \texttt{0} \\ \texttt{001} \end{bmatrix},\ b = \begin{bmatrix} \texttt{001} \\ \texttt{1} \end{bmatrix},\ c = \begin{bmatrix} \texttt{1} \\ \texttt{0} \end{bmatrix}$$

have length 75; one such solution is $aacaacabbabccaaccaaaacbaabbaacbacbbccbbacbaccbcb$ $acbbacbaccbacbbbacccbabbccbaacaacaaacbabbaacacbccbbabacbcaaccbacabbbbabcccc$ $bcaababaaccbcbbbacccbabbccb$. The shortest solution for the instance

$$a = \begin{bmatrix} \texttt{0} \\ \texttt{000} \end{bmatrix},\ b = \begin{bmatrix} \texttt{0} \\ \texttt{0101} \end{bmatrix},\ c = \begin{bmatrix} \texttt{01} \\ \texttt{1} \end{bmatrix},\ d = \begin{bmatrix} \texttt{1111} \\ \texttt{10} \end{bmatrix}$$

is the unbelievable $a^2b^8a^4c^{16}ab^4a^2b^4ad^4b^3c^8a^6c^8b^2c^4bc^6d^2a^{18}d^2c^4dcad^2cb^{54}c^3dca^2c^{111}dc$ $a^6d^{28}cb^{17}c^{63}d^{16}c^{16}d^4c^4dc$, which has total length 451. Finally, the shortest solution for the instance

$$a = \begin{bmatrix} \texttt{0} \\ \texttt{00010} \end{bmatrix},\ b = \begin{bmatrix} \texttt{010} \\ \texttt{01} \end{bmatrix},\ c = \begin{bmatrix} \texttt{100} \\ \texttt{0} \end{bmatrix},$$

has length 528.

★★★

The simplest universality proof simulates a tag-Turing machine.

8.6 Matrix Mortality

★★★

Given a set of integer matrices $A_1, \ldots, A_k$, is the product of any sequence of these matrices (with repetition) equal to 0? Undecidable by reduction from PCP, even for two 15×15 matrices or six 3×3 matrices [Cassaigne, Halava, Harju, Nicolas 2014]

8.7 Dynamical Systems

★★★

Ray Tracing [Reif, Tygar, and Yoshida 1994] The configuration of a Turing machine is encoded as the (x, y) coordinates of a light path crossing the unit square $[0,1] \times [0,1]$, where the x-(resp. y-)coordinate encodes the tape contents to the left (resp. right) of the head. Need either quadratic-surface mirrors or refraction to simulate transitions.

N-body problem [Smith 2006]: Similar idea

Skolem-Pisot reachability: Given an integer vector x and an integer matrix A, does $A^n x = (0, \dots)$ for any integer n? [Halava, Harju, Hirvensalo, Karhumäki 2005] It's surprising that this problem is undecidable; the similar mortality problem for one matrix is not.

8.8 Wang Tiles

★★★

Turing machine simulation is straightforward. *Small* Turing-complete tile sets via affine maps (via two-stack machines) are a little harder.

8.9 Combinator Calculus

In the 1920s, Moses Schönfinkel developed what can now be interpreted as a model of computation now called *combinator calculus* or *combinatory logic*. Combinator calculus operates on ***terms***, where every term is either one of a finite number of ***combinators*** (represented here by upper case letters) or an ordered pair of terms. For notational convenience, we omit commas between components of every pair and parentheses around the *left* term in every pair. Thus, $\mathtt{SKK}(\mathtt{IS})$ is shorthand for the term $(((\mathtt{S}, \mathtt{K}), \mathtt{K}), (\mathtt{I}, \mathtt{S}))$.

We can "evaluate" any term by a sequence of rewriting rules that depend on its first primitive combinator. Schönfinkel defined three primitive combinators with the following evaluation rules:

- Identity: $\mathtt{I}x \mapsto x$
- Constant: $\mathtt{K}xy \mapsto x$
- Substitution: $\mathtt{S}xyz \mapsto xz(yz)$

Here, x, y, and z are variables representing unknown but arbitrary terms. "Computation" in the combinator calculus is performed by repeatedly evaluating arbitrary (sub)terms with one of these three structures, until all such (sub)terms are gone.

For example, the term $\mathtt{S}(\mathtt{K}(\mathtt{SI}))\mathtt{K}xy$ (for any terms x and y) evaluates as follows:

$$
\begin{aligned}
\underline{\mathtt{S}(\mathtt{K}(\mathtt{SI}))\mathtt{K}\mathtt{x}}y &\mapsto \underline{\mathtt{K}(\mathtt{SI})\ \mathtt{x}}(\mathtt{K}x)y && \text{Substitution} \\
&\mapsto \underline{\mathtt{SI}(\mathtt{Kx})\ \mathtt{y}} && \text{Constant} \\
&\mapsto \underline{\mathtt{I}\mathtt{y}}(\mathtt{K}xy) && \text{Substitution} \\
&\mapsto y(\underline{\mathtt{Kxy}}) && \text{Identity} \\
&\mapsto yx && \text{Constant}
\end{aligned}
$$

Thus, we can define a new combinator $\mathtt{R} := \mathtt{S}(\mathtt{K}(\mathtt{SI}))\mathtt{K}$ that upon evaluation reverses the next two terms: $\mathtt{R}xy \mapsto yx$.

On the other hand, evaluating SII(S(KI)(SII)) leads to an infinite loop:

$\underline{\texttt{SII(S(KI)(SII))}}$	$\mapsto \underline{\texttt{I(S(KI)(SII))}}\texttt{(I(S(KI)(SII)))}$	Substitution
	$\mapsto \texttt{S(KI)(SII)}(\underline{\texttt{I(S(KI)(SII))}})$	Identity
	$\mapsto \underline{\texttt{S(KI)(SII)(S(KI)(SII))}}$	Identity
	$\mapsto \underline{\texttt{KI(S(KI)(SII))}}\texttt{(SII(S(KI)(SII)))}$	Substitution
	$\mapsto \underline{\texttt{I(SII(S(KI)(SII)))}}$	Constant
	$\mapsto \texttt{SII(S(KI)(SII))}$	Identity

★★★ Wikipedia sketches a direct *undecidability* proof. Is there a Turing-completeness proof that avoids λ-calculus?

Exercises

1. A ***tag***-Turing machine has two heads: one can only read, the other can only write. Initially, the read head is located at the left end of the tape, and the write head is located at the first blank after the input string. At each transition, the read head can either move one cell to the right or stay put, but the write head *must* write a symbol to its current cell and move one cell to the right. Neither head can ever move to the left.

   Prove that any standard Turing machine can be simulated by a tag-Turing machine. That is, given any standard Turing machine M, describe a tag-Turing machine M' that accepts and rejects exactly the same input strings as M.

2. *(a) Prove that any standard Turing machine can be simulated by a Turing machine with only three states. *[Hint: Use the tape to store an encoding of the state of the machine yours is simulating.]*

   ★(b) Prove that any standard Turing machine can be simulated by a Turing machine with only *two* states.

3. A ***two-dimensional*** Turing machine uses an infinite two-dimensional grid of cells as the tape; at each transition, the head can move from its current cell to any of its four neighbors on the grid. The transition function of such a machine has the form $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\uparrow, \leftarrow, \downarrow, \rightarrow\}$, where the arrows indicate which direction the head should move.

   (a) Prove that any two-dimensional Turing machine can be simulated by a standard Turing machine.

   (b) Suppose further that we endow our two-dimensional Turing machine with the following additional actions, in addition to moving the head:

   - Insert row: Move all symbols on or above the row containing the head up one row, leaving the head's row blank.
   - Insert column: Move all symbols on or to the right of the column containing the head one column to the right, leaving the head's column blank.
   - Delete row: Move all symbols above the row containing the head down one row, deleting the head's row of symbols.
   - Delete column: Move all symbols the right of the column containing the head one column to the right, deleting the head's column of symbols.

Show that any two-dimensional Turing machine that can add an delete rows can be simulated by a standard Turing machine.

4. A ***binary-tree*** Turing machine uses an infinite binary tree as its tape; that is, *every* cell in the tape has a left child and a right child. At each step, the head moves from its current cell to its Parent, its Left child, or to its Right child. Thus, the transition function of such a machine has the form $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathsf{P}, \mathsf{L}, \mathsf{R}\}$. The input string is initially given along the left spine of the tape.

   Show that any binary-tree Turing machine can be simulated by a standard Turing machine.

5. A ***stack-tape*** Turing machine uses an semi-infinite tape, where every cell is actually the top of an independent stack. The behavior of the machine at each iteration is governed by its internal state and the symbol *at the top* of the current cell's stack. At each transition, the head can optionally push a new symbol onto the stack, or pop the top symbol off the stack. (If a stack is empty, its "top symbol" is a blank and popping has no effect.)

   Show that any stack-tape Turing machine can be simulated by a standard Turing machine. (Compare with Problem **??**!)

6. A ***tape-stack*** Turing machine has two actions that modify its work tape, in addition to simply writing individual cells: it can ***save*** the entire tape by pushing in onto a stack, and it can ***restore*** the entire tape by popping it off the stack. Restoring a tape returns the content of every cell to its content when the tape was saved. Saving and restoring the tape do not change the machine's state or the position of its head. If the machine attempts to "restore" the tape when the stack is empty, the machine crashes.

   Show that any tape-stack Turing machine can be simulated by a standard Turing machine.

★★★

- Tape alphabet = $\mathbb{N}$.
  - Read: zero or positive. Write: $+1, -1$
  - Read: even or odd. Write: $+1, -1, \times 2, \div 2$
  - Read: positive, negative, or zero. Write: $x + y$ (merge), $x - y$ (merge), 1, 0
- Never three times in a row in the same direction
- Hole-punch TM: tape alphabet $\{\square, \blacksquare\}$, and only $\square \mapsto \blacksquare$ transitions allowed.

© Copyright 2017 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/ for the most recent revision.

Caveat lector: This is a first draft; some topics still need to be written. Please send bug reports and suggestions to jeffe@illinois.edu.

If first you don't succeed, then try and try again.
And if you don't succeed again, just try and try and try.

— Marc Blitzstein, "Useless Song", *The Three Penny Opera* (1954)
Adaptation of Bertold Brecht, "Das Lied von der Unzulänglichkeit menschlichen Strebens" *Die Dreigroschenoper* (1928)

Children need encouragement.
If a kid gets an answer right, tell him it was a lucky guess.
That way he develops a good, lucky feeling.

— Jack Handey, "Deep Thoughts", *Saturday Night Live* (March 21, 1992)

9 Nondeterministic Turing Machines

9.1 Definitions

In his seminal 1936 paper, Turing also defined an extension of his "automatic machines" that he called ***choice machines***, which are now more commonly known as ***nondeterministic Turing machines***. The execution of a nondeterministic Turing machine is *not determined* entirely by its input and its transition function; rather, at each step of its execution, the machine can *choose* from a set of possible transitions. The distinction between deterministic and nondeterministic Turing machines exactly parallels the distinction between deterministic and nondeterministic finite-state automata.

Formally, a nondeterministic Turing machine has all the components of a standard deterministic Turing machine—a finite tape alphabet Γ that contains the input alphabet Σ and a blank symbol $\square$; a finite set Q of internal states with special start, accept, and reject states; and a transition function δ. However, the transition function now has the signature

$$\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{-1,+1\}}.$$

That is, for each state p and tape symbol a, the output $\delta(p,a)$ of the transition function is a *set* of triples of the form $(q,b,\Delta) \in Q \times \Gamma \times \{-1,+1\}$. Whenever the machine finds itself in state p reading symbol a, the machine *chooses* an arbitrary triple $(q,b,\Delta) \in \delta(p,a)$, and then changes its state to q, writes b to the tape, and moves the head by Δ. If the set $\delta(p,a)$ is empty, the machine moves to the reject state and halts.

The set of all possible transition sequences of a nondeterministic Turing machine N on a given input string w define a rooted tree, called a ***computation tree***. The initial configuration $(\text{start}, w, 0)$ is the root of the computation tree, and the children of any configuration (q,x,i) are the configurations that can be reached from (q,x,i) in one transition. In particular, any configuration whose state is accept or reject is a leaf. For deterministic Turing machines, this computation tree is just a single path, since there is at most one valid transition from every configuration.

© Copyright 2016 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

9.2 Acceptance and Rejection

Unlike deterministic Turing machines, there is a fundamental asymmetry between the acceptance and rejection criteria for nondeterministic Turing machines. Let N be any nondeterministic Turing machine, and let w be any string.

- ***N accepts w*** if and only if there is *at least one* sequence of valid transitions from the initial configuration $(\mathsf{start}, w, 0)$ that leads to the accept state. Equivalently, N accepts w if the computation tree contains at least one accept leaf.
- ***N rejects w*** if and only if *every* sequence of valid transitions from the initial configuration $(\mathsf{start}, w, 0)$ leads to the reject state. Equivalently, N rejects w if every path through the computation tree ends with a reject leaf.

In particular, N can accept w even when there are choices that allow the machine to run forever, but rejection requires N to halt after only a finite number of transitions, no matter what choices it makes along the way. Just as for deterministic Turing machines, it is possible that N neither accepts nor rejects w.

Acceptance and rejection of *languages* are defined exactly as they are for deterministic machines. A non-deterministic Turing machine N ***accepts*** a language $L \subseteq \Sigma^*$ if M accepts all strings in L and nothing else; N ***rejects*** L if M rejects every string in L and nothing else; and finally, N ***decides*** L if M accepts L and rejects $\Sigma^* \setminus L$.

9.3 Time and Space Complexity

★★★

- Define "time" and "space".
- TIME$(f(n))$ is the class of languages that can be decided by a deterministic multi-tape Turing machine in $O(f(n))$ time.
- NTIME$(f(n))$ is the class of languages that can be decided by a nondeterministic multi-tape Turing machine in $O(f(n))$ time.
- SPACE$(f(n))$ is the class of languages that can be decided by deterministic multi-tape Turing machine in $O(f(n))$ space.
- NSPACE$(f(n))$ is the class of languages that can be decided by a nondeterministic multi-tape Turing machine in $O(f(n))$ space.
- Why multi-tape TMs? Because t steps on any k-tape Turing machine can be simulated in $O(t \log t)$ steps on a two-tape machine [Hennie and Stearns 1966, essentially using lazy counters and amortization], and in $O(t^2)$ steps on a single-tape machine [Hartmanis and Stearns 1965; realign multiple tracks at every simulation step]. Moreover, the latter quadratic bound is tight [Hennie 1965 (palindromes, via communication complexity)].

9.4 Deterministic Simulation via Dovetailing

Theorem 1. *For any nondeterministic Turing machine N, there is a deterministic Turing machine M that accepts exactly the same strings and N and rejects exactly the same strings as N. Moreover, if every computation path of N on input x halts after at most t steps, then M halts on input x after at most $O(t^2 r^{2t})$ steps, where r is the maximum size of any transition set in N.*

Proof: I'll describe a deterministic machine M that performs a breadth-first search of the computation tree of N. (The depth-first search performed by a standard recursive backtracking algorithm won't work here. If N's computation tree contains an infinite path, a depth-first search would get stuck in that path without exploring the rest of the tree.)

At the beginning of each simulation round, M's tape contains a string of the form

$$\square\square\cdots\square\bullet\bullet y_1q_1z_1\bullet y_2q_2z_2\bullet\cdots\bullet y_kq_kz_k\bullet\bullet$$

where each substring $y_iq_iz_i$ encodes a configuration $(q_i, y_iz_i, |y_i|)$ of some computation path of N, and $\bullet$ is a new symbol not in the tape alphabet of N. The machine M interprets this sequence of encoded configurations as a queue, with new configurations inserted on the right and old configurations removed from the left. The double-separators $\bullet\bullet$ uniquely identify the start and end of this queue; outside this queue, the tape is entirely blank.

Specifically, in each round, first M appends the encodings of all configurations than N can reach in one transition from the first encoded configuration $(q_1, y_1z_1, |y_1|)$; then M erases the first encoded configuration.

$$\begin{array}{c}
\cdots\square\square\underline{\bullet\bullet y_1q_1z_1}\bullet y_2q_2z_2\bullet\cdots\bullet y_rq_rz_r\underline{\bullet\bullet\square\square\cdots} \\
\Downarrow \qquad\qquad\qquad\qquad \Downarrow \\
\cdots\square\square\underline{\square\square\cdots\square\bullet\bullet}y_2q_2z_2\bullet\cdots\bullet y_kq_kz_k\underline{\bullet\tilde{y}_1\tilde{q}_1\tilde{z}_1\bullet\tilde{y}_2\tilde{q}_2\tilde{z}_2\bullet\cdots\bullet\tilde{y}_r\tilde{q}_r\tilde{z}_r\bullet\bullet\square\square\cdots}
\end{array}$$

Suppose each transition set $\delta_N(q,a)$ has size at most r. Then after simulating t steps of N, the tape string of M encoding $O(r^t)$ different configurations of N and therefore has length $L = O(tr^t)$ (not counting the initial blanks). If M begins each simulation phase by moving the initial configuration from the beginning to the end of the tape string, which takes $O(t^2r^t)$ time, the time for the rest of the the simulation phase is negligible. Altogether, simulating all r^t possibilities for the the tth step of N requires $O(t^2r^{2t})$ time. We conclude that M can simulate the first t steps of every computation path of N in $\boldsymbol{O(t^2r^{2t})}$ ***time***, as claimed. □

The running time of this simulation is dominated by the time spent reading from one end of the tape string and writing to the other. It is fairly easy to reduce the running time to $O(tr^t)$ by using either two tapes (a "read tape" containing N-configurations at time t and a "write tape" containing N-configurations at time $t+1$) or two independent heads on the same tape (one at each end of the queue).

9.5 Nondeterminism as Advice

Any nondeterministic Turing machine N can also be simulated by a *deterministic* machine M with *two* inputs: the user input string $w \in \Sigma^*$, and a so-called ***advice*** string $x \in \Omega^*$, where Ω is another finite alphabet. Only the first input string w is actually given by the user. At least for now, we assume that the advice string x is given on a separate read-only tape.

The deterministic machine M simulates N step-by-step, but whenever N has a choice of how to transition, M reads a new symbol from the advice string, and that symbol determines the choice. In fact, without loss of generality, we can assume that M reads a new symbol from the advice string and moves the advice-tape's head to the right on *every* transition. Thus, M's transition function has the form $\delta_M : Q \times \Gamma \times \Omega \to Q \times \Gamma \times \{-1, +1\}$, and we require that

$$\delta_N(q,a) = \{\delta_M(q,a,\omega) \mid \omega \in \Omega\}$$

For example, if N has a binary choice

$$\delta_N(\mathsf{branch}, ?) = \big\{(\mathsf{left}, \mathsf{L}, -1), (\mathsf{right}, \mathsf{R}, +1)\big\},$$

then M might determine this choice by defining

$$\delta_M(\mathsf{branch}, ?, 0) = (\mathsf{left}, \mathsf{L}, -1) \qquad \text{and} \qquad \delta_M(\mathsf{branch}, ?, 1) = (\mathsf{right}, \mathsf{R}, +1)$$

More generally, if every set $\delta_N(p, a)$ has size r, then we let $\Omega = \{1, 2, \ldots, r\}$ and define $\delta_M(q, a, i)$ to be the ith element of $\delta_N(q, a)$ in some canonical order.

Now observe that N accepts a string w if and only if M accepts the pair (w, x) for *some* string $x \in \Omega^*$, and N rejects w if and only if M rejects the pair (w, x) for *all* strings $x \in \Omega^*$.

The "advice" formulation of nondeterminism allows a different strategy for simulation by a standard deterministic Turing machine, which is often called ***dovetailing***. Consider all possible advice strings x, in increasing order of length; listing these advice strings is equivalent to repeatedly incrementing a base-r counter. For each advice string x, simulate M on input (w, x) for exactly $|x|$ transitions.

```
DOVETAIL_M(w):
  for t ← 1 to ∞
      done ← TRUE
      for all strings x ∈ Ω^t
          if M accepts (w, x) in at most t steps
              accept
          if M(w, x) does not halt in at most t steps
              done ← FALSE
      if done
          reject
```

The most straightforward Turing-machine implementation of this algorithm requires three tapes: A read-only input tape containing w, an advice tape containing x (which is also used as a timer for the simulation), and the work tape. This simulation requires $\boldsymbol{O(t\,r^t)}$ ***time*** to simulate all possibilities for t steps of the original non-deterministic machine N.

If we insist on using a standard Turing machine with a single tape and a single head, the simulation becomes slightly more complex, but (unlike our earlier queue-based strategy) not significantly slower. This standard machine S maintains a string of the form $\bullet w \bullet x \bullet z$, where z is the current work-tape string of M (or equivalently, of N), with marks (on a second track) indicating the current positions of the heads on M's work tape and M's advice tape. Simulating a single transition of M now requires $O(|x|)$ steps, because S needs to shuttle its single head between these two marks. Thus, S requires $\boldsymbol{O(t^2 r^t)}$ ***time*** to simulate all possibilities for t steps of the original non-deterministic machine N. This is significantly faster than the queue-based simulation, because we don't record (and therefore don't have to repeatedly scan over) intermediate configurations; recomputing everything from scratch is actually cheaper!

9.6 The Cook-Levin Theorem

★★★

Define SAT and CIRCUITSAT. Non-determinism is fundamentally different from other Turing machine extensions, in that it seems to provide an exponential speedup for some problems, just like NFAs can use exponentially fewer states than DFAs for the same language.

The Cook-Levin Theorem. *If SAT ∈ P, then P=NP.*

Proof: Let $L \subseteq \Sigma^*$ be an arbitrary language in NP, over some fixed alphabet Σ. There must be an integer k and Turing machine M that satisfies the following conditions:

- For all strings $w \in L$, there is at least one string $x \in \Sigma^*$ such that M accepts the string $w\Box x$.
- For all strings $w \notin L$ and $x \in \Sigma^*$, M rejects the string $w\Box x$.
- For all strings $w, x \in \Sigma^*$, M halts on input $w\Box x$ after at most $\max\{1, |w|^k\}$ steps.

Now suppose we are given a string $w \in \Sigma^*$. Let $n = |w|$ and let $N = \max\{1, |w|^k\}$. We construct a boolean formula Φ_w that is satisfiable if and only if $w \in L$, by following the execution of M on input $w\Box x$ for some unknown advice string x. Without loss of generality, we can assume that $|x| = N - n - 1$ (since we can extend any shorter string x with blanks.) Our formula Φ_w uses the following boolean variables for all symbols $a \in \Gamma$, all states $q \in Q$, and all integers $0 \le t \le N$ and $0 \le i \le N + 1$.

- $Q_{t,i,q}$ — M is in state q with its head at position i after t transitions.
- $T_{t,i,a}$ — The kth cell of M's work tape contains a after t transitions.

The formula Φ_w is the conjunction of the following constraints:

- **Boundaries:** To simplify later constraints, we include artificial boundary variables just past both ends of the tape:

$$\begin{aligned} Q_{t,i,q} &= Q_{t,N+1,q} = \textsc{False} && \text{for all } 0 \le t \le N \text{ and } q \in Q \\ T_{t,0,a} &= T_{t,N+1,a} = \textsc{False} && \text{for all } 0 \le t \le N \text{ and } a \in \Gamma \end{aligned}$$

- **Initialization:** We have the following values for variables with $t = 0$:

$$\begin{aligned} Q_{0,1,\mathsf{start}} &= \textsc{True} && \\ Q_{0,1,q} &= \textsc{False} && \text{for all } q \ne \mathsf{start} \\ H_{0,i,q} &= \textsc{False} && \text{for all } i \ne 1 \text{ and } q \in Q \\ T_{0,i,w_i} &= \textsc{True} && \text{for all } 1 \le i \le n \\ T_{0,i,a} &= \textsc{False} && \text{for all } 1 \le i \le n \text{ and } a \ne w_i \\ T_{0,n+1,\Box} &= \textsc{True} && \\ T_{0,n+1,a} &= \textsc{False} && \text{for all } a \ne \Box \end{aligned}$$

- **Uniqueness:** The variables $T_{0,i,a}$ with $n+2 \le i \le N$ represent the unknown advice string x; these are the "inputs" to Φ_w. We need some additional constraints ensure that for each i, *exactly one* of these variables is TRUE:

$$\left(\bigvee_{a \in \Gamma} T_{0,j,a}\right) \wedge \bigwedge_{a \ne b} \left(\overline{T_{0,j,a}} \vee \overline{T_{0,j,b}}\right)$$

- **Transitions:** For all $1 \le t \le N$ and $1 \le i \le N$, the following constraints simulate the transition from time $t-1$ to time t.

$$Q_{t,i,q} = \bigvee_{\delta(p,a)=(q,\cdot,+1)} \left(Q_{t-1,i-1,p} \wedge T_{t,i-1,a}\right) \vee \bigvee_{\delta(p,a)=(q,\cdot,-1)} \left(Q_{t-i,i+1,p} \wedge T_{t,i+1,a}\right)$$

$$T_{t,i,b} = \bigvee_{\delta(p,a)=(\cdot,b,\cdot)} \left(Q_{t-1,i,p} \wedge T_{t-1,i,a}\right) \vee \left(\bigwedge_{q \in Q} \overline{Q_{t-1,i,q}} \wedge T_{t-1,i,b}\right)$$

- **Output:** We have one final constraint that indicates acceptance.

$$z = \bigvee_{t=0}^{N} \bigvee_{i=1}^{N} Q_{t,i,\mathsf{accept}}$$

By definition, Φ_w is satisfiable if and only if some input values $T_{0,i,a}$, all constraints are satisfied, including acceptance. A straightforward induction argument implies that *even without the acceptance constraint*, any assignment of values to the unknown variables $T_{0,i,a}$ that satisfies the uniqueness constraints determines *unique* values for the other variables in Φ_w, which consistently describe the execution of M. To satisfy the acceptance constraint, this execution of M must lead to the accept state. Thus, Φ_w is satisfiable if and only if there is a string $x \in \Gamma^*$ such that M accepts the input $w\square x$. We conclude that Φ_w is satisfiable if and only if $w \in L$.

It remains only to argue that the reduction requires only polynomial time. For any input string w of length n, the formula Φ_w has $O(N^2)$ variables and $O(N^2)$ constraints (where the hidden constants depend on the machine M). Every constraint except acceptance has constant length, so altogether Φ_w has length $O(N^2)$. Moreover, we can construct Φ_w in $O(N^2) = O(n^{2k})$ time.

In conclusion, if we could decide SAT for formulas of size M in $O(M^c)$ time, then we could decide membership in L in $O(n^{2kc})$ time, which implies that $L \in \mathrm{P}$. $\square$

Exercises

1. Prove that the following problem is NP-hard, *without* using the Cook-Levin Theorem. Given a string $\langle M, w\rangle$ that encodes a non-deterministic Turing machine M and a string w, does M accept w in at most $|w|$ transitions?

★★★ More excerises!

© Copyright 2016 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

The first lot fell to Jehoiarib, the second to Jedaiah, the third to Harim, the fourth to Seorim, the fifth to Malkijah, the sixth to Mijamin, the seventh to Hakkoz, the eighth to Abijah, the ninth to Jeshua, the tenth to Shekaniah, the eleventh to Eliashib, the twelfth to Jakim, the thirteenth to Huppah, the fourteenth to Jeshebeab, the fifteenth to Bilgah, the sixteenth to Immer, the seventeenth to Hezir, the eighteenth to Happizzez, the nineteenth to Pethahiah, the twentieth to Jehezkel, the twenty-first to Jakin, the twenty-second to Gamul, the twenty-third to Delaiah, and the twenty-fourth to Maaziah.

This was their appointed order of ministering when they entered the temple of the LORD, according to the regulations prescribed for them by their ancestor Aaron, as the LORD, the God of Israel, had commanded him.

— 1 Chronicles 24:7–19 (New International Version)

The ring worm is not ringed, nor is it worm. It is a fungus.
The puff adder is not a puff, nor can it add. It is a snake.
The funny bone is not funny, nor is it a bone. It is a nerve.
The fishstick is not a fish, nor is it a stick. It is a fungus.

— Matt Groening, "Life in Hell" (1986)

When Pierre-Simon Laplace described probability theory as "good sense reduced to a calculus," he intended to disparage neither good sense nor probability theory thereby.

— Lorraine Daston, "Enlightenment Calculations" (1994)

1 Discrete Probability

Before I start discussing randomized *algorithms* at all, I need to give a quick formal overview of the relatively small subset of probability theory that we will actually use. The first two sections of this note are deliberately written more as a review or reference than an introduction, although they do include a few illustrative (and hopefully helpful) examples.

1.1 Discrete Probability Spaces

A ***discrete***[1] ***probability space*** $(\Omega, \Pr)$ consists of a non-empty countable set Ω, called the ***sample space***, together with a ***probability mass function*** $\Pr\colon \Omega \to \mathbb{R}$ such that

$$\Pr[\omega] \geq 0 \text{ for all } \omega \in \Omega \qquad \text{and} \qquad \sum_{\omega \in \Omega} \Pr[\omega] = 1.$$

The latter condition implies that $\Pr[\omega] \leq 1$ for all $\omega \in \Omega$. I don't know why the probability function is written with brackets instead of parentheses, but that's the standard;[2] just go with it. Here are a few simple examples:

- A fair coin: $\Omega = \{\mathsf{heads}, \mathsf{tails}\}$ and $\Pr[\mathsf{heads}] = \Pr[\mathsf{tails}] = 1/2$.
- A fair six-sided die: $\Omega = \{1, 2, 3, 4, 5, 6\}$ and $\Pr[\omega] = 1/6$ for all $\omega \in \Omega$.
- A strangely loaded six-sided die: $\Omega = \{1, 2, 3, 4, 5, 6\}$ with $\Pr[\omega] = \omega/21$ for all $\omega \in \Omega$. (For example, $\Pr[4] = 4/21$.)

[1] Correctly defining *continuous* (or otherwise uncountable) probability spaces and continuous random variables requires considerably more care and subtlety than the discrete definitions given here. There is *no* well-defined probability measure satisfying the discrete axioms when Ω is, for instance, an interval on the real line. This way lies the Banach-Tarski paradox.

[2] Or more honestly: one of many standards

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

- Bart's rock-paper-scissors strategy: $\Omega = \{\mathsf{rock}, \mathsf{paper}, \mathsf{scissors}\}$ and $\Pr[\mathsf{rock}] = 1$ and $\Pr[\mathsf{paper}] = \Pr[\mathsf{scissors}] = 0$.

Other common examples of countable sample spaces include the 52 cards in a standard deck, the 52! permutations of the cards in a standard deck, the natural numbers, the integers, the rationals, the set of all (finite) bit strings, the set of all (finite) rooted trees, the set of all (finite) graphs, and the set of all (finite) execution traces of an algorithm.

The precise choice of probability space is rarely important; we can usually implicitly define Ω to be the set of all possible tuples of values, one for each random variable under discussion.

1.1.1 Events and Probability

Subsets of Ω are usually called ***events***, and individual elements of Ω are usually called ***sample points*** or ***elementary events*** or ***atoms***. However, it is often useful to think of the elements of Ω as possible *states* of a system or *outcomes* of an experiment, and subsets of Ω as *conditions* that some states/outcomes satisfy and others don't.

The probability of an event A, denoted $\mathbf{Pr}[A]$, is defined as the sum of the probabilities of its constituent sample points:

$$\Pr[A] := \sum_{\omega \in A} \Pr[\omega]$$

In particular, we have $\Pr[\varnothing] = 0$ and $\Pr[\Omega] = 1$. Here we are extending (or overloading) the function $\Pr\colon \Omega \to [0,1]$ on atoms to a function $\Pr\colon 2^{\Omega} \to [0,1]$ on events.

For example, suppose we roll two fair dice, one red and the other blue. The underlying probability space consists of the sample space $\Omega = \{1,2,3,4,5,6\} \times \{1,2,3,4,5,6\}$ and the probabilities $\Pr[\omega] = 1/36$ for all $\omega \in \Omega$.

- The probability of rolling two 5s is $\Pr[\{(5,5)\}] = \Pr[(5,5)] = 1/36$.
- The probability of rolling a total of 6 is

$$\Pr\Big[\{(1,5),(2,4),(3,3),(4,2),(5,1)\}\Big] = \frac{5}{36}.$$

- The probability that the red die shows a 5 is

$$\Pr\Big[\{(5,1),(5,2),(5,3),(5,4),(5,5),(5,6)\}\Big] = \frac{1}{6}.$$

- The probability that at least one die shows a 5 is

$$\Pr\Big[\{(1,5),(2,5),(3,5),(4,5),(5,1),(5,2),(5,3),(5,4),(5,5),(5,6),(6,5)\}\Big] = \frac{11}{36}.$$

- The probability that the red die shows a smaller number than the blue die is

$$\begin{aligned}\Pr\Big[\{&(1,2),(1,3),(1,4),(1,5),(1,6),\\ &(2,3),(2,4),(2,5),(2,6),(3,4),\\ &(3,5),(3,6),(4,5),(4,6),(5,6)\}\Big] = \frac{5}{12}.\end{aligned}$$

1.1.2 Combining Events

Because they are formally just sets, events can be combined using arbitrary set operations. However, in keeping with the intuition that events are *conditions*, these operations are usually written using Boolean logic notation $\wedge, \vee, \neg$ and vocabulary ("and, or, not") instead of the equivalent set notation $\cap, \cup, \overline{\cdot}$ and vocabulary ("intersection, union, complement"). For example, consider our earlier experiment rolling two fair six-sided dice, one red and the other blue.

$$\begin{aligned}
\Pr[\text{red } 5] &= 1/6 \\
\Pr[\text{two } 5\text{s}] &= \Pr[\text{red } 5 \wedge \text{blue } 5] = 1/36 \\
\Pr[\text{at least one } 5] &= \Pr[\text{red } 5 \vee \text{blue } 5] = 11/36 \\
\Pr[\text{at most one } 5] &= \Pr[\neg(\text{two } 5\text{s})] = 1 - \Pr[\text{two } 5\text{s}] = 35/36 \\
\Pr[\text{no } 5\text{s}] &= \Pr[\neg(\text{at least one } 5)] \\
&= 1 - \Pr[\text{at least one } 5] = 25/36 \\
\Pr[\text{exactly one } 5] &= \Pr[\text{at least one } 5 \wedge \text{at most one } 5] \\
&= \Pr[\text{red } 5 \oplus \text{blue } 5] = 5/18 \\
\Pr[\text{blue } 5 \Rightarrow \text{red } 5] &= \Pr[\neg(\text{blue } 5) \vee \text{red } 5] = 31/36
\end{aligned}$$

(As usual, $p \Rightarrow q$ is just shorthand for $\neg p \vee q$; implication does *not* indicate causality!)

For any two events A and B with $\Pr[B] > 0$, the ***conditional probability of A given B*** is defined as

$$\mathbf{Pr}[\boldsymbol{A} \mid \boldsymbol{B}] := \frac{\Pr[A \wedge B]}{\Pr[B]}.$$

For example, in our earlier red-blue dice experiment:

$$\begin{aligned}
\Pr[\text{blue } 5 \mid \text{red } 5] &= \Pr[\text{two } 5\text{s} \mid \text{red } 5] = 1/6 \\
\Pr[\text{at most one } 5 \mid \text{red } 5] &= \Pr[\text{exactly one } 5 \mid \text{red } 5] \\
&= \Pr[\neg(\text{blue } 5) \mid \text{red } 5] = 5/6 \\
\Pr[\text{at least one } 5 \mid \text{at most one } 5] &= 2/7 \\
\Pr[\text{at most one } 5 \mid \text{at least one } 5] &= 10/11 \\
\Pr[\text{red } 5 \mid \text{at least one } 5] &= 6/11 \\
\Pr[\text{red } 5 \mid \text{at most one } 5] &= 1/7 \\
\Pr[\text{blue } 5 \mid \text{blue } 5 \Rightarrow \text{red } 5] &= 1/31 \\
\Pr[\text{red } 5 \mid \text{blue } 5 \Rightarrow \text{red } 5] &= 6/31 \\
\Pr[\text{blue } 5 \Rightarrow \text{red } 5 \mid \text{blue } 5] &= 1/6 \\
\Pr[\text{blue } 5 \Rightarrow \text{red } 5 \mid \text{red } 5] &= 1 \\
\Pr[\text{blue } 5 \Rightarrow \text{red } 5 \mid \text{red } 5 \Rightarrow \text{blue } 5] &= 26/31
\end{aligned}$$

Two events A and B are ***disjoint*** if they are disjoint as sets, meaning $A \cap B = \varnothing$. For example, in our two-dice experiment, the events "red 5" and "blue 5" and "total 5" are pairwise disjoint. Note that it is possible for $\Pr[A \wedge B] = 0$ even when the events A and B are not disjoint; consider the events "Bart plays paper" and "Bart does not play rock".

Two events A and B are ***independent*** if and only if $\Pr[A \wedge B] = \Pr[A] \cdot \Pr[B]$. For example, in our two-dice experiment, the events "red 5" and "blue 5" are independent, but "red 5" and "total 5" are not.

More generally, a countable set of events $\{A_i \mid i \in I\}$ is ***fully*** or ***mutually independent*** if and only if $\Pr[\bigwedge_{i\in I} A_i] = \prod_{i\in I} A_i$. A set of events is ***k-wise independent*** if every subset of k events is fully independent, and ***pairwise independent*** if every pair of events in the set is independent. For example, in our two-dice experiment, the events "red 5" and "blue 5" and "total 7" are pairwise independent, but not mutually independent.

1.1.3 Identities and Inequalities

Fix n arbitrary events $A_1, A_2, \ldots, A_n$ from some sample space Ω. The following observations follow immediately from similar observations about sets.

- ***Union bound:*** For *any* events $A_1, A_2, \ldots A_n$, the definition of probability implies that

$$\Pr\left[\bigvee_{i=1}^{n} A_i\right] \;\le\; \sum_{i=1}^{n} \Pr[A_i].$$

  The expression on the right counts each atom in the union of the events exactly once; the left summation counts each atom once for each event A_i that contains it.

- ***Disjoint union:*** If the events $A_1, A_2, \ldots A_n$ are pairwise disjoint, meaning $A_i \cap A_j = \varnothing$ for all $i \neq j$, the union bound becomes an equation:

$$\Pr\left[\bigvee_{i=1}^{n} A_i\right] \;=\; \sum_{i=1}^{n} \Pr[A_i].$$

- ***The principle of inclusion-exclusion*** describes a simple relationship between probabilities of unions (disjunctions) and intersections (conjunctions) of arbitrary events:

$$\Pr[A \vee B] \;=\; \Pr[A] + \Pr[B] - \Pr[A \wedge B]$$

  This principle follows directly from elementary boolean algebra and the disjoint union bound:

$$\begin{aligned}
\Pr[A \vee B] + \Pr[A \wedge B] &= \Pr[(A \wedge B) \vee (A \wedge \overline{B}) \vee (\overline{A} \wedge B)] + \Pr[A \wedge B] \\
&= \big(\Pr[A \wedge B] + \Pr[A \wedge \overline{B}] + \Pr[\overline{A} \wedge B]\big) + \Pr[A \wedge B] \\
&= \big(\Pr[A \wedge B] + \Pr[A \wedge \overline{B}]\big) + \big(\Pr[\overline{A} \wedge B] + \Pr[A \wedge B]\big) \\
&= \Pr[A] + \Pr[B]
\end{aligned}$$

  Inclusion-exclusion generalizes inductively any finite number of events as follows:

$$\Pr\left[\bigvee_{i=1}^{n} A_i\right] \;=\; 1 - \sum_{I \subseteq [1..n]} (-1)^{|I|} \Pr\left[\bigwedge_{i\in I} A_i\right]$$

- ***Independent union:*** For any pair A and B of independent events, we have

$$\begin{aligned}
\Pr[A \vee B] &= \Pr[A] + \Pr[B] - \Pr[A \wedge B] && \text{[inclusion-exclusion]} \\
&= \Pr[A] + \Pr[B] - \Pr[A]\Pr[B] && \text{[independence]} \\
&= 1 - (1 - \Pr[A])(1 - \Pr[B]).
\end{aligned}$$

  More generally, if the events $A_1, A_2, \ldots A_n$ are *mutually* independent, then

$$\Pr\left[\bigvee_{i=1}^{n} A_i\right] = 1 - \prod_{i=1}^{n} (1 - \Pr[A_i]).$$

- ***Bayes' Theorem:*** If events A and B both have non-zero probability, the definition of conditional probability immediately implies

$$\frac{\Pr[A \mid B]}{\Pr[A]} = \Pr[A \wedge B] = \frac{\Pr[B \mid A]}{\Pr[B]}.$$

  and therefore

$$\Pr[A \mid B] \cdot \Pr[B] = \Pr[B \mid A] \cdot \Pr[A].$$

1.2 Random Variables

Formally, a ***random variable X*** is a function from a sample space Ω (with an associated probability measure) to some other *value set*. For example, the identity function on Ω is a random variable, as is the function that maps everything in Ω to Queen Elizabeth II, or any function mapping Ω to the real numbers. Random variables are almost universally denoted by upper-case letters.

A random variable is not random, nor is it a variable.

The value space of a random variable is commonly described either by an adjective preceding the phrase "random variable" or a noun replacing the word "variable". For example:

- A function from Ω to $\mathbb{Z}$ is called an *integer random variable* or a *random integer*.
- A function from Ω to $\mathbb{R}$ is called an *real random variable* or a *random real number*.
- A function from Ω to $\{0, 1\}$ is called an *indicator random variable* or a *random bit*.

Since every integer is a real number, every integer random variable is also a real random variable; similarly, every random bit is also a random real number. Not all random variables are numerical; for example:

- A *random graph* is function from some sample space Ω to the set of all graphs.
- A *random point in the plane* is a function from some sample space Ω to $\mathbb{R}^2$.
- The function mapping every element of Ω to Queen Elizabeth II could be called a *random Queen of England*, or a *random monarch*, or (if certain unlikely conspiracy theories are taken seriously) a *random shape-shifting alien reptile*.

1.2.1 It is a fungus.

Although random variables are formally not variables at all, we typically describe and manipulate them *as if they were* variables representing unknown elements of their value sets, without referring to any particular sample space.

In particular, we can apply arbitrary functions to random variables by composition. Fir any random variable $X : \Omega \to V$ and any function $f : V \to V'$, the function $f(X) := f \circ X$ is a random variable over the value set V'. In particular, if X is a real random variable and α is any real number, then $X + \alpha$ and $\alpha \cdot X$ are also real random variables. More generally, if X and X' are random variables with value sets V and V', then for any function $f : V \times V' \to V''$, the function $f(X, X')$ is a random variable over V'', formally defined as

$$f(X, X')(\omega) := f(X(\omega), X'(\omega)).$$

These definitions extend in the obvious way to functions with an arbitrary number of arguments.

If ϕ is a boolean function or predicate over the value set of X, we implicitly identify the random variable $\phi(X)$ with the event $\{\omega \in \Omega \mid \phi(X(\omega))\}$. For example, if X is an integer random variable, then

$$\Pr[X = x] \;:=\; \Pr\big[\{\omega \mid X(\omega) = x\}\big]$$

and

$$\Pr[X \le x] \;:=\; \Pr\big[\{\omega \mid X(\omega) \le x\}\big]$$

and

$$\Pr[X \text{ is prime}] \;:=\; \Pr\big[\{\omega \mid X(\omega) \text{ is prime}\}\big].$$

In particular, we typically identify the boolean values **True** and **False** with the events Ω and $\varnothing$, respectively. Predicates with more than one random variable are handled similarly; for example,

$$\Pr[X = Y] \;:=\; \Pr\big[\{\omega \mid X(\omega) = Y(\omega)\}\big].$$

1.2.2 Expectation

For any real (or complex or vector) random variable X, the ***expectation of X*** is defined as

$$\mathbf{E}[X] := \sum_x x \cdot \Pr[X = x].$$

This sum is always well-defined, because the set $\{x \mid \Pr[X = x] \neq 0\} \subseteq \Omega$ is countable. For *integer* random variables, the following definition is equivalent:

$$\begin{aligned} \mathrm{E}[X] &= \sum_{x \ge 0} \Pr[X \ge x] - \sum_{x \le 0} \Pr[X \le x] \\ &= \sum_{x \ge 0} \big(\Pr[X \ge x] - \Pr[X \le -x]\big). \end{aligned}$$

If moreover A is an arbitrary event with non-zero probability, then the ***conditional expectation of X given A*** is defined as

$$\mathbf{E}[X \mid A] := \sum_x x \cdot \Pr[X = x \mid A] = \sum_x \frac{x \cdot \Pr[X = x \wedge A]}{\Pr[A]}$$

For any event A with $0 < \Pr[A] < 1$, we immediately have

$$\mathrm{E}[X] \;=\; \mathrm{E}[X \mid A] \cdot \Pr[A] \;+\; \mathrm{E}[X \mid \neg A] \cdot \Pr[\neg A].$$

In particular, for any random variables X and Y, we have

$$\mathrm{E}[X] = \sum_y \mathrm{E}[X \mid Y = y] \cdot \Pr[Y = y].$$

Two random variables X and Y are ***independent*** if, for all x and y, the events $X = x$ and $Y = y$ are independent. If X and Y are independent real random variables, then $\mathrm{E}[X \cdot Y] = \mathrm{E}[X] \cdot \mathrm{E}[Y]$. (However, this equation does *not* imply that X and Y are independent.) We can extend the notions of full, k-wise, and pairwise independence from events to random variables in similar fashion. In particular, if $X_1, X_2, \ldots, x_n$ are *fully independent* real random variables, then

$$\mathrm{E}\left[\prod_{i=1}^{n} X_i\right] = \prod_{i=1}^{n} \mathrm{E}[X_i].$$

Linearity of expectation refers to the following important fact: The expectation of any weighted sum of random variables is equal to the weighted sum of the expectations of those variables. More formally, for any real random variables $X_1, X_2, \ldots, X_n$ and any real coefficients $\alpha_1, \alpha_2, \ldots, \alpha_n$,

$$\mathrm{E}\left[\sum_{i=1}^{n}(\alpha_i \cdot X_i)\right] = \sum_{i=1}^{n}\big(\alpha_i \cdot \mathrm{E}[X_i]\big).$$

Linearity of expectation does *not* require the variables to be independent.

1.2.3 Examples

Consider once again our experiment with two standard fair six-sided dice, one red and the other blue. We define several random variables:

- R is the value (on the top face) of the red die.
- B is the value (on the top face) of the blue die.
- $S = R + B$ is the total value (on the top faces) of both dice.
- $\mathcal{R} = 7 - R$ is the value on the *bottom* face of the red die.

The variables R and B are independent, as are the variables $\mathcal{R}$ and B, but no other pair of these variables is independent.

$$\begin{aligned}
\mathrm{E}[R] = \mathrm{E}[B] = \mathrm{E}[\mathcal{R}] &= \frac{1+2+3+4+5+6}{6} = \frac{7}{2} \\
\mathrm{E}[R+B] &= \mathrm{E}[R] + \mathrm{E}[B] = 7 && \text{[linearity]} \\
\mathrm{E}[R+\mathcal{R}] &= 7 && \text{[trivial distribution]} \\
\mathrm{E}[R+B+\mathcal{R}] &= \mathrm{E}[R] + \mathrm{E}[B] + \mathrm{E}[\mathcal{R}] = \frac{21}{2} && \text{[linearity]} \\
\mathrm{E}[R \cdot B] &= \mathrm{E}[R] \cdot \mathrm{E}[B] = \frac{49}{4} && \text{[independence]} \\
\mathrm{E}[R \cdot \mathcal{R}] &= \frac{1\cdot 6 + 2\cdot 5 + 3\cdot 4}{3} = \frac{28}{3} \\
\mathrm{E}[R^2] &= \frac{1^2+2^2+3^2+4^2+5^2+6^2}{6} = \frac{91}{6} \\
\mathrm{E}[(R+B)^2] &= \mathrm{E}[R^2] + 2\,\mathrm{E}[RB] + \mathrm{E}[B^2] = \frac{329}{6} && \text{[linearity]} \\
\mathrm{E}[R+B \mid R=6] &= \mathrm{E}[R \mid R=6] + \mathrm{E}[B \mid R=6] && \text{[linearity]} \\
&= 6 + \mathrm{E}[B] = 19/2 && \text{[independence]}
\end{aligned}$$

$$\begin{aligned}
\mathrm{E}[R \mid R+B=6] &= \frac{1+2+3+4+5}{5} = 3 \\
\mathrm{E}[R^2 \mid R+B=6] &= \frac{1^2+2^2+3^2+4^2+5^2}{5} = 11 \\
\mathrm{E}[R+B \mid R\cdot B=6] &= \frac{(1+6)+(2+3)}{2} = 6 \\
\mathrm{E}[R\cdot B \mid R+B=6] &= \frac{(1\cdot 5)+(2\cdot 4)+(3\cdot 3)+(4\cdot 2)+(5\cdot 1)}{5} = 7
\end{aligned}$$

1.3 Common Probability Distributions

A ***probability distribution*** assigns a probability to each possible value of a random variable. More formally, $X : \Omega \to V$ is a random variable over some probability space $(\Omega, \Pr)$, the probability distribution of X is the function $P : V \to [0,1]$ such that

$$P(x) = \Pr[X = x] = \sum \{\Pr(\omega) \mid X(\omega) = x\}.$$

The ***support*** of a probability distribution is the set of values with non-zero probability; this is a subset of the value set V. The following table summarizes several of the most useful discrete probability distributions.

name	intuition	parameters	support	$\Pr[X = x]$	$\mathrm{E}[X]$
trivial	Good ol' Rock, nothing beats that!	—	singleton set $\{a\}$	1	a
uniform	fair die roll	—	finite set $S \neq \varnothing$	$\frac{1}{\lvert S\rvert}$	$\frac{\sum S}{\lvert S\rvert}$
Bernoulli	biased coin flip	$0 \le p \le 1$	$\{0,1\}$	$\begin{cases} p & \text{if } x = 1 \\ 1-p & \text{if } x = 0 \end{cases}$	p
binomial	n biased coin flips	$0 \le p \le 1$ $n \ge 0$	$[0..n]$	$\binom{n}{x} p^x (1-p)^{n-x}$	np
geometric	#tails before first head	$0 < p \le 1$	$\mathbb{N}$	$(1-p)^x p$	$\frac{1-p}{p}$
negative binomial	#tails before nth head	$0 < p \le 1$ $n \ge 0$	$\mathbb{N}$	$\binom{n+x-1}{x}(1-p)^x p^n$	$\frac{n(1-p)}{p}$

Figure 1.1. Common discrete probability distributions.

- The ***trivial*** distribution describes the outcome of a "random" experiment that *always* has the same result. A trivially distributed random variable takes some fixed value with probability 1. Yes, this is still randomness.

- The ***uniform*** distribution assigns the same probability to every element of some finite non-empty set S. For example, if the random variable X is uniformly distributed over the integer range $[1..n]$, then $\Pr[X = x] = 1/n$ for each integer x between 1 and n, and $\mathrm{E}[X] = (n+1)/2$. This distribution models (idealized) fair coin flips, die rolls, and lotteries; consequently, this is what many people *incorrectly* think of as the definition of "random".

- The ***Bernoulli*** distribution models a random experiment (called a ***Bernoulli trial***) with two possible outcomes: ***success*** and ***failure***. The probability of success, usually denoted p, is a parameter of the distribution; the failure probability is often denoted $q = 1 - p$. Success and failure are usually represented by the *values* 1 and 0. Thus, every indicator random variable has a Bernoulli distribution, and its expected value is equal to its success probability:

  $$\mathrm{E}[X] \;=\; \sum_x x \cdot \Pr[X = x] \;=\; 0 \cdot \Pr[X = 0] + 1 \cdot \Pr[X = 1] \;=\; \Pr[X = 1] \;=\; p.$$

  The special case $p = 1/2$ is a uniform distribution with two values: a fair coin flip. The special cases $p = 0$ and $p = 1$ are trivial distributions.

- The ***geometric*** distribution describes the number of independent Bernoulli trials (all with the same success probability p) before the first success. If X is a geometrically distributed random variable, then $X = x$ if and only if the first x Bernoulli trials fail and the $(x+1)$th trial succeeds:

$$\Pr[X = x] \;=\; \left(\prod_{i=1}^{x} \Pr[i\text{th trial fails}]\right) \cdot \Pr[(x+1)\text{th trial succeeds}] \;=\; (1-p)^x p.$$

- The ***binomial*** distribution is the sum of n independent Bernoulli distributions, all with the same probability p of success. If X is a binomially distributed random variable, then $X = x$ if and only if x of the n trials succeed and $n - x$ fail:

$$\Pr[X = x] = \binom{n}{x} p^x (1-p)^{n-x}.$$

  If $n = 1$, this is just the Bernoulli distribution.

- The ***negative binomial*** distribution describes the number of independent Bernoulli trials (all with the same success probability p) that fail before the nth successful trial. If X is a negative-binomially distributed random variable, then $X = x$ if and only if exactly x of the first $n + x - 1$ trials are failures, and the $(n + x)$th trial is a success:

$$\Pr[X = x] = \binom{n+x-1}{x} (1-p)^x p^n$$

  If $n = 1$, this is just the geometric distribution.

1.4 Coin Flips

Suppose you are given a coin and you are asked generate a uniform random bit. We distinguish between two types of coins: A coin is *fair* if the probability of heads (1) and the probability of tails (0) are both exactly 1/2. A coin where one side is more likely than the other is said to be *biased*. Actual physical coins are reasonable approximations of abstract fair coins for most purposes, at least if they're flipped high into the air and allowed to bounce.[3] Physical coins can be biased by bending them.

1.4.1 Removing Unknown Bias

In 1951, John von Neumann discovered the following simple technique to simulate fair coin flips using an arbitrarily biased coin, *even without knowing the bias*. Flip the biased coin twice. If the two flips yield different results, return the first result; otherwise, repeat the experiment from scratch.

```
VonNeumannCoin():
  x ← BiasedCoin()
  y ← BiasedCoin()
  if x ≠ y
      return x
  else
      return VonNeumannCoin()
```

[3] Persi Diaconis, Susan Holmes, and Richard Montgomery published a thorough analysis of physical coin-flipping in 2007, which concluded (among other things) that coins that are flipped vigorously and then caught come up in the same state they started about 51% of the time. The small amount of bias arises because flipping coins tend to precess as they rotate. Letting the coin bounce instead of catching them appears to remove the bias from precession.

This is weird sort of algorithm, isn't it? There is *no* upper bound on the worst-case running time; in principle, the algorithm could run *forever*, because the biased coin always just happens to flip heads. Nevertheless, I claim that this is a useful algorithm for generating fair random bits. We need two technical assumptions:

(1) The biased coin always flips heads with the same fixed (but unknown!) probability p. To simplify notation, we let $q = 1 - p$ denote the probability of flipping tails. For example, a fair coin would have $p = q = 1/2$.

(2) All flips of the biased coin are mutually independent.

First, I claim that *if* the algorithm halts, then it returns a uniformly distributed random bit. Because the two biased coin flips are independent, we have

$$\Pr[x = 0 \wedge y = 1] = \Pr[x = 1 \wedge y = 0] = pq,$$

and therefore (assuming $pq > 0$)

$$\Pr[x = 0 \wedge y = 1 \mid x \neq y] = \Pr[x = 1 \wedge y = 0 \mid x \neq y] = \frac{pq}{2pq} = \frac{1}{2}.$$

Because the biased coin flips are mutually independent, the same analysis applies without modification to every recursive call to VonNeumannCoin. Thus, if *any* recursive call returns a bit, that bit is uniformly distributed.

Now let T denote the actual running time of this algorithm; T is a random variable that depends on the biased coin flips.[4] We can compute the expected running time $\mathrm{E}[T]$ by considering the conditional expectations in two cases: The first two flips are either different or equal.

$$\mathrm{E}[T] \;=\; \mathrm{E}[T \mid x \neq y] \cdot \Pr[x \neq y] \;+\; \mathrm{E}[T \mid x = y] \cdot \Pr[x = y]$$

Because the two biased coin flips are independent, we have

$$\Pr[x \neq y] = \Pr[x = 0 \wedge y = 1] + \Pr[x = 1 \wedge y = 0] = 2pq$$

and therefore $\Pr[x = y] = 1 - 2pq$. If the first two coin flips are different, the algorithm ends after two flips; thus, $\mathrm{E}[T \mid x \neq y] = 2$. Finally, if the first two coin flips are the same, the experiment starts over from scratch after the first two flips, so $\mathrm{E}[T \mid x = y] = 2 + \mathrm{E}[T]$. Putting all the pieces together, we have

$$\mathrm{E}[T] = 2 \cdot 2pq + (2 + \mathrm{E}[T]) \cdot (1 - 2pq).$$

Solving this equation for $\mathrm{E}[T]$ yields the solution $\mathbf{E[T] = 1/pq}$. For example, if $p = 1/3$, the expected number of coin flips is $9/2$.

Alternatively, we can think of VonNeumannCoin as performing an experiment with a different biased coin, which returns a result ("heads") with probability $2pq$. Thus, the expected number of *unsuccessful* iterations ("tails" before the first "head") is a geometric random variable with expectation $1/2pq - 1$, and thus the expected number of iterations is $1/2pq$. Because each iteration flips two coins, the expected number of coin flips is $1/pq$.

[4] Normally we use $T(n)$ to denote the worst-case running time of an algorithm as a function of some input parameter n, but this algorithm has no input parameters!

1.4.2 Removing Known Bias

But what if we *know* that $p = 1/3$? In that case, the following algorithm simulates a fair coin with fewer biased flips (on average):

```
FairCoin():
  x ← BiasedCoin(1/3)
  y ← BiasedCoin(1/3)
  if x ≠ y                ⟨⟨probability 4/9⟩⟩
      return 0
  else if x = y = 1       ⟨⟨probability 4/9⟩⟩
      return 1
  else                    ⟨⟨probability 1/9⟩⟩
      return FairCoin()
```

The algorithm returns a fair coin because

$$\Pr[x \neq y] = \frac{4}{9} = \Pr[x = y = 0].$$

The expected number of flips satisfies the equation

$$\mathrm{E}[T] = 2 + \frac{1}{9}\mathrm{E}[T],$$

which implies that $\mathrm{E}[T] = 9/4$, a factor of 2 better than von Neumann's algorithm.

1.5 Pokémon Collecting

A distressingly large fraction of my daughters' friends are obsessed with Pokémon—not the cartoon or the mobile game, but the collectible card game. The Pokémon Company sells small packets, each containing half a dozen cards, each describing a different Pokémon character. The cards can be used to play a complex turn-based combat game; the more cards a player owns, the more likely they are to win. So players are strongly motivated to collect as many cards, and in particular, as many *different* cards, as possible. Pokémon reinforces this message with their oh-so-subtle theme song "Gotta Catch 'Em All!" Unfortunately, the packets are opaque; the only way to find out which cards are in a pack is to buy the pack and tear it open.[5]

Let's consider the following oversimplified model of the Pokémon-collection process. In each trial, we purchase one Pokémon card, chosen independently and uniformly at random from the set of n possible card types. We repeat these trials until we have purchased at least one of each type of card. This problem was first considered by the French mathematician Abraham de Moivre in his seminal 1712 treatise *De Necessitate ut Caperent Omnium Eorum*.[6]

1.5.1 After n Trials

How many different types of Pokémon do we actually own after we buy n cards? Obviously in the worst case, we might just have n copies of one Pokémon,[7] but that's not particularly likely. To

[5]See also: cigarette cards, Dixie cups, baseball cards, Pez dispensers, Beanie Babies, Iwako puzzle erasers, Shopkins, and Guys Under Your Supervision.

[6]The actual title was *De Mensura Sortis seu; de Probabilitate Eventuum in Ludis a Casu Fortuito Pendentibus*, which means "On the measurement of chance, or on the probability of events in games depending on fortuitous chance".

[7]"Dave Guy!"

analyze the *expected* number of types that we own, we introduce an incredibly useful technique that lets us exploit linearity of expectation: decomposing more complex random variables into ***sums of indicator variables***.

For each index i, define an indicator variable $X_i = [\text{we own Pokémon } i]$ so that $X = \sum_i X_i$ is the number of cards we own. Linearity of expectation implies

$$\mathrm{E}[X] = \sum_{i=1}^{n} \mathrm{E}[X_i] = \sum_{i=1}^{n} \Pr[X_i = 1].$$

The probability that we *don't* own card i is $(1-1/n)^n \approx 1/e$, so

$$\mathrm{E}[X] = \sum_{i=1}^{n} \mathrm{E}[X_i] \approx \sum_{i=1}^{n} (1-1/e) = (1-1/e)n \approx 0.63212n.$$

In other words, after buying n cards, we expect to own a bit less than $2/3$ of the Pokémon.

Similar calculations implies that we expect to own about 86% of the Pokémon after buying $2n$ cards, about 95% after buying $3n$ cards, about 98% after buying $4n$ cards, and so on.

1.5.2 Gotta Catch 'em *All*

So how many Pokémon packs do we need to buy to catch 'em *all*? Obviously in the worst case, we might *never* have a complete collection[8], but assuming each type of card has some non-zero probability of being in each pack, the *expected* number of packs we need to buy is finite. Let $T(n)$ denote the number of packs (or "time") required to collect all n Pokémon. For purposes of analysis, we partition the random purchasing algorithm into n *phases*, where the ith phase ends just after we see the ith distinct card type. Let us write

$$T(n) = \sum_i T_i(n)$$

where $T_i(n)$ is the number of cards bought during the ith phase. Linearity of expectation implies

$$\mathrm{E}[T(n)] = \sum_i \mathrm{E}[T_i(n)].$$

We can think of each card purchase as a biased coin flip, where "heads" means "got a new Pokémon" and "tails" means "got a Pokémon we already own". For each index i, the probability of heads (that is, the probability of a single purchase being a new Pokémon) is exactly $p = (n-i+1)/n$: each of the n Pokémon is equally likely, and there are $n-i+1$ Pokémon that we don't already own. By our earlier analysis, the expected number of flips until the first head is $\mathrm{E}[T_i] = 1/p = n/(n-i+1)$. We conclude that

$$\mathrm{E}[T(n)] \;=\; \sum_i \mathrm{E}[T_i(n)] \;=\; \sum_{i=1}^{n} \frac{n}{n-i+1} \;=\; \sum_{j=1}^{n} \frac{n}{j} \;=\; nH_n.$$

Here $\boldsymbol{H_n}$ denotes the nth ***harmonic number***, defined recursively as

$$H_n = \begin{cases} 0 & \text{if } n = 0 \\ H_{n-1} + \dfrac{1}{n} & \text{otherwise} \end{cases}$$

[8] "Dave Guy!"

Approximating the summation $H_n = \sum_{i=1}^{n} \frac{1}{i}$ above and below by integrals implies the bounds

$$\ln(n+1) \leq H_n \leq (\ln n) + 1.$$

Thus, the expected number of cards we need to buy to get all n Pokémon is $\Theta(n \log n)$.

In particular, to catch all 150 of the original Pokémon, we should expect to buy $150 \cdot H_{150} \approx 838.67709$ cards, and to own at least one copy of each of the 9184 Pokémon card types available in 2013, we should expect to buy $9184 \cdot H_{9184} \approx 89107.65186$ cards. (In practice, of course, this estimate is far too low, because some cards are considerably more common than others.)

1.6 Random Permutations

Now suppose we are given a deck of n (Pokémon?) cards and are asked to shuffle them. Ideally, we would like an algorithm that produces each of the $n!$ possible permutations of the deck with the same probability $1/n!$.

There are many such algorithms, but the gold standard is ultimately based on the millennia-old tradition of drawing or casting lots. "Lots" are traditionally small pieces of wood, stone, or paper, which were blindly drawn from an opaque container. The following algorithm takes a set L of n distinct Lots (arbitrary objects) as input and returns an array $R[1..n]$ containing a Random permutation of those n lots.

```
DrawLots(L):
  n ← |L|
  for i ← 1 to n
      remove a random lot x from L
      R[i] ← x
  return R[1..n]
```

There are exactly $n!$ possible outcomes for this algorithm—exactly n choices for the first lot, then exactly $n-1$ choices for the second lot, and so on—each with exactly the same probability and each leading to a different permutation. Thus, every permutation of lots is equally likely to be output.

A modern formulation of lot-casting was described by (and is frequently misattributed to) statisticians Ronald Fisher and Frank Yates in 1938. Fisher and Yates formulated their algorithm as a method for randomly reordering a List of numbers. In their original formulation, the algorithm repeatedly chooses at random a number from the input list that has not been previously chosen, adds the chosen number to the output list, and then strikes the chosen number from the input list.

We can implement this formulation of the algorithm using a secondary boolean array $Chosen[1..n]$ indicating which items have already been chosen. The randomness is provided by a subroutine ***Random*(*n*)**, which returns an integer chosen independently and uniformly at random from the set $\{1, 2, \ldots, n\}$ in $O(1)$ time; in other words, Random(n) simulates a fair n-sided die.

```
FisherYates(L[1..n]):
  for i ← 1 down to n
      Chosen[i] ← False
  for i ← n down to 1
      repeat
          r ← Random(n)
      until ¬Chosen[r]
      R[i] ← L[r]
      Chosen[r] ← True
  return R[1..n]
```

The repeat-until loop chooses an index r uniformly at random from the set of previously unchosen indices. Thus, this algorithm really is an implementation of DrawLots. But now choosing the next random ~~lot~~ element may require several iterations of the repeat-until loop. How slow is this algorithm?

In fact, FisherYates is equivalent to our earlier Pokémon-collecting algorithm! Each call to Random is a purchase, and $Chosen[i]$ indicates whether we've already purchased the ith Pokémon. By our earlier analysis, the expected number of calls to Random before the algorithm halts is *exactly* nH_n. We conclude that this algorithm runs in $\boldsymbol{\Theta(n \log n)}$ ***expected time***.[9]

A most efficient implementation of lot-casting, which permutes the input array in place, was described by Richard Durstenfeld in 1961. In full accordance with Stigler's Law, this algorithm is almost universally called "the Fisher-Yates shuffle".[10]

```
SelectionShuffle(A[1..n]):
  for i ← n down to 1
      swap A[i] ↔ A[Random(i)]
```

The algorithm clearly runs in $O(n)$ time. Correctness follows from exactly the same argument as DrawLots: There are $n!$ equally likely possibilities from the n calls to Random—n for the first call, $n-1$ for the second, and so on—each leading to a different output permutation.

Although it may not appear so at first glance, SelectionShuffle is an implementation of lot-casting. After each iteration of the main loop, the suffix $A[i..n]$ (on the *R*ight) plays the role of R, storing the previously chosen input elements, and the prefix $A[1..i-1]$ (on the *L*eft) plays the role of L, containing all the unchosen input elements. Unlike the original FisherYates algorithm, SelectionShuffle changes the *order* of the unchosen elements as it runs. Fortunately, and obviously, that order is utterly irrelevant; only the *set* of unchosen elements matters.

We can also uniformly shuffle by reversing the order of the loop in the previous algorithm. Again, this algorithm is usually misattributed to Fisher and Yates.

```
InsertionShuffle(A[1..n]):
  for i ← 1 to n
      swap A[i] ↔ A[Random(i)]
```

Again, correctness follows from the observation that there are $n!$ equally likely output permutations. Alternatively, we can argue inductively that for every index i, the prefix $A[1..i]$ is

[9] In later editions of Fisher and Yates' monograph, they instead described a different algorithm due to C. Radhakrishna Rao, dismissing their earlier method as "tiresome, since each [item] must be deleted from a list as it is selected and a fresh count made for each further selection."

[10] However, some authors call this algorithm the "Knuth shuffle", because Donald Knuth described it in his landmark *Art of Computer Programming*, even though he attributed the algorithm to Durstenfeld in the first edition, and to Fisher and Yates in the second. It's actually rather shocking that Knuth did not attribute the algorithm to Aaron, citing the First Chronicles.

uniformly shuffled after the ith iteration of the loop. Alternatively, we can observe that running InsertionShuffle is the same as running SelectionShuffle *backward in time*—essentially putting the lots back into the bag—and that the inverse of a uniformly-distributed permutation is also uniformly distributed.

(The names SelectionShuffle and InsertionShuffle for these two variants are non-standard. SelectionShuffle randomly *selects* the next card and then adds to one end of the random permutation, just as selection sort repeatedly selects the largest element of the unsorted portion of the array. InsertionShuffle randomly *inserts* the first card in the untouched portion of the array into the random permutation, just as insertion sort repeatedly inserts the next item in the unsorted portion of the input into the sorted portion.)

1.7 Properties of Random Permutations

★★★

- For any subsequence of indices: the set of values and the permutation of those values are uniformly and independently distributed.
- For any subsequence of values: the set of indices and the permutation of those indices are uniformly and independently distributed.
- For example: In a randomly shuffled deck, the expected number of hearts among the first 5 face cards is 5/4.

Exercises

Several of these problems refer to decks of playing cards. A standard (Anglo-American) deck of 52 playing cards contains 13 cards in each of four suits: ♠ (spades), ♥ (hearts), ♦ (diamonds), and ♣ (clubs). The 13 cards in each suit have distinct *ranks*: A (ace), 2 (deuce), 3 (trey), 4, 5, 6, 7, 8, 9, 10, J (jack), Q (queen), and K (king). Cards are normally named by writing their rank followed by their suit; for example, J♠ is the jack of spades, and 10♥ is the ten of hearts. For purposes of comparing ranks in the problems below, aces have rank 1, jacks have rank 11, queens have rank 12, and kings have rank 13; for example, J♠ has higher rank than 8♦, but lower rank than Q♣.

---

1. On their long journey from Denmark to England, Rosencrantz and Guildenstern amuse themselves by playing the following game with a fair coin. First Rosencrantz flips the coin over and over until it comes up tails. Then Guildenstern flips the coin over and over until he gets as many heads in a row as Rosencrantz got on his turn. Here are three typical games:

   Rosencrantz: **H H** T
   Guildenstern: H T **H H**

   Rosencrantz: T
   Guildenstern: (no flips)

   Rosencrantz: **H H H** T
   Guildenstern: T H H T H H T H T T **H H H**

   (a) What is the expected number of flips in one of Rosencrantz's turns?

   (b) Suppose Rosencrantz happens to flip k heads in a row on his turn. What is the expected number of flips in Guildenstern's next turn?

(c) What is the expected total number of flips (by both Rosencrantz and Guildenstern) in a single game?

Prove that your answers are correct. If you have to appeal to "intuition" or "common sense", your answer is almost certainly wrong! Full credit requires *exact* answers, but a correct asymptotic bound (as a function of k) in part (b) is worth significant partial credit.

2. After sending his loyal friends Rosencrantz and Guildenstern off to Norway, Hamlet decides to amuse himself by repeatedly flipping a fair coin until the sequence of flips satisfies some condition. For each of the following conditions, compute the *exact* expected number of flips until that condition is met. Some conditions depend on a positive integer parameter n.

   (a) Hamlet flips n times.

   (b) Hamlet flips heads.

   (c) Hamlet flips both heads and tails (in different flips, of course).

   (d) Hamlet flips heads twice.

   (e) Hamlet flips heads twice in a row.

   (f) Hamlet flips heads followed immediately by tails.

   (g) Hamlet flips the sequence heads, tails, heads, tails.

   (h) Hamlet flips heads n times.

   (i) Hamlet flips heads n times in a row.

   (j) Hamlet flips more heads than tails.

   (k) Hamlet flips the same positive number of heads and tails.

   (l) Either Hamlet flips more heads than tails or he flips n times, whichever comes first.

   For each condition, prove that your answer is correct. If you have to appeal to "intuition" or "common sense", your answer is almost certainly wrong! Correct asymptotic bounds for the conditions that depend on n are worth significant partial credit.

3. Suppose you have access to a function FairCoin that returns a single random bit, chosen uniformly and independently from the set $\{0, 1\}$, in $O(1)$ time. Consider the following randomized algorithm for generating biased random bits.

```
OneInThree:
  if FairCoin = 0
      return 0
  else
      return 1 − OneInThree
```

   (a) Prove that OneInThree returns 1 with probability $1/3$.

   (b) What is the *exact* expected number of times that this algorithm calls FairCoin?

   (c) Now suppose instead of FairCoin you are given a subroutine BiasedCoin that returns an independent random bit equal to 1 with some fixed *but unknown* probability p, in $O(1)$ time. Describe an algorithm OneInThree that returns either 0 or 1 with equal probability, using BiasedCoin as its only source of randomness.

(d) What is the *exact* expected number of times that your OneInThree algorithm calls BiasedCoin?

4. Describe an algorithm that takes a real number $0 \leq p \leq 1$ as input, and returns an independent random bit that is equal to 1 with the given probability p, using independent fair coin flips as the only source of randomness. What is the expected running time of your algorithm (as a function of p)?

5. (a) Describe an algorithm that simulates a fair three-sided die, using independent fair coin flips as the only source of randomness. Your algorithm should return 1, 2, or 3, each with probability $1/3$. What is the expected number of coin flips used by your algorithm?

   (b) Describe an algorithm that simulates a fair coin flip, using independent rolls of a fair three-sided die as the only source of randomness. What is the expected number of die rolls used by your algorithm?

   *(c) Describe an algorithm that simulates a fair n-sided die, using independent rolls of a fair k-sided die as the only source of randomness. What is the expected number of die rolls used by your algorithm (as a function of n and k)? You may assume n and k are relatively prime.

6. Describe an algorithm FairDie that returns an integer chosen uniformly at random from the set $\{1, 2, 3, 4, 5, 6\}$, using an algorithm LoadedDie that returns an algorithm from the same set with some fixed *but unknown* non-trivial probability distribution. What is the expected number of times your FairDie algorithm calls LoadedDie? *[Hint:* $3! = 6$.*]*

7. (a) Suppose you have access to a function FairCoin that returns a single random bit, chosen uniformly and independently from the set $\{0, 1\}$, in $O(1)$ time. Describe and analyze an algorithm Random(n) that returns an integer chosen uniformly and independently at random from the set $\{1, 2, \ldots, n\}$, given a non-negative integer n as input, using FairCoin as its only source of randomness.

   (b) Suppose you have access to a function FairCoins(k) that returns an integer chosen uniformly and independently at random from the set $\{0, 1, \ldots, 2^k - 1\}$ in $O(1)$ time, given *any* non-negative integer k as input. Describe and analyze an algorithm Random(n) that returns an integer chosen uniformly and independently at random from the set $\{1, 2, \ldots, n\}$, given *any* non-negative integer n as input, using FairCoins as its only source of randomness.

8. You are applying to participate in this year's Trial of the Pyx, the annual ceremony where samples of all British coinage are tested, to ensure that they conform as strictly as possible to legal standards. As a test of your qualifications, your interviewer at the Worshipful Company of Goldsmiths has given you a bag of n commemorative Alan Turing half-guinea coins, exactly two of which are counterfeit. One counterfeit coin is very slightly lighter than a genuine Turing; the other is very slightly heavier. Together, the two counterfeit coins have *exactly* the same weight as two genuine coins. Your task is to identify the two counterfeit coins.

The weight difference between the real and fake coins is too small to be detected by anything other than the Royal Pyx Coin Balance. You can place any two disjoint sets of coins in each of the Balance's two pans; the Balance will then indicate which of the two subsets has larger total weight, or that the two subsets have the same total weight. Unfortunately, each use of the Balance requires completing a complicated authorization form (in triplicate), submitting a blood sample, and scheduling the Royal Bugle Corps, so you *really* want to use the Balance as few times as possible.

(a) Suppose you *randomly* choose $n/2$ of your n coins to put on one pan of the Balance, and put the remaining $n/2$ coins on the other pan. What is the probability that the two subsets have equal weight?

(b) Describe and analyze a randomized algorithm to identify the two fake coins. What is the expected number of times your algorithm uses the Balance?

9. Consider the following algorithm for shuffling a deck of n cards, initially numbered in order from 1 on the top to n on the bottom. At each step, we remove the top card from the deck an insert it randomly back into in the deck, choosing one of the n possible positions uniformly at random. The algorithm ends immediately after we pick up card $n-1$ and insert it randomly into the deck.

(a) Prove that this algorithm uniformly shuffles the deck, so that each permutation of the deck has equal probability. *[Hint: Prove that at all times, the cards below card $n-1$ are uniformly shuffled.]*

(b) Prove that before the algorithm ends, the deck is *not* uniformly shuffled.

(c) What is the expected number of steps before this algorithm ends?

10. Suppose we want to write an efficient algorithm SHUFFLE($A[1..n]$) that randomly permutes the input array, so that each of the $n!$ permutations is equally likely.

(a) Prove that the following algorithm is ***not*** correct. *[Hint: Consider the case $n = 3$.]*

```
NaiveShuffle(A[1..n]):
  for i ← 1 to n
      swap A[i] ↔ A[Random(n)]
```

(The only difference from InsertionShuffle is that the argument to Random is n instead of i.)

(b) The algorithm BufferShuffle, shown in Figure 1.2, takes an extra parameter b describing the size of the internal buffer array $B[1..b]$.

i. Prove that BufferShuffle is correct for all $b \geq n$.

ii. What is the expected running time of BufferShuffle when $b = n$?

iii. What is the expected running time of BufferShuffle when $b = 2n$?

*(c) Prove that the algorithm RaoShuffle, shown in Figure 1.2, is correct and analyze its expected running time. This is the algorithm of C. Radhakrishna Rao that Fisher and Yates found less "tiresome" than their own.

```
BufferShuffle(n, b):
  ⟨⟨Initialize buffer⟩⟩
  for j ← 1 to 2n
      B[j] ← Null
  ⟨⟨Copy items randomly into buffer⟩⟩
  for i ← 1 to n
      j ← Random(b)
      while (B[j] == Null)
          j ← Random(n)
      B[j] ← A[i]
  ⟨⟨Compress buffer into input array⟩⟩
  i ← 1
  for j ← 1 to b
      if B[j] ≠ Null
          A[i] ← B[j]
          i ← i + 1
```

```
RaoShuffle(A[1..n]):
  ⟨⟨Initialize n lists⟩⟩
  for j ← 1 to n
      ℓ[j] ← 0
  ⟨⟨Add each A[i] to a random list⟩⟩
  for i ← 1 to n
      ℓ[j] ← ℓ[j] + 1
      L[j][ℓ[j]] ← A[i]
  ⟨⟨Recursively shuffle the lists⟩⟩
  for j ← 1 to n
      if ℓ[j] > 1
          Shuffle(L[j][1..ℓ[j]])
  ⟨⟨Concatenate the lists⟩⟩
  i ← 0
  for j ← 1 to n
      for k ← 1 to ℓ[j]
          A[i] ← L[j][k]
          i ← i + 1
```

Figure 1.2. Left: Shuffling using a buffer array. Right: Rao's less "tiresome" shuffling algorithm.

11. (a) Prove that the following algorithm, modeled after quicksort, uniformly permutes its input array, meaning each of the $n!$ possible output permutations is equally likely. *[Hint:* $\binom{n}{k} = \frac{n!}{k!(n-k)!}$*]*

```
QuickShuffle(A[1..n]):
  if n ≤ 1
      return
  j ← 1;  k ← n
  while j ≤ k
      with probability 1/2
          swap A[j] ↔ A[k]
          k ← k − 1
      else
          j ← j + 1
  QuickShuffle(A[1..k])
  QuickShuffle(A[j..n])
```

(b) Prove that QuickShuffle runs in $O(n \log n)$ expected time. *[Hint: This will be much easier after reading the next chapter.]*

(c) Prove that the algorithm MergeShuffle shown on the next page, which is modeled after mergesort, does ***not*** uniformly permute its input array.

(d) Describe how to modify RandomMerge so that MergeShuffle ***does*** uniformly permute its input array, and prove that your modification is correct. *[Hint: Modify the line in red.]*

```
MergeShuffle(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MergeShuffle(A[1..m])
      MergeShuffle(A[m+1..n])
      RandomMerge(A[1..n], m)
```

```
RandomMerge(A[1..n], m):
  i ← 1;  j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i+1
      else if i > m
          B[k] ← A[j];  j ← j+1
      else with probability 1/2
          B[k] ← A[i];  i ← i+1
      else
          B[k] ← A[j];  j ← j+1
  for k ← 1 to n
      A[k] ← B[k]
```

12. Clock Solitaire is played with a standard deck of playing cards. To set up the game, deal the cards face down into 13 piles of four cards each, one in each of the "hour" positions of a clock and one in the center. Each pile corresponds to a particular rank—A through Q in clockwise order for the hour positions, and K for the center. To start the game, turn over a card in the center pile. Then repeatedly turn over a card in the pile corresponding to the value of the previous card. The game ends when you try to turn over a card from a pile whose four cards are already face up. (This is always the center pile—why?) You win if and only if every card is face up when the game ends.

    What is the *exact* probability that you win a game of Clock Solitaire, assuming that the cards are permuted uniformly at random before they are dealt into their piles?

13. Professor Jay is about to perform a public demonstration with two decks of cards, one with red backs ("the red deck") and one with blue backs ("the blue deck"). Both decks lie face-down on a table in front of the good Professor, shuffled so that every permutation of each deck is equally likely.

    To begin the demonstration, Professor Jay turns over the top card from each deck. If one of these two cards is the three of clubs (3♣), the demonstration ends immediately. Otherwise, the good Professor repeatedly hurls the cards he just turned over into the thick, pachydermatous outer melon layer of a nearby watermelon, and then turns over the next card from the top of each deck. The demonstration ends the first time a 3♣ is turned over. Thus, if 3♣ is the last card in both decks, the demonstration ends with 102 cards embedded in the watermelon, that most prodigious of household fruits.

    (a) What is the *exact* expected number of cards that Professor Jay hurls into the watermelon?

    (b) For each of the statements below, give the *exact* probability that the statement is true of the ***first*** pair of cards Professor Jay turns over.

        i. Both cards are threes.

        ii. One card is a three, and the other card is a club.

        iii. If (at least) one card is a heart, then (at least) one card is a diamond.

        iv. The card from the red deck has higher rank than the card from the blue deck.

    (c) For each of the statements below, give the *exact* probability that the statement is true of the ***last*** pair of cards Professor Jay turns over.

i. Both cards are threes.
ii. One card is a three, and the other card is a club.
iii. If (at least) one card is a heart, then (at least) one card is a diamond.
iv. The card from the red deck has higher rank than the card from the blue deck.

14. Penn and Teller agree to play the following game. Penn shuffles a standard deck of playing cards so that every permutation is equally likely. Then Teller draws cards from the deck, one at a time without replacement, until he draws the three of clubs (3♣), at which point the remaining undrawn cards instantly burst into flames.

    The first time Teller draws a card from the deck, he gives it to Penn. From then on, until the game ends, whenever Teller draws a card whose value is smaller than the last card he gave to Penn, he gives the new card to Penn.[11] To make the rules unambiguous, they agree beforehand that $A = 1$, $J = 11$, $Q = 12$, and $K = 13$.

    (a) What is the expected number of cards that Teller draws?
    (b) What is the expected *maximum* value among the cards Teller gives to Penn?
    (c) What is the expected *minimum* value among the cards Teller gives to Penn?
    (d) What is the expected number of cards that Teller gives to Penn? *[Hint: Let* $13 = n$*.]*

15. Consider a random walk on a path with vertices numbered $1, 2, \ldots, n$ from left to right. At each step, we flip a coin to decide which direction to walk, moving one step left or one step right with equal probability. The random walk ends when we fall off one end of the path, either by moving left from vertex 1 or by moving right from vertex n.

    (a) Prove that the probability that the walk ends by falling off the *right* end of the path is exactly $1/(n+1)$.
    (b) Prove that if we start at vertex k, the probability that we fall off the *right* end of the path is exactly $k/(n+1)$.
    (c) Prove that if we start at vertex 1, the expected number of steps before the random walk ends is exactly n.
    (d) What is the *exact* expected length of the random walk if we start at vertex k, as a function of n and k? Prove your result is correct. (For partial credit, give a tight Θ-bound for the case $k = (n+1)/2$, assuming n is odd.)

    [Hint: Trust the recursion fairy.]

16. Suppose n lights labeled $0, \ldots, n-1$ are placed clockwise around a circle. Initially, every light is off. Consider the following random process.

[11] Specifically, he hurls it directly into the back of Penn's right hand.

```
LightTheCircle(n):
  k ← 0
  turn on light 0
  while at least one light is off
      with probability 1/2
          k ← (k + 1) mod n
      else
          k ← (k − 1) mod n
      if light k is off, turn it on
```

(a) Let $p(i, n)$ denote the probability that the last light turned on by LightTheCircle$(n, 0)$ is light i. For example, $p(0, 2) = 0$ and $p(1, 2) = 1$. Find an exact closed-form expression for $p(i, n)$ in terms of n and i. Prove your answer is correct.

(b) Give the tightest upper bound you can on the expected running time of this algorithm. *[Hint: You may find the previous exercise helpful.]*

17. Let T be an arbitrary, ***not*** necessarily balanced, binary tree T with n nodes.

(a) Consider a random walk downward from the root of T, as described by the following algorithm.

```
RandomTreeDescent(T):
  v ← T.root
  while v ≠ Null
      with probability 1/2
          v ← v.left
      else
          v ← v.right
```

Find a tight bound on the worst-case expected running time of this algorithm, as a function of n, and prove your answer is correct. What is the worst-case tree?

(b) Now consider a different random walk starting at the root of T, as described by the following algorithm.

```
RandomTreeWalk(T):
  v ← T.root
  while v ≠ Null
      with probability 1/3
          v ← v.left
      else with probability 1/3
          v ← v.right
      else
          v ← v.parent
```

Find a tight bound on the worst-case expected running time of this algorithm, as a function of n, and prove your answer is correct. What is the worst-case tree?

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

The first nuts and bolts appeared in the middle 1400's. The bolts were just screws with straight sides and a blunt end. The nuts were hand-made, and very crude. When a match was found between a nut and a bolt, they were kept together until they were finally assembled.

In the Industrial Revolution, it soon became obvious that threaded fasteners made it easier to assemble products, and they also meant more reliable products. But the next big step came in 1801, with Eli Whitney, the inventor of the cotton gin. The lathe had been recently improved. Batches of bolts could now be cut on different lathes, and they would all fit the same nut.

Whitney set up a demonstration for President Adams, and Vice-President Jefferson. He had piles of musket parts on a table. There were 10 similar parts in each pile. He went from pile to pile, picking up a part at random. Using these completely random parts, he quickly put together a working musket.

— Karl S. Kruszelnicki ("Dr. Karl"), *Karl Trek*, December 1997

Dr [John von] Neumann in his Theory of Games and Economic Behavior *introduces the cut-up method of random action into game and military strategy: Assume that the worst has happened and act accordingly. If your strategy is at some point determined. . . by random factor your opponent will gain no advantage from knowing your strategy since he cannot predict the move.*

— William S. Burroughs, "The Cut-Up Method of Brion Gysin"
in *The Third Mind* by William S. Burroughs and Brion Gysin (1978)

2 Randomized Algorithms

2.1 Nuts and Bolts

The following interesting problem was first proposed by Gregory Rawlins around 1992. Suppose we are given n nuts and n bolts of different sizes. Each nut matches exactly one bolt and vice versa. The nuts and bolts are all almost exactly the same size, and we're playing with them in the dark, so we can't tell if one bolt is bigger than the other, or if one nut is bigger than the other. If we try to match a nut witch a bolt, however, the nut will be either too big, too small, or just right for the bolt. How quickly can we match each but to the corresponding bolt?

Before we attack this problem, let's think about something simpler. Suppose we want to find the nut that matches a particular bolt. The obvious algorithm — test every nut until we find a match — requires exactly $n-1$ tests in the worst case. We might have to check every but except one; if we get to the last nut without finding a match, we know that the last nut is the one we're looking for.

Intuitively, in the "average" case, this algorithm will look at approximately $n/2$ nuts. But what exactly does "average case" mean?

2.2 Deterministic vs. Randomized Algorithms

Normally, when we talk about the running time of an algorithm, we mean its *worst-case* running time. This is the maximum, over all problems of a certain size, of the running time of that algorithm on that input:

$$T_{\text{worst-case}}(n) = \max_{|X|=n} T(X).$$

The *average-case* running time is best defined by the *expected value*, over all inputs X of a certain size, of the algorithm's running time for input X:

$$T_{\text{average-case}}(n) = \mathop{\mathrm{E}}_{|X|=n}[T(X)] = \sum_{|X|=n} T(X) \cdot \Pr[X].$$

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

The problem with this definition is that we rarely, if ever, know the probability of getting any particular input X. We could compute average-case running times by *assuming* a particular probability distribution—for example, every possible input is equally likely—but this assumption doesn't describe reality very well. Most real-life data, if it can be considered random at all, is random in some interesting way that can't be predicted in advance.

Instead of considering the rather questionable notion of average case running time, we will make a distinction between two kinds of algorithms: ***deterministic*** and ***randomized***. A deterministic algorithm is one that always behaves the same way given the same input; the input completely *determines* the sequence of computations performed by the algorithm. Randomized algorithms, on the other hand, base their behavior not only on the input but also on several *random* choices. The same randomized algorithm, given the same input multiple times, may perform different computations in each invocation.

This definition implies that the running time of a randomized algorithm on a given input is no longer fixed, but is itself a random variable. When we analyze randomized algorithms, we are typically interested in the ***worst-case expected*** running time. That is, we consider the expected running time for each input, and then choose the maximum over all inputs of a certain size:

$$T_{\text{worst-case expected}}(n) = \max_{|X|=n} \mathrm{E}[T(X)].$$

It's important to note here that we are making *no* assumptions about the probability distribution of possible inputs. All the randomness is inside the algorithm, where we can control it!

2.3 Back to Nuts and Bolts

Let's go back to the problem of finding the nut that matches a given bolt. Suppose we use the same algorithm as before, but at each step we choose a nut *uniformly at random* from the untested nuts. So if there are k nuts left to test, each one will be chosen with probability $1/k$. Now what's the expected number of comparisons we have to perform? Intuitively, it should be about $n/2$, but let's formalize our intuition.

Let $T(n)$ denote the number of comparisons our algorithm uses to find a match for a single bolt out of n nuts.[1] We still have some simple base cases $T(1) = 0$ and $T(2) = 1$, but when $n > 2$, $T(n)$ is a random variable. $T(n)$ is always between 1 and $n-1$; it's actual value depends on our algorithm's random choices. We are interested in the *expected value* or *expectation* of $T(n)$, which is defined as follows:

$$\mathrm{E}[T(n)] = \sum_{k=1}^{n-1} k \cdot \Pr[T(n) = k]$$

If the target nut is the kth nut tested, our algorithm performs $\min\{k, n-1\}$ comparisons. In particular, if the target nut is the last nut chosen, we don't actually test it. Because we choose the next nut to test uniformly at random, the target nut is equally likely—with probability exactly $1/n$—to be the first, second, third, or kth bolt tested, for any k. Thus:

$$\Pr[T(n) = k] = \begin{cases} 1/n & \text{if } k < n-1, \\ 2/n & \text{if } k = n-1. \end{cases}$$

[1]Note that for this algorithm, the input is completely specified by the number n. Since we're choosing the nuts to test at random, even the order in which the nuts and bolts are presented doesn't matter. That's why I'm using the simpler notation $T(n)$ instead of $T(X)$.

Plugging this into the definition of expectation gives us our answer.

$$\begin{aligned}\mathrm{E}[T(n)] &= \sum_{k=1}^{n-2} \frac{k}{n} + \frac{2(n-1)}{n} \\ &= \sum_{k=1}^{n} \frac{k}{n} - \frac{1}{n} \\ &= \frac{n+1}{2} - \frac{1}{n}\end{aligned}$$

We can get exactly the same answer by thinking of this algorithm recursively. We always have to perform at least one test. With probability $1/n$, we successfully find the matching nut and halt. With the remaining probability $1-1/n$, we recursively solve the same problem but with one fewer nut. We get the following recurrence for the expected number of tests:

$$T(1) = 0, \qquad \mathrm{E}[T(n)] = 1 + \frac{n-1}{n}\,\mathrm{E}[T(n-1)]$$

To get the solution, we define a new function $t(n) = n\,\mathrm{E}[T(n)]$ and rewrite:

$$t(1) = 0, \qquad t(n) = n + t(n-1)$$

This recurrence translates into a simple summation, which we can easily solve.

$$\begin{aligned} t(n) &= \sum_{k=2}^{n} k = \frac{n(n+1)}{2} - 1 \\ \implies \mathrm{E}[T(n)] &= \frac{t(n)}{n} = \frac{n+1}{2} - \frac{1}{n}\end{aligned}$$

2.4 Finding All Matches

Not let's go back to the problem introduced at the beginning of the lecture: finding the matching nut for every bolt. The simplest algorithm simply compares every nut with every bolt, for a total of n^2 comparisons. The next thing we might try is repeatedly finding an arbitrary matched pair, using our very first nuts and bolts algorithm. This requires

$$\sum_{i=1}^{n} (i-1) = \frac{n^2-n}{2}$$

comparisons in the worst case. So we save roughly a factor of two over the really stupid algorithm. Not very exciting.

Here's a more interesting strategy, which Rawlins proposed when he introduced the problem. Choose a *pivot* bolt, and test it against every nut. Then test the matching pivot nut against every other bolt. After these $2n-1$ tests, we have one matched pair, and the remaining nuts and bolts are partitioned into two subsets: those smaller than the pivot pair and those larger than the pivot pair. Finally, recursively match up the two subsets. The worst-case number of tests made by this algorithm is given by the recurrence

$$\begin{aligned} T(n) &= 2n - 1 + \max_{1 \le k \le n} \big\{ T(k-1) + T(n-k) \big\} \\ &= 2n - 1 + T(n-1)\end{aligned}$$

Along with the trivial base case $T(0) = 0$, this recurrence solves to

$$T(n) = \sum_{i=1}^{n}(2n-1) = n^2.$$

In the worst case, this algorithm tests *every* nut-bolt pair! We could have been a little more clever—for example, if the pivot bolt is the smallest bolt, we only need $n-1$ tests to partition everything, not $2n-1$—but cleverness doesn't actually help that much. We still end up with about $n^2/2$ tests in the worst case.

However, since this recursive algorithm looks almost exactly like quicksort, and everybody 'knows' that the 'average-case' running time of quicksort is $\Theta(n \log n)$, it seems reasonable to guess that the average number of nut-bolt comparisons is also $\Theta(n \log n)$. As we shall see shortly, if the pivot bolt is always chosen *uniformly at random*, this intuition is exactly right.

2.5 Reductions to and from Sorting

Rawlins' algorithm for matching the nuts and bolts strongly resembles quicksort. The algorithm not only matches up the nuts and bolts, but also sorts them by size.

In fact, the problems of sorting and matching nuts and bolts are equivalent, in the following sense. If the bolts were sorted, we could match the nuts and bolts in $O(n \log n)$ time by performing a binary search with each nut. Thus, if we had an algorithm to sort the bolts in $O(n \log n)$ time, we would immediately have an algorithm to match the nuts and bolts, starting from scratch, in $O(n \log n)$ time. This process of *assuming* a solution to one problem and using it to solve another is called *reduction*—we can *reduce* the matching problem to the sorting problem in $O(n \log n)$ time.

There is a reduction in the other direction, too. If the nuts and bolts were matched, we could sort them in $O(n \log n)$ time using, for example, merge sort. Thus, if we have an $O(n \log n)$ time algorithm for either sorting or matching nuts and bolts, we automatically have an $O(n \log n)$ time algorithm for the other problem.

Unfortunately, since we aren't allowed to directly compare two bolts or two nuts, we can't use heapsort or mergesort to sort the nuts and bolts in $O(n \log n)$ worst case time. In fact, the problem of sorting nuts and bolts *deterministically* in $O(n \log n)$ time was only "solved" in 1995[2], but both the algorithms and their analysis are incredibly technical and the constant hidden in the $O(\cdot)$ notation is quite large.

2.6 Recursive Analysis

Intuitively, we can argue that our quicksort-like algorithm will usually choose a bolt of approximately median size, and so the average numbers of tests should be $O(n \log n)$. We can now finally formalize this intuition. To simplify the notation slightly, I'll write $\overline{T}(n)$ in place of $\mathrm{E}[T(n)]$ everywhere.

Our randomized matching/sorting algorithm chooses its pivot bolt *uniformly at random* from the set of unmatched bolts. Since the pivot bolt is equally likely to be the smallest, second smallest, or kth smallest for any k, the expected number of tests performed by our algorithm is

[2] János Komlós, Yuan Ma, and Endre Szemerédi, Sorting nuts and bolts in $O(n \log n)$ time, *SIAM J. Discrete Math* 11(3):347–372, 1998. See also Phillip G. Bradford, Matching nuts and bolts optimally, Technical Report MPI-I-95-1-025, Max-Planck-Institut für Informatik, September 1995. Bradford's algorithm is *slightly* simpler.

given by the following recurrence:

$$\begin{aligned}\overline{T}(n) &= 2n-1+\mathrm{E}_k\big[\overline{T}(k-1)+\overline{T}(n-k)\big] \\ &= 2n-1+\frac{1}{n}\sum_{k=1}^{n}\big(\overline{T}(k-1)+\overline{T}(n-k)\big)\end{aligned}$$

The base case is $T(0)=0$. (We can save a few tests by setting $T(1)=0$ instead of 1, but the analysis will be easier if we're a little stupid.)

Yuck. At this point, we could simply *guess* the solution, based on the incessant rumors that quicksort runs in $O(n\log n)$ time in the average case, and prove our guess correct by induction. (See Section 2.8 below for details.)

However, if we're only interested in asymptotic bounds, we can afford to be a little conservative. What we'd *really* like is for the pivot bolt to be the median bolt, so that half the bolts are bigger and half the bolts are smaller. This isn't very likely, but there is a good chance that the pivot bolt is close to the median bolt. Let's say that a pivot bolt is *good* if it's in the middle half of the final sorted set of bolts, that is, bigger than at least $n/4$ bolts and smaller than at least $n/4$ bolts. If the pivot bolt is good, then the *worst* split we can have is into one set of $3n/4$ pairs and one set of $n/4$ pairs. If the pivot bolt is bad, then our algorithm is still better than starting over from scratch. Finally, a randomly chosen pivot bolt is good with probability $1/2$.

These simple observations give us the following simple recursive *upper bound* for the expected running time of our algorithm:

$$\overline{T}(n) \le 2n-1+\frac{1}{2}\left(\overline{T}\left(\frac{3n}{4}\right)+\overline{T}\left(\frac{n}{4}\right)\right)+\frac{1}{2}\cdot\overline{T}(n)$$

A little algebra simplifies this even further:

$$\overline{T}(n) \le 4n-2+\overline{T}\left(\frac{3n}{4}\right)+\overline{T}\left(\frac{n}{4}\right)$$

We can solve this recurrence using the recursion tree method, giving us the unsurprising upper bound $\overline{T}(n)=O(n\log n)$. A similar argument gives us the matching lower bound $\overline{T}(n)=\Omega(n\log n)$.

Unfortunately, while this argument may be convincing, it is *not* a formal proof, because it relies on the unproven assumption that $\overline{T}(n)$ is a *convex* function, which means that $\overline{T}(n+1)+\overline{T}(n-1)\ge 2\overline{T}(n)$ for all n. $\overline{T}(n)$ is actually convex, but we never proved it. Convexity follows form the closed-form solution of the recurrence, but using that fact would be circular logic. Sadly, formally proving convexity seems to be almost as hard as solving the recurrence. If we want a *proof* of the expected cost of our algorithm, we need another way to proceed.

2.7 Iterative Analysis

By making a simple change to our algorithm, which has no effect on the number of tests, we can analyze it much more directly and exactly, without solving a recurrence or relying on hand-wavy intuition.

The recursive subproblems solved by quicksort can be laid out in a binary tree, where each node corresponds to a subset of the nuts and bolts. In the usual recursive formulation, the algorithm partitions the nuts and bolts at the root, then the left child of the root, then the leftmost grandchild, and so forth, recursively sorting everything on the left before starting on the right subproblem.

★★★ Figure

But we don't have to solve the subproblems in this order. In fact, we can visit the nodes in the recursion tree in any order we like, as long as the root is visited first, and any other node is visited after its parent. Thus, we can recast quicksort in the following iterative form. Choose a pivot bolt, find its match, and partition the remaining nuts and bolts into two subsets. Then pick a second pivot bolt and partition whichever of the two subsets contains it. At this point, we have two matched pairs and three subsets of nuts and bolts. Continue choosing new pivot bolts and partitioning subsets, each time finding one match and increasing the number of subsets by one, until every bolt has been chosen as the pivot. At the end, every bolt has been matched, and the nuts and bolts are sorted.

Suppose we always choose the next pivot bolt *uniformly at random* from the bolts that haven't been pivots yet. Then no matter which subset contains this bolt, the pivot bolt is equally likely to be any bolt *in that subset*. It follows (by induction) that our randomized iterative algorithm performs *exactly* the same set of tests as our randomized recursive algorithm, but possibly in a different order.

Now let B_i denote the ith smallest bolt, and N_j denote the jth smallest nut. For each i and j, define an indicator variable X_{ij} that equals 1 if our algorithm compares B_i with N_j and zero otherwise. Then the total number of nut/bolt comparisons is exactly

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{n} X_{ij}.$$

We are interested in the expected value of this double summation; linearity of expectation allows us to write this expected value as follows:

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\sum_{i=1}^{n}\sum_{j=1}^{n} X_{ij}\right] = \sum_{i=1}^{n}\sum_{j=1}^{n} \mathrm{E}[X_{ij}].$$

By definition of expectation,

$$\mathrm{E}[X_{ij}] = 0 \cdot \Pr[X_{ij} = 0] + 1 \cdot \Pr[X_{ij} = 1] = \Pr[X_{ij} = 1].$$

Thus, to analyze our algorithm, we only need to calculate $\Pr[X_{ij} = 1]$ for all i and j.

First let's assume that $i < j$. The only comparisons our algorithm performs are between some pivot bolt (or its partner) and a nut (or bolt) in the same subset. The only event that can prevent a comparison between B_i and N_j is choosing some intermediate pivot bolt B_k, with $i < k < j$, before B_i or B_j. In other words:

Our algorithm compares B_i and N_j if and only if the first pivot chosen from the set $\{B_i, B_{i+1}, \ldots, B_j\}$ is either B_i or B_j.

Since the set $\{B_i, B_{i+1}, \ldots, B_j\}$ contains $j - i + 1$ bolts, each of which is equally likely to be chosen first, we immediately have

$$\mathrm{E}[X_{ij}] = \frac{2}{j - i + 1} \qquad \text{for all } i < j.$$

Symmetric arguments give us $\mathrm{E}[X_{ij}] = \frac{2}{i-j+1}$ for all $i > j$. Since our algorithm is a little stupid, every bolt is compared with its partner, so $X_{ii} = 1$ for all i. (In fact, if a pivot bolt is the only bolt in its subset, we don't need to compare it against its partner, but this improvement complicates the analysis.)

Putting everything together, we get the following summation.

$$\begin{aligned}
\mathrm{E}[T(n)] &= \sum_{i=1}^{n}\sum_{j=1}^{n}\mathrm{E}[X_{ij}] \\
&= \sum_{i=1}^{n}\mathrm{E}[X_{ii}] + 2\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathrm{E}[X_{ij}] \\
&= n + 4\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{1}{j-i+1}
\end{aligned}$$

This is quite a bit simpler than our earlier recurrence. Just a few more lines of algebra gives us an exact, closed-form expression for the expected number of comparisons.

$$\begin{aligned}
\mathrm{E}[T(n)] &= n + 4\sum_{i=1}^{n}\sum_{k=2}^{n-i+1}\frac{1}{k} && \text{[substitute } k = j-i+1\text{]} \\
&= n + 4\sum_{k=2}^{n}\sum_{i=1}^{n-k+1}\frac{1}{k} && \text{[reorder summations]} \\
&= n + 4\sum_{k=2}^{n}\frac{n-k+1}{k} \\
&= n + 4\left((n+1)\sum_{k=2}^{n}\frac{1}{k} - \sum_{k=2}^{n}1\right) \\
&= n + 4((n+1)(H_n - 1) - (n-1)) \\
&= 4nH_n - 7n + 4H_n
\end{aligned}$$

Sure enough, it's $\Theta(n \log n)$.

*2.8 Masochistic Analysis

If we're feeling particularly masochistic, we can actually solve the recurrence directly, all the way to an exact closed-form solution. I'm including this only to show you it can be done; this won't be on the test.

First we simplify the recurrence slightly by combining symmetric terms.

$$\overline{T}(n) \;=\; \frac{1}{n}\sum_{k=1}^{n}\big(\overline{T}(k-1) + \overline{T}(n-k)\big) \;+\; 2n-1 \;=\; \frac{2}{n}\sum_{k=0}^{n-1}\overline{T}(k) \;+\; 2n-1$$

We then convert this 'full history' recurrence into a 'limited history' recurrence by shifting and subtracting away common terms. (I call this "Magic step #1".) To make this step slightly easier, we first multiply both sides of the recurrence by n to get rid of the fractions.

$$n\overline{T}(n) = 2\sum_{k=0}^{n-1}\overline{T}(k) \;+\; 2n^2 - n$$

$$
\begin{aligned}
(n-1)\overline{T}(n-1) &= 2\sum_{k=0}^{n-2}\overline{T}(k) \;+\; 2(n-1)^2-(n-1) \\
&= 2\sum_{k=0}^{n-2}\overline{T}(k) \;+\; 2n^2-5n+3 \\
n\overline{T}(n)-(n-1)\overline{T}(n-1) &= 2T(n-1)+4n-3 \\
\overline{T}(n) &= \frac{n+1}{n}\overline{T}(n-1)+4-\frac{3}{n}
\end{aligned}
$$

To solve this limited-history recurrence, we define a new function $t(n) = \overline{T}(n)/(n+1)$. (I call this "Magic step #2".) This gives us an even simpler recurrence for $t(n)$ in terms of $t(n-1)$:

$$
\begin{aligned}
t(n) &= \frac{\overline{T}(n)}{n+1} \\
&= \frac{1}{n+1}\left((n+1)\frac{T(n-1)}{n}+4-\frac{3}{n}\right) \\
&= t(n-1)+\frac{4}{n+1}-\frac{3}{n(n+1)} \\
&= t(n-1)+\frac{7}{n+1}-\frac{3}{n}
\end{aligned}
$$

I used the technique of partial fractions (remember calculus?) to replace $\frac{1}{n(n+1)}$ with $\frac{1}{n}-\frac{1}{n+1}$ in the last step. The base case for this recurrence is $t(0)=0$. Once again, we have a recurrence that translates directly into a summation, which we can solve with just a few lines of algebra.

$$
\begin{aligned}
t(n) &= \sum_{i=1}^{n}\left(\frac{7}{i+1}-\frac{3}{i}\right) \\
&= 7\sum_{i=1}^{n}\frac{1}{i+1}-3\sum_{i=1}^{n}\frac{1}{i} \\
&= 7(H_{n+1}-1)-3H_n \\
&= 4H_n-7+\frac{7}{n+1}
\end{aligned}
$$

The last step uses the recursive definition of the harmonic numbers: $H_{n+1} = H_n + \frac{1}{n+1}$. Finally, substituting $\overline{T}(n) = (n+1)t(n)$ and simplifying gives us the exact solution to the original recurrence.

$$\overline{T}(n) \;=\; 4(n+1)H_n-7(n+1)+7 \;=\; 4nH_n-7n+4H_n$$

Surprise, surprise, we get exactly the same solution!

Exercises

Unless explicitly stated otherwise, you may assume there is a function RANDOM that, given any positive integer k as input, returns an integer chosen independently and uniformly at random from the set $\{1,2,\ldots,k\}$ as output, in $O(1)$ time. For example, to perform a fair coin flip, one could call RANDOM(2).

1. The following algorithm finds the second smallest element in an unsorted array:

```
RANDOM2NDMIN(A[1..n]):
  min1 ← ∞
  min2 ← ∞
  for i ← 1 to n in random order
      if A[i] < min1
          min2 ← min1          (⋆)
          min1 ← A[i]
      else if A[i] < min2
          min2 ← A[i]          (⋆⋆)
  return min2
```

(a) In the worst case, how many times does RANDOM2NDMIN execute line ($\star$)?

(b) What is the probability that line ($\star$) is executed during the nth iteration of the loop?

(c) What is the *exact* expected number of executions of line ($\star$)?

(d) In the worst case, how many times does RANDOM2NDMIN execute line ($\star\star$)?

(e) What is the probability that line ($\star\star$) is executed during the nth iteration of the loop?

(f) What is the *exact* expected number of executions of line ($\star\star$)?

2. Consider the following randomized algorithm for choosing the largest bolt. Draw a bolt uniformly at random from the set of n bolts, and draw a nut uniformly at random from the set of n nuts. If the bolt is smaller than the nut, discard the bolt, draw a new bolt uniformly at random from the unchosen bolts, and repeat. Otherwise, discard the nut, draw a new nut uniformly at random from the unchosen nuts, and repeat. Stop either when every nut has been discarded, or every bolt except the one in your hand has been discarded.

   What is the *exact* expected number of nut-bolt tests performed by this algorithm? Prove your answer is correct. *[Hint: What is the expected number of unchosen nuts and bolts when the algorithm terminates?]*

3. Let S be a set of n points in the plane. A point p in S is called *Pareto-optimal* if no other point in S is both above and to the right of p.

   (a) Describe and analyze a deterministic algorithm that computes the Pareto-optimal points in S in $O(n \log n)$ time.

   (b) Suppose each point in S is chosen independently and uniformly at random from the unit square $[0,1] \times [0,1]$. What is the *exact* expected number of Pareto-optimal points in S?

4. A *data stream* is an extremely long sequence of items that you can only read only once, in order. A good example of a data stream is the sequence of packets that pass through a router. Data stream algorithms must process each item in the stream quickly, using very little memory; there is simply too much data to store, and it arrives too quickly for any complex computations. Every data stream algorithm looks roughly like this:

```
DoSomethingInteresting(stream S):
  repeat
      x ← next item in S
      ⟨⟨do something fast with x⟩⟩
  until S ends
  return ⟨⟨something⟩⟩
```

(a) Describe and analyze an algorithm that chooses one element uniformly at random from a data stream, *without knowing the length of the stream in advance*. Your algorithm should spend $O(1)$ time per stream element and use $O(1)$ space (not counting the stream itself). You may assume that the stream is non-empty.

(b) Describe and analyze an algorithm that chooses k elements uniformly at random from a data stream, *without knowing the length of the stream in advance*. Your algorithm should spend $O(1)$ time per stream element and use $O(k)$ space (not counting the stream itself). You may assume that the stream has at least k elements.

5. Consider the following randomized variant of one-armed quicksort, which selects the kth smallest element in an unsorted array $A[1..n]$. As usual, $\textsc{Partition}(A[1..n], p)$ partitions the array A into three parts by comparing the pivot element $A[p]$ to every other element, using $n-1$ comparisons, and returns the new index of the pivot element.

```
QuickSelect(A[1..n], k):
  r ← Partition(A[1..n], Random(n))
  if k < r
      return QuickSelect(A[1..r−1], k)
  else if k > r
      return QuickSelect(A[r+1..n], k−r)
  else
      return A[k]
```

(a) State a recurrence for the expected running time of QuickSelect, as a function of n and k.

(b) What is the *exact* probability that QuickSelect compares the ith smallest and jth smallest elements in the input array? The correct answer is a simple function of i, j, and k. *[Hint: Check your answer by trying a few small examples.]*

(c) What is the *exact* probability that in one of the recursive calls to QuickSelect, the first argument is the subarray $A[i..j]$? The correct answer is a simple function of i, j, and k. *[Hint: Check your answer by trying a few small examples.]*

(d) Show that for any n and k, the expected running time of QuickSelect is $\Theta(n)$. You can use either the recurrence from part (a) or the probabilities from part (b) or (c). For extra credit, find the *exact* expected number of comparisons, as a function of n and k.

(e) What is the expected number of times that QuickSelect calls itself recursively?

6. Suppose we are given a two-dimensional array $M[1..n, 1..n]$ in which every row and every column is sorted in increasing order and no two elements are equal.

(a) Describe and analyze an algorithm to solve the following problem in $O(n)$ time: Given indices i, j, i', j' as input, compute the number of elements of M smaller than $M[i, j]$ and larger than $M[i', j']$.

(b) Describe and analyze an algorithm to solve the following problem in $O(n)$ time: Given indices i, j, i', j' as input, return an element of M chosen uniformly at random from the elements smaller than $M[i, j]$ and larger than $M[i', j']$. Assume the requested range is always non-empty.

(c) Describe and analyze a randomized algorithm to compute the median element of M in $O(n \log n)$ expected time.

7. Suppose we are given a *sorted* circular linked list of numbers, implemented as a pair of arrays, one storing the actual numbers and the other storing successor pointers. Specifically, we are given an array $X[1..n]$ of distinct real numbers and an array $next[1..n]$ be an array of indices with the following property:

   - If $X[i]$ is the largest element of X, then $X[next[i]]$ is the smallest element of X
   - Otherwise, $X[next[i]]$ is the smallest element of X that is larger than $X[i]$.

   For example:

i	1	2	3	4	5	6	7	8	9
$X[i]$	83	54	16	31	45	99	78	62	27
$next[i]$	6	8	9	5	2	3	1	7	4

   Describe and analyze a randomized algorithm that determines whether a given number x appears in the array X in $O(\sqrt{n})$ expected time ***without modifying the input arrays***.

8. Death knocks on your door one cold blustery morning and challenges you to a game. Death knows that you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with 4^n leaves, each colored either black or white. There is a token at the root of the tree. To play the game, you and Death will take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, you die; if it's white, you will live forever. You move first, so Death gets the last turn.

   You can decide whether it's worth playing or not as follows. Imagine that the nodes at even levels (where it's your turn) are OR gates, the nodes at odd levels (where it's Death's turn) are AND gates. Each gate gets its input from its children and passes its output to its parent. White and black stand for TRUE and FALSE. If the output at the top of the tree is TRUE, then you can win and live forever! If the output at the top of the tree is FALSE, you should challenge Death to a game of Twister instead.

(a) Describe and analyze a deterministic algorithm to determine whether or not you can win. *[Hint: This is easy!]*

(b) Unfortunately, Death won't give you enough time to look at every node in the tree. Describe a *randomized* algorithm that determines whether you can win in $O(3^n)$ expected time. *[Hint: Consider the case $n = 1$.]*

*(c) Describe and analyze a randomized algorithm that determines whether you can win in $O(c^n)$ expected time, for some constant $c < 3$. *[Hint: You may not need to change your algorithm from part (b) at all!]*

9. Death knocks on Dirk Gently's door one cold blustery morning and challenges him to a game. Emboldened by his experience with algorithms students, Death presents Dirk with a complete binary tree with 4^n leaves, each colored either black or white. There is a token at the root of the tree. To play the game, Dirk and Death will take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, Dirk dies; if it's white, Dirk lives forever. Dirk moves first, so Death gets the last turn.[3]

Unfortunately, Dirk slept through Death's explanation of the rules, so he decides to just play randomly. Whenever it's Dirk's turn, he flips a fair coin and moves left on heads, or right on tails, confident that the Fundamental Interconnectedness of All Things will keep him alive, unless it doesn't. Death plays much more purposefully, of course, always choosing the move that maximizes the probability that Dirk loses the game.

(a) Describe an algorithm that computes *the probability* that Dirk wins the game against Death.

(b) Realizing that Dirk is not taking the game seriously, Death gives up in desperation and decides to play randomly as well! Describe an algorithm that computes *the probability* that Dirk wins the game again Death, assuming *both* players flip fair coins to decide their moves.

10. A *majority tree* is a complete ternary tree with depth n, where every leaf is labeled either 0 or 1. The *value* of a leaf is its label; the *value* of any internal node is the majority of the values of its three children. Consider the problem of computing the value of the root of a majority tree, given the sequence of 3^n leaf labels as input. For example, if $n = 2$ and the leaves are labeled $1, 0, 0, 0, 1, 0, 1, 1, 1$, the root has value 0.

[3] Yes, this is precisely the same game from the previous exercise.

A majority tree with depth $n = 2$.

(a) Prove that *any* deterministic algorithm that computes the value of the root of a majority tree *must* examine every leaf. *[Hint: Consider the special case $n = 1$. Recurse.]*

(b) Describe and analyze a randomized algorithm that computes the value of the root in worst-case expected time $O(c^n)$ for some constant $c < 3$. *[Hint: Consider the special case $n = 1$. Recurse.]*

(c) Suppose each node in the given tree has five children instead of three. Describe and analyze a randomized algorithm that computes the value of the root in worst-case expected time $O(c^n)$ for some constant $c < 5$.

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

I thought the following four [rules] would be enough, provided that I made a firm and constant resolution not to fail even once in the observance of them. The first was never to accept anything as true if I had not evident knowledge of its being so. . . . The second, to divide each problem I examined into as many parts as was feasible, and as was requisite for its better solution. The third, to direct my thoughts in an orderly way. . . establishing an order in thought even when the objects had no natural priority one to another. And the last, to make throughout such complete enumerations and such general surveys that I might be sure of leaving nothing out.

— René Descartes, *Discours de la Méthode* (1637)

There are those who think that life has nothing left to chance
A host of holy horrors to direct our aimless dance

— Rush, "Freewill", *Permanent Waves* (1980), lyrics by Neal Peart

What is luck?
Luck is probability taken personally.
It is the excitement of bad math.

— Penn Jillette (2001), quoting Chip Denman (1998)

3 Randomized Binary Search Trees

In this lecture, we consider two randomized alternatives to balanced binary search tree structures such as AVL trees, red-black trees, B-trees, or splay trees, which are arguably simpler than any of these deterministic structures.

3.1 Treaps

3.1.1 Definitions

A *treap* is a binary tree in which every node has both a *search key* and a *priority*, where the inorder sequence of search keys is sorted and each node's priority is smaller than the priorities of its children.[1] In other words, a treap is simultaneously a binary search tree for the search keys and a (min-)heap for the priorities. In our examples, we will use letters for the search keys and numbers for the priorities.

A treap. Letters are search keys; numbers are priorities.

I'll assume from now on that all the keys and priorities are distinct. Under this assumption, we can easily prove by induction that the structure of a treap is completely determined by the

[1] Sometimes I hate English. Normally, 'higher priority' means 'more important', but 'first priority' is also more important than 'second priority'. Maybe 'posteriority' would be better; one student suggested 'unimportance'.

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

search keys and priorities of its nodes. Since it's a heap, the node v with highest priority must be the root. Since it's also a binary search tree, any node u with $key(u) < key(v)$ must be in the left subtree, and any node w with $key(w) > key(v)$ must be in the right subtree. Finally, since the subtrees are treaps, by induction, their structures are completely determined. The base case is the trivial empty treap.

Another way to describe the structure is that a treap is exactly the binary search tree that results by inserting the nodes one at a time into an initially empty tree, in order of increasing priority, using the standard textbook insertion algorithm. This characterization is also easy to prove by induction.

A third description interprets the keys and priorities as the coordinates of a set of points in the plane. The root corresponds to a T whose joint lies on the topmost point. The T splits the plane into three parts. The top part is (by definition) empty; the left and right parts are split recursively. This interpretation has some interesting applications in computational geometry, which (unfortunately) we won't have time to talk about.

A geometric interpretation of the same treap.

Treaps were first discovered by Jean Vuillemin in 1980, but he called them *Cartesian trees*.[2] The word 'treap' was first used by Edward McCreight around 1980 to describe a slightly different data structure, but he later switched to the more prosaic name *priority search trees*.[3] Treaps were rediscovered and used to build randomized search trees by Cecilia Aragon and Raimund Seidel in 1989.[4] A different kind of randomized binary search tree, which uses random rebalancing instead of random priorities, was later discovered and analyzed by Conrado Martínez and Salvador Roura in 1996.[5]

3.1.2 Treap Operations

The search algorithm is the usual one for binary search trees. The time for a successful search is proportional to the depth of the node. The time for an unsuccessful search is proportional to the depth of either its successor or its predecessor.

To insert a new node z, we start by using the standard binary search tree insertion algorithm to insert it at the bottom of the tree. At the point, the search keys still form a search tree, but the

[2] J. Vuillemin, A unifying look at data structures. *Commun. ACM* 23:229–239, 1980.

[3] E. M. McCreight. Priority search trees. *SIAM J. Comput.* 14(2):257–276, 1985.

[4] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica* 16:464–497, 1996.

[5] C. Martínez and S. Roura. Randomized binary search trees. *J. ACM* 45(2):288-323, 1998. The results in this paper are virtually identical (including the constant factors!) to the corresponding results for treaps, although the analysis techniques are quite different.

priorities may no longer form a heap. To fix the heap property, as long as z has smaller priority than its parent, perform a *rotation* at z, a local operation that decreases the depth of z by one and increases its parent's depth by one, while maintaining the search tree property. Rotations can be performed in constant time, since they only involve simple pointer manipulation.

A right rotation at x and a left rotation at y are inverses.

The overall time to insert z is proportional to the depth of z before the rotations—we have to walk down the treap to insert z, and then walk back up the treap doing rotations. Another way to say this is that the time to insert z is roughly twice the time to perform an unsuccessful search for $key(z)$.

Left to right: After inserting S with priority -1, rotate it up to fix the heap property.
Right to left: Before deleting S, rotate it down to make it a leaf.

To delete a node, we just run the insertion algorithm backward in time. Suppose we want to delete node z. As long as z is not a leaf, perform a rotation at the child of z with smaller priority. This moves z down a level and its smaller-priority child up a level. The choice of which child to rotate preserves the heap property everywhere except at z. When z becomes a leaf, chop it off.

We sometimes also want to *split* a treap T into two treaps $T_<$ and $T_>$ along some pivot key π, so that all the nodes in $T_<$ have keys less than π and all the nodes in $T_>$ have keys bigger then π. A simple way to do this is to insert a new node z with $key(z) = \pi$ and $priority(z) = -\infty$. After the insertion, the new node is the root of the treap. If we delete the root, the left and right sub-treaps are exactly the trees we want. The time to split at π is roughly twice the time to (unsuccessfully) search for π.

Similarly, we may want to *join* two treaps $T_<$ and $T_>$, where every node in $T_<$ has a smaller search key than any node in $T_>$, into one super-treap. Merging is just splitting in reverse—create a dummy root whose left sub-treap is $T_<$ and whose right sub-treap is $T_>$, rotate the dummy node down to a leaf, and then cut it off.

The cost of each of these operations is proportional to the depth of some node v in the treap.

- **Search:** A successful search for key k takes $O(depth(v))$ time, where v is the node with $key(v) = k$. For an unsuccessful search, let v^- be the inorder *predecessor* of k (the node whose key is just barely smaller than k), and let v^+ be the inorder *successor* of k (the node whose key is just barely larger than k). Since the last node examined by the binary search is either v^- or v^+, the time for an unsuccessful search is either $O(depth(v^+))$ or $O(depth(v^-))$.

- **Insert/Delete:** Inserting a new node with key k takes either $O(depth(v^+))$ time or $O(depth(v^-))$ time, where v^+ and v^- are the predecessor and successor of the new node. Deletion is just insertion in reverse.
- **Split/Join:** Splitting a treap at pivot value k takes either $O(depth(v^+))$ time or $O(depth(v^-))$ time, since it costs the same as inserting a new dummy root with search key k and priority $-\infty$. Merging is just splitting in reverse.

In the worst case, the depth of an n-node treap is $\Theta(n)$, so each of these operations has a worst-case running time of $\Theta(n)$.

3.1.3 Random Priorities

A *randomized treap* is a treap in which the priorities are *independently and uniformly distributed continuous random variables*. Whenever we insert a new search key into the treap, we generate a random real number between (say) 0 and 1 and use that number as the priority of the new node. The only reason we're using real numbers is so that the probability of two nodes having the same priority is zero, since equal priorities make the analysis slightly messier. (In practice, we could just choose random integers from a large range like 0 to $2^{31}-1$ and break ties arbitrarily; occasional ties have almost no practical effect on the performance of the data structure.) Also, since the priorities are independent, each node is equally likely to have the smallest priority.

The cost of all the operations we discussed—search, insert, delete, split, join—is proportional to the depth of some node in the tree. Here we'll see that the *expected* depth of *any* node is $O(\log n)$, which implies that the expected running time for any of those operations is also $O(\log n)$.

Let x_k denote the node with the kth smallest search key. To simplify notation, let us write $\boldsymbol{i \uparrow k}$ (read "i above k") to mean that x_i is a proper ancestor of x_k. Since the depth of v is just the number of proper ancestors of v, we have the following identity:

$$depth(x_k) = \sum_{i=1}^{n} [i \uparrow k].$$

(Again, we're using Iverson bracket notation.) Now we can express the *expected* depth of a node in terms of these indicator variables as follows.

$$\mathrm{E}[depth(x_k)] = \sum_{i=1}^{n} E\big[[i \uparrow k]\big] = \sum_{i=1}^{n} \Pr[i \uparrow k]$$

(Just as in our analysis of matching nuts and bolts, we're using linearity of expectation and the fact that $\mathrm{E}[X] = \Pr[X = 1]$ for any zero-one variable X; in this case, $X = [i \uparrow k]$.) So to compute the expected depth of a node, we just have to compute the probability that some node is a proper ancestor of some other node.

Fortunately, we can do this easily once we prove a simple structural lemma. Let $X(i,k)$ denote either the subset of treap nodes $\{x_i, x_{i+1}, \ldots, x_k\}$ or the subset $\{x_k, x_{k+1}, \ldots, x_i\}$, depending on whether $i < k$ or $i > k$. The order of the arguments is unimportant; the subsets $X(i,k)$ and $X(k,i)$ are identical. The subset $X(1,n) = X(n,1)$ contains all n nodes in the treap.

Lemma 1. *For all $i \neq k$, we have $i \uparrow k$ if and only if x_i has the smallest priority among all nodes in $X(i,k)$.*

Proof: There are four cases to consider.

If x_i is the root, then $i \uparrow k$, and by definition, it has the smallest priority of *any* node in the treap, so it must have the smallest priority in $X(i,k)$.

On the other hand, if x_k is the root, then $k \uparrow i$, so $i \not\uparrow k$. Moreover, x_i does not have the smallest priority in $X(i,k)$ — x_k does.

On the gripping hand[6], suppose some other node x_j is the root. If x_i and x_k are in different subtrees, then either $i < j < k$ or $i > j > k$, so $x_j \in X(i,k)$. In this case, we have both $i \not\uparrow k$ and $k \not\uparrow i$, and x_i does not have the smallest priority in $X(i,k)$ — x_j does.

Finally, if x_i and x_k are in the same subtree, the lemma follows from the inductive hypothesis (or, if you prefer, the Recursion Fairy), because the subtree is a smaller treap. The empty treap is the trivial base case. □

Since each node in $X(i,k)$ is equally likely to have smallest priority, we immediately have the probability we wanted:

$$\Pr[i \uparrow k] = \frac{[i \neq k]}{|k-i|+1} = \begin{cases} \dfrac{1}{k-i+1} & \text{if } i < k \\ 0 & \text{if } i = k \\ \dfrac{1}{i-k+1} & \text{if } i > k \end{cases}$$

To compute the expected depth of a node x_k, we just plug this probability into our formula and grind through the algebra.

$$\begin{aligned} \mathrm{E}[\mathit{depth}(x_k)] = \sum_{i=1}^{n} \Pr[i \uparrow k] &= \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^{n} \frac{1}{i-k+1} \\ &= \sum_{j=2}^{k} \frac{1}{j} + \sum_{i=2}^{n-k+1} \frac{1}{j} \\ &= H_k - 1 + H_{n-k+1} - 1 \\ &< \ln k + \ln(n-k+1) - 2 \\ &< 2\ln n - 2. \end{aligned}$$

In conclusion, every search, insertion, deletion, split, and join operation in an n-node randomized binary search tree takes $O(\log n)$ expected time.

Since a treap is exactly the binary tree that results when you insert the keys in order of increasing priority, a randomized treap is the result of inserting the keys in *random* order. So our analysis also automatically gives us the expected depth of any node in a binary tree built by random insertions (without using priorities).

3.1.4 Randomized Quicksort (Again!)

We've already seen two completely different ways of describing randomized quicksort. The first is the familiar recursive one: choose a random pivot, partition, and recurse. The second is a less familiar iterative version: repeatedly choose a new random pivot, partition whatever subset contains it, and continue. But there's a third way to describe randomized quicksort, this time in terms of binary search trees.

[6] See Larry Niven and Jerry Pournelle, *The Gripping Hand*, Pocket Books, 1994.

```
RandomizedQuicksort:
  T ← an empty binary search tree
  insert the keys into T in random order
  output the inorder sequence of keys in T
```

Our treap analysis tells us is that this algorithm will run in $O(n \log n)$ expected time, since each key is inserted in $O(\log n)$ expected time.

Why is this quicksort? Just like last time, all we've done is rearrange the order of the comparisons. Intuitively, the binary tree is just the recursion tree created by the normal version of quicksort. In the recursive formulation, we compare the initial pivot against everything else and then recurse. In the binary tree formulation, the first "pivot" becomes the root of the tree without any comparisons, but then later as each other key is inserted into the tree, it is compared against the root. Either way, the first pivot chosen is compared with everything else. The partition splits the remaining items into a left subarray and a right subarray; in the binary tree version, these are exactly the items that go into the left subtree and the right subtree. Since both algorithms define the same two subproblems, by induction, both algorithms perform the same comparisons.

We even saw the probability $1/(|k-i|+1)$ before, when we were talking about sorting nuts and bolts with a variant of randomized quicksort. In the more familiar setting of sorting an array of numbers, the probability that randomized quicksort compares the ith largest and kth largest elements is exactly $2/(|k-i|+1)$. The binary tree version of quicksort compares x_i and x_k if and only if $i \uparrow k$ or $k \uparrow i$, so the probabilities are exactly the same.

3.2 Skip Lists

Skip lists, first discovered by Bill Pugh in the late 1980's,[7] have many of the usual desirable properties of balanced binary search trees, but their superficial structure is very different.

At a high level, a skip list is just a sorted linked list with some random shortcuts. To do a search in a normal singly-linked list of length n, we obviously need to look at n items in the worst case. To speed up this process, we can make a second-level list that contains roughly half the items from the original list. Specifically, for each item in the original list, we duplicate it with probability $1/2$. We then string together all the duplicates into a second sorted linked list, and add a pointer from each duplicate back to its original. Just to be safe, we also add sentinel nodes at the beginning and end of both lists.

A linked list with some randomly-chosen shortcuts.

Now we can find a value x in this augmented structure using a two-stage algorithm. First, we scan for x in the shortcut list, starting at the $-\infty$ sentinel node. If we find x, we're done. Otherwise, we reach some value bigger than x and we know that x is not in the shortcut list. Let w be the largest item less than x in the shortcut list. In the second phase, we scan for x in the original list, starting from w. Again, if we reach a value bigger than x, we know that x is not in the data structure.

Since each node appears in the shortcut list with probability $1/2$, the expected number of nodes examined in the first phase is at most $n/2$. Only one of the nodes examined in the second

[7] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM* 33(6):668–676, 1990.

Searching for 5 in a list with shortcuts.

phase has a duplicate. The probability that any node is followed by k nodes without duplicates is 2^{-k}, so the expected number of nodes examined in the second phase is at most $1 + \sum_{k \geq 0} 2^{-k} = 2$. Thus, by adding these random shortcuts, we've reduced the cost of a search from n to $n/2 + 2$, roughly a factor of two in savings.

3.2.1 Recursive Random Shortcuts

Now there's an obvious improvement—add shortcuts to the shortcuts, and repeat recursively. That's exactly how skip lists are constructed. For each node in the original list, we repeatedly flip a coin until we get tails. Each time we get heads, we make a new copy of the node. The duplicates are stacked up in levels, and the nodes on each level are strung together into sorted linked lists. Each node v stores a search key $key(v)$, a pointer $down(v)$ to its next lower copy, and a pointer $right(v)$ to the next node in its level.

A skip list is a linked list with recursive random shortcuts.

The search algorithm for skip lists is very simple. Starting at the leftmost node L in the highest level, we scan through each level as far as we can without passing the target value x, and then proceed down to the next level. The search ends when we either reach a node with search key x or fail to find x on the lowest level.

```
SkipListFind(x, L):
  v ← L
  while (v ≠ Null and key(v) ≠ x)
      if key(right(v)) > x
          v ← down(v)
      else
          v ← right(v)
  return v
```

Intuitively, since each level of the skip lists has about half the number of nodes as the previous level, the total number of levels should be about $O(\log n)$. Similarly, each time we add another level of random shortcuts to the skip list, we cut the search time roughly in half, except for a constant overhead, so $O(\log n)$ levels should give us an overall expected search time of $O(\log n)$. Let's formalize each of these two intuitive observations.

Searching for 5 in a skip list.

3.2.2 Number of Levels

The actual values of the search keys don't affect the skip list analysis, so let's assume the keys are the integers 1 through n. Let $L(x)$ be the number of levels of the skip list that contain some search key x, not counting the bottom level. Each new copy of x is created with probability $1/2$ from the previous level, essentially by flipping a coin. We can compute the expected value of $L(x)$ recursively—with probability $1/2$, we flip tails and $L(x) = 0$; and with probability $1/2$, we flip heads, increase $L(x)$ by one, and recurse:

$$E[L(x)] = \frac{1}{2} \cdot 0 + \frac{1}{2}\big(1 + E[L(x)]\big)$$

Solving this equation gives us $\boldsymbol{E[L(x)] = 1}$.

In order to analyze the expected worst-case cost of a search, however, we need a bound on the *number of levels* $L = \max_x L(x)$. Unfortunately, we can't compute the average of a maximum the way we would compute the average of a sum. Instead, we derive a stronger result: ***The depth of a skip list storing n keys is $O(\log n)$ with high probability.*** "High probability" is a technical term that means the probability is at least $1 - 1/n^c$ for some constant $c \geq 1$; the hidden constant in the $O(\log n)$ bound could depend on c.

In order for a search key x to appear on level ℓ, it must have flipped ℓ heads in a row when it was inserted, so $\boldsymbol{\Pr[L(x) \geq \ell] = 2^{-\ell}}$. The skip list has at least ℓ levels if and only if $L(x) \geq \ell$ for at least one of the n search keys.

$$\Pr[L \geq \ell] = \Pr\big[(L(1) \geq \ell) \vee (L(2) \geq \ell) \vee \cdots \vee (L(n) \geq \ell)\big]$$

Using the ***union bound*** — $\Pr[A \vee B] \leq \Pr[A] + \Pr[B]$ for any random events A and B — we can simplify this as follows:

$$\Pr[L \geq \ell] \;\leq\; \sum_{x=1}^{n} \Pr[L(x) \geq \ell] \;=\; n \cdot \Pr[L(x) \geq \ell] \;=\; \frac{n}{2^\ell}.$$

When $\ell \leq \lg n$, this bound is trivial. However, for any constant $c > 1$, we have a strong upper bound

$$\Pr[L \geq c \lg n] \leq \frac{1}{n^{c-1}}.$$

We conclude that ***with high probability, a skip list has $O(\log n)$ levels.***

This high-probability bound indirectly implies a bound on the *expected* number of levels. Some simple algebra gives us the following alternate definition for expectation:

$$\mathrm{E}[L] = \sum_{\ell \geq 0} \ell \cdot \Pr[L = \ell] = \sum_{\ell \geq 1} \Pr[L \geq \ell]$$

Clearly, if $\ell < \ell'$, then $\Pr[L(x) \geq \ell] > \Pr[L(x) \geq \ell']$. So we can derive an upper bound on the expected number of levels as follows:

$$\begin{aligned}
\mathrm{E}[L(x)] = \sum_{\ell \geq 1} \Pr[L \geq \ell] &= \sum_{\ell=1}^{\lg n} \Pr[L \geq \ell] \;+ \sum_{\ell \geq \lg n + 1} \Pr[L \geq \ell] \\
&\leq \sum_{\ell=1}^{\lg n} 1 \;+ \sum_{\ell \geq \lg n + 1} \frac{n}{2^\ell} \\
&= \lg n + \sum_{i \geq 1} \frac{1}{2^i} && [i = \ell - \lg n] \\
&= \mathbf{\lg n + 2}
\end{aligned}$$

So in expectation, a skip list has *at most two* more levels than an ideal version where each level contains exactly half the nodes of the next level below. Notice that this is an *additive* penalty over a perfectly balanced structure, as opposed to treaps, where the expected depth is a constant *multiple* of the ideal $\lg n$.

3.2.3 Logarithmic Search Time

It's a little easier to analyze the cost of a search if we imagine running the algorithm backwards. SkipListFind takes the output from SkipListFind as input and traces back through the data structure to the upper left corner. Skip lists don't really have up and left pointers, but we'll pretend that they do so we don't have to write '$v \to \mathit{down}(v)$' or '$v \to \mathit{right}(v)$'.[8]

```
SkipListFind(v):
  while (level(v) ≠ L)
      if up(v) exists
          v ← up(v)
      else
          v ← left(v)
```

Now for *every* node v in the skip list, $up(v)$ exists with probability $1/2$. So for purposes of analysis, SkipListFind is equivalent to the following algorithm:

```
FlipWalk(v):
  while (v ≠ L)
      if CoinFlip = Heads
          v ← up(v)
      else
          v ← left(v)
```

[8] Leonardo da Vinci wrote all his notes using mirror-writing, but not because he wanted to keep his discoveries secret. He just had really bad arthritis in his right hand!

Obviously, the expected number of heads is exactly the same as the expected number of TAILS. Thus, the expected running time of this algorithm is twice the expected number of upward jumps. But we already know that the number of upward jumps is $O(\log n)$ with high probability. It follows the running time of FLIPWALK is $O(\log n)$ with high probability (and therefore in expectation).

Exercises

1. Prove that a treap is exactly the binary search tree that results from inserting the nodes one at a time into an initially empty tree, in order of increasing priority, using the standard textbook insertion algorithm.

2. Consider a treap T with n vertices. As in the notes above, identify nodes in T by the ranks of their search keys; thus, 'node 5' means the node with the 5th smallest search key. Let i, j, and k be integers such that $1 \leq i \leq j \leq k \leq n$.

   (a) Prove that the expected number of proper descendants of any node in a treap is exactly equal to the expected depth of that node.

   (b) The *left spine* of a binary tree is a path starting at the root and following only left-child pointers. What is the expected number of nodes in the left spine of T?

   (c) What is the expected number of leaves in T? *[Hint: What is the probability that node k is a leaf?]*

   (d) What is the expected number of nodes in T with two children?

   (e) What is the expected number of nodes in T with exactly one child?

   *(f) What is the expected number of nodes in T with exactly one *grandchild*?

   (g) Define the *priority rank* of a node in T to be one more than the number of nodes with smaller priority. For example, the root of T always has priority rank 1, and one of the children of the root has priority rank 2. What is the expected priority rank of node i?

   (h) What is the expected priority rank of the left child of the root (given that such a node exists)?

   *(i) What is the expected priority rank of the leftmost grandchild of the root (given that such a node exists)?

   *(j) What is the expected priority rank of a node with depth d?

   (k) What is the *exact* probability that node j is a common ancestor of node i and node k?

   (l) What is the *exact* expected length of the unique path in T from node i to node k?

   (m) What is the expected (key) rank of the leftmost leaf in T?

   (n) What is the expected (key) rank of the leftmost node in T with two children (given that such a node exists)?

   (o) What is the probability that T has no nodes with two children?

3. Recall that a *priority search tree* is a binary tree in which every node has both a *search key* and a *priority*, arranged so that the tree is simultaneously a binary search tree for the keys and a min-heap for the priorities. A ***heater*** is a priority search tree in which the *priorities*

are given by the user, and the *search keys* are distributed uniformly and independently at random in the real interval $[0, 1]$. Intuitively, a heater is a sort of anti-treap.[9]

The following problems consider an n-node heater T whose priorities are the integers from 1 to n. We identify nodes in T by their *priorities*; thus, 'node 5' means the node in T with *priority* 5. For example, the min-heap property implies that node 1 is the root of T. Finally, let i and j be integers with $1 \leq i < j \leq n$.

(a) Prove that in a random permutation of the $(i+1)$-element set $\{1, 2, \ldots, i, j\}$, elements i and j are adjacent with probability $2/(i+1)$.

(b) Prove that node i is an ancestor of node j with probability exactly $2/(i+1)$. *[Hint: Use part (a)!]*

(c) What is the exact probability that node i is a *descendant* of node j? *[Hint: Don't use part (a)!]*

(d) What is the *exact* expected depth of node j?

(e) Describe and analyze an algorithm to insert a new item into a heater. Express the expected running time of the algorithm in terms of the rank of the newly inserted item.

(f) Describe an algorithm to delete the minimum-priority item (the root) from an n-node heater. What is the expected running time of your algorithm?

4. Prove the following basic facts about skip lists, where n is the number of keys.

(a) The expected number of nodes is $O(n)$.

(b) The number of nodes is $O(n)$ with high probability.

(c) A new key can be inserted in $O(\log n)$ time with high probability.

(d) A key can be deleted in $O(\log n)$ time with high probability.

5. Suppose we are given two skip lists, one storing a set A of m keys, and the other storing a set B of n keys. Describe and analyze an algorithm to merge these into a single skip list storing the set $A \cup B$ in $O(n+m)$ expected time. Do *not* assume that every key in A is smaller than every key in B; the two sets could be arbitrarily intermixed. *[Hint: Do the obvious thing.]*

6. Any skip list $\mathcal{L}$ can be transformed into a binary search tree $T(\mathcal{L})$ as follows. The root of $T(\mathcal{L})$ is the leftmost node on the highest non-empty level of $\mathcal{L}$; the left and right subtrees are constructed recursively from the nodes to the left and to the right of the root. Let's call the resulting tree $T(\mathcal{L})$ a *skip list tree*.

(a) Show that any search in $T(\mathcal{L})$ is no more expensive than the corresponding search in $\mathcal{L}$. (Searching in $T(\mathcal{L})$ could be *considerably* cheaper—why?)

(b) Describe an algorithm to insert a new search key into a skip list tree in $O(\log n)$ expected time. Inserting key x into the tree $T(\mathcal{L})$ should produce *exactly* the same tree as inserting x into the skip list $\mathcal{L}$ and then transforming $\mathcal{L}$ into a tree. *[Hint: You need to maintain some additional information in the tree nodes.]*

[9] If you choose not to decide, you still have made a choice.

(c) Describe an algorithm to delete a search key from a skip list tree in $O(\log n)$ expected time. Again, deleting key x from $T(\mathcal{L})$ should produce *exactly* the same tree as deleting x from $\mathcal{L}$ and then transforming $\mathcal{L}$ into a tree.

7. Consider the following "loose" variant of treaps. Instead of generating priorities uniformly at random, for each node, we flip an independent fair coin until it comes up heads, and define the priority of the node to be the number of flips. Thus, for every positive integer k, a node has priority k with probability 2^{-k}. In addition, we invert the heap property, by requiring that the priority of any node is *not* larger than the priority of its parent.

   This method creates many nodes with the same priority; in particular, about half the nodes will have priority 1. Thus, a single set of search keys and priorities may be consistent with many different loose treaps.

   Prove that the expected depth of any node in an n-node "loose treap" is $O(\log n)$. To be pedantic, the expectation is over the random choice of priorities, but for each choice of priorities, we consider the worst possible loose treap.

   [Hint: This is almost exactly the same as the previous question.]

*8. In the usual theoretical presentation of treaps, the priorities are random real numbers chosen uniformly from the interval $[0, 1]$. In practice, however, computers have access only to random *bits*. This problem asks you to analyze an implementation of treaps that takes this limitation into account.

   Suppose the priority of a node v is abstractly represented as an infinite sequence $\pi_v[1..\infty]$ of random bits, which is interpreted as the rational number

$$priority(v) = \sum_{i=1}^{\infty} \pi_v[i] \cdot 2^{-i}.$$

   However, only a finite number ℓ_v of these bits are actually known at any given time. When a node v is first created, *none* of the priority bits are known: $\ell_v = 0$. We generate (or "reveal") new random bits only when they are necessary to compare priorities. The following algorithm compares the priorities of any two nodes in $O(1)$ expected time:

```
LargerPriority(v, w):
  for i ← 1 to ∞
      if i > ℓ_v
          ℓ_v ← i;  π_v[i] ← RandomBit
      if i > ℓ_w
          ℓ_w ← i;  π_w[i] ← RandomBit
      if π_v[i] > π_w[i]
          return v
      else if π_v[i] < π_w[i]
          return w
```

   Suppose we insert n items one at a time into an initially empty treap. Let $L = \sum_v \ell_v$ denote the total number of random bits generated by calls to LargerPriority during these insertions.

(a) Prove that $E[L] = \Theta(n)$.

(b) Prove that $E[\ell_v] = \Theta(1)$ for any node v. *[Hint: This is equivalent to part (a). Why?]*

(c) Prove that $E[\ell_{\text{root}}] = \Theta(\log n)$. *[Hint: Why doesn't this contradict part (b)?]*

9. A ***meldable priority queue*** stores a set of keys from some totally-ordered universe (such as the integers) and supports the following operations:

- MakeQueue: Return a new priority queue containing the empty set.
- FindMin(Q): Return the smallest element of Q (if any).
- DeleteMin(Q): Remove the smallest element in Q (if any).
- Insert(Q, x): Insert element x into Q, if it is not already there.
- DecreaseKey(Q, x, y): Replace an element $x \in Q$ with a smaller key y. (If $y > x$, the operation fails.) The input is a pointer directly to the node in Q containing x.
- Delete(Q, x): Delete the element $x \in Q$. The input is a pointer directly to the node in Q containing x.
- Meld(Q_1, Q_2): Return a new priority queue containing all the elements of Q_1 and Q_2; this operation destroys Q_1 and Q_2.

A simple way to implement such a data structure is to use a heap-ordered binary tree, where each node stores a key, along with pointers to its parent and two children. Meld can be implemented using the following randomized algorithm:

```
MELD(Q1, Q2):
  if Q1 is empty return Q2
  if Q2 is empty return Q1

  if key(Q1) > key(Q2)
      swap Q1 ↔ Q2

  with probability 1/2
      left(Q1) ← MELD(left(Q1), Q2)
  else
      right(Q1) ← MELD(right(Q1), Q2)

  return Q1
```

(a) Prove that for *any* heap-ordered binary trees Q_1 and Q_2 (not just those constructed by the operations listed above), the expected running time of Meld(Q_1, Q_2) is $O(\log n)$, where $n = |Q_1| + |Q_2|$. *[Hint: What is the expected length of a random root-to-leaf path in an n-node binary tree, where each left/right choice is made with equal probability?]*

(b) Prove that Meld(Q_1, Q_2) runs in $O(\log n)$ time with high probability.

(c) Show that each of the other meldable priority queue operations can be implemented with at most one call to Meld and $O(1)$ additional time. (This implies that every operation takes $O(\log n)$ time with high probability.)

*10. Our probabilistic analysis of treaps and skip lists assumes that the sequence of operations used to query and update the data structures are independent of the random choices used to build the data structure. However, if a malicious adversary has additional information about the data structure, he can force significantly worse performance.

(a) Let $\pi[0..n^2]$ be an arbitrary permutation of the numbers $0, 1, 2, \ldots, n^2$. For each index i, define two integers:

- a_i is the length of the longest increasing subsequence of π that ends with $\pi[i]$.
- b_i be the length of the longest decreasing subsequence of π that starts with $\pi[i]$.

Argue that the $n^2 + 1$ ordered pairs (a_i, b_i) are all distinct.

(b) Prove the ***Erdős-Szekeres Theorem***: Any permutation $\pi[0..n^2]$ of the numbers $0, 1, 2, \ldots, n^2$ contains either an increasing subsequence of length $n+1$ or a decreasing subsequence of length $n + 1$.

(c) Suppose after constructing a skip list with n items, the adversary discovers the number of levels that contain each item. Prove that by deleting a subset of the nodes, the adversary can force the maximum search time to be $\Omega(\sqrt{n})$.

(d) Suppose after constructing a skip list with n items, the adversary discovers the number of levels that contain each item. Prove that by deleting a subset of the nodes, the adversary can force the maximum search time to be $\Omega(n)$ *with high probability*.

(e) Suppose after constructing a treap with n items, the adversary discovers the random priorities of each item. Prove that by deleting a subset of the nodes, the adversary can force the maximum depth of the treap to be $\Omega(\sqrt{n})$.

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

Popular tradition may be used to defend what seems irrational, and you can also say that sometimes it is not irrational, for it is likely that unlikely things should happen.

— Aristotle, *Poetics* (c.335 BCE)

But, on the other hand, Uncle Abner said that the person that had took a bull by the tail once had learnt sixty or seventy times as much as a person that hadn't, and said a person that started in to carry a cat home by the tail was gitting knowledge that was always going to be useful to him, and warn't ever going to grow dim or doubtful.

— Mark Twain, *Tom Sawyer Abroad* (1894)

*4 Tail Inequalities

The simple recursive structure of skip lists made it relatively easy to derive an upper bound on the expected *worst-case* search time, by way of a stronger high-probability upper bound on the worst-case search time. We can prove similar results for treaps, but because of the more complex recursive structure, we need slightly more sophisticated probabilistic tools. These tools are usually called ***tail inequalities***; intuitively, they bound the probability that a random variable with a bell-shaped distribution takes a value in the *tails* of the distribution, far away from the mean.

4.1 Markov's Inequality

Perhaps the simplest tail inequality was named after the Russian mathematician Andrey Markov; however, in strict accordance with Stigler's Law of Eponymy, it first appeared in the works of Markov's probability teacher, Pafnuty Chebyshev.[1]

Markov's Inequality. *Let Z be a non-negative integer random variable. For any real number $z > 0$, we have* $\Pr[Z \geq z] \leq \mathrm{E}[Z]/z$.

Proof: Suppose we plot $\Pr[Z \geq z]$ as a function of z, as in the figure below. Because Z takes only integer values, the resulting curve consists of horizontal and vertical line segments. By splitting the region between this curve and the coordinate axes into horizontal rectangles, we see that the area of this region is exactly $\sum_z \Pr[Z \geq z] = \sum_z z \cdot \Pr[Z = z] = \mathrm{E}[Z]$. On the other hand, for any particular value of z, the rectangle with width z and height $\Pr[Z \geq z]$ (shaded darker in the figure) fits entirely under the curve. We conclude that $z \cdot \Pr[Z \geq z] \leq \mathrm{E}[Z]$. □

Proof of Markov's inequality.

[1]The closely related tail bound traditionally called Chebyshev's inequality was actually discovered by the French statistician Irénée-Jules Bienaymé, a friend and colleague of Chebyshev's. Just to be extra confusing, some sources refer to what we're calling Markov's inequality as "Chebyshev's inequality" or "Bienaymé's inequality".

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

In particular, Markov's inequality implies the following bound on the probability that any random variable X is significantly larger than its expectation:

$$\Pr[X \geq (1+\delta)\,\mathrm{E}[X]] \leq \frac{1}{1+\delta}.$$

Unfortunately, the bounds that Markov's inequality (directly) implies are generally too weak to be useful. (For example, Markov's inequality implies that with high probability, every node in an n-node treap has depth $O(n^2 \log n)$. Well, *duh!*) To get stronger bounds, we need to exploit some additional structure in our random variables.

4.2 Independence

Two random variables are ***independent*** if knowing the value of one variable gives us no additional knowledge about the distribution of the other. More formally, X and Y are and only if

$$\Pr[X = x \;\wedge\; Y = y] = \Pr[X = x] \cdot \Pr[Y = y],$$

or equivalently,

$$\Pr[X = x \mid Y = y] = \Pr[X = x]$$

for all possible values x and y. For example, two flips of the same (ideal) fair coin are independent, but the number of heads and the number of tails in a sequence of n coin flips are not independent (because they must add to n). Variables that are not independent are called ***dependent***. Independence of X and Y has two important consequences.

- The expectation of the product of two independent random variables is equal to the product of their expectations $\mathrm{E}[X \cdot Y] = \mathrm{E}[X] \cdot \mathrm{E}[Y]$.
- If X and Y are independent, then for any function f, the random variables $f(X)$ and $f(Y)$ are also independent.

Neither of these properties hold for dependent random variables.

More generally, a collection of random variables $X_1, X_2, \ldots, X_n$ are said to be ***mutually independent*** (or ***fully independent***) if and only if

$$\Pr\left[\bigwedge_{i=1}^{n}(X_i = x_i)\right] = \prod_{i=1}^{n}\Pr[X_i = x_i]$$

for all possible values $x_1, x_2, \ldots, x_n$. Mutual independence of the X_i's implies that the expectation of the product of the X_i's is equal to the product of the expectations:

$$\mathrm{E}\left[\prod_{i=1}^{n} X_i\right] = \prod_{i=1}^{n}\mathrm{E}[X_i].$$

Finally, if $X_1, X_2, \ldots, X_n$ are mutually independent, then for any function f, the random variables $f(X_1), f(X_2), \ldots, f(X_n)$ are also mutually independent.

In some contexts, especially in the analysis of hashing, we can only realistically assume a limited form of independence. A set of random variables is ***pairwise independent*** if every pair of variables in the set is independent. More generally, for any positive integer k, we say that a set of random variables is ***k-wise independent*** if every subset of size k is mutually independent. A set of variables is fully independent if and only if it is k-wise independent for *all* k.

Every k-wise independent set of random variables is also j-wise independent for all $j < k$, but the converse is not true. For example, let X, Y, and Z be independent random bits, and let $W = (X + Y + Z) \bmod 2$. The set of random variables $\{W, X, Y, Z\}$ is 3-wise independent but not 4-wise (or fully) independent.

4.3 Chebyshev's Inequality

Most of our analysis of randomized algorithms and data structures boils down to analyzing sums of indicator variables.[2] So far we have only been interested in the expected value of these sums, but we can prove stronger conditions under various assumptions about independence.

Consider a collection $X_1, X_2, \ldots, X_n$ of random bits, and for each index i, let $p_i = \mathrm{E}[X_i] = \Pr[X_i = 1]$. Let X denote the sum $\sum_i X_i$ and let $\mu = \mathrm{E}[X]$; linearity of expectation immediately implies that $\mu = \sum_i p_i$.

Chebyshev's Inequality. *If the indicator variables $X_1, X_2, \ldots, X_n$ are pairwise independent, then $\Pr[(X-\mu)^2 \geq z] < \mu/z$ for all $z > 0$.*

Proof: For each index i, let $Y_i := X_i - p_i$, and let $Y := \sum_i Y_i = X - \mu$.

$$
\begin{aligned}
\mathrm{E}[Y^2] &= \mathrm{E}\Big[\sum_{i,j} Y_i Y_j\Big] && \text{[definition of } Y\text{]} \\
&= \sum_{i,j} \mathrm{E}[Y_i Y_j] && \text{[linearity]} \\
&= \sum_i \mathrm{E}[Y_i^2] + \sum_{i \neq j} \mathrm{E}[Y_i Y_j] \\
&= \sum_i \mathrm{E}[Y_i^2] + \sum_{i \neq j} \mathrm{E}[Y_i] \cdot \mathrm{E}[Y_j] && \text{[pairwise independence]} \\
&= \sum_i \mathrm{E}[Y_i^2] + 0 && \text{[}\mathrm{E}[Y_i] = 0\text{, by linearity]} \\
&= \sum_i \big((p_i(1-p_i)^2 + (1-p_i)(0-p_i)^2\big) && \text{[definition of E]} \\
&= \sum_i p_i(1-p_i) \;<\; \sum_i p_i \;=\; \mu
\end{aligned}
$$

Because the random variable Y^2 is non-negative, Markov's inequality now directly implies the bound $\Pr[Y^2 \geq z] \leq \mathrm{E}[Y^2]/z < \mu/z$, which completes the proof. □

Chebyshev's inequality immediately gives us significantly tighter bounds than Markov's inequality when the component indicator variables X_i are pairwise independent. The following bounds hold for all positive real numbers Δ and δ:

$$
\boxed{
\begin{array}{ll}
\Pr[X \geq \mu + \Delta] < \dfrac{\mu}{\Delta^2} & \Pr[X \geq (1+\delta)\mu] < \dfrac{1}{\delta^2 \mu} \\[2ex]
\Pr[X \leq \mu - \Delta] < \dfrac{\mu}{\Delta^2} & \Pr[X \leq (1-\delta)\mu] < \dfrac{1}{\delta^2 \mu}
\end{array}
}
$$

[2] This focus on sums of indicator variables is a feature (or if you prefer, a handicap) of these lecture notes, not of randomized algorithm analysis more broadly!

The inequalities on the left are called *additive* tail bounds; the inequalities on the right are called *multiplicative* tail bounds. The inequalities in the top row bounds the *upper tail* of X's probability distribution; the inequalities in the bottom row bound the *lower tail*.

An important consequence of Chebyshev's inequality is the so-called ***Law of Large Numbers***, which states that if we repeat the same experiment many times, the statistical average of the outcomes tends toward the expected value of a single experiment with overwhelming probability. More formally, let $X_1, X_2, X_3, \ldots$ be independent random bits, with $\Pr[X_i = 1] = p$ for all i, and for any index n, let $\overline{X}_n = \sum_{i=1}^{n} X_i/n$ denote the mean of the first n bits. The ***weak*** law of large numbers states that

$$\lim_{n\to\infty} \Pr\left[|\overline{X}_n - p| > \varepsilon\right] = 0$$

for any real $\varepsilon > 0$; this law follows almost immediately from Chebyshev's inequality. The ***strong*** law of large numbers states that

$$\Pr\left[\lim_{n\to\infty} \overline{X}_n = p\right] = 1;$$

the proof of this law is considerably more involved.

4.4 Higher Moment Inequalities

Chebyshev's inequality and its corollaries are special cases of a more general family of so-called ***moment inequalities***, which consider the distribution of higher even powers of the random variable X under stronger assumptions about the independence of the components X_i. Here I'll state the most general result without proof; the derivation follows the same general outline as the proof of Chebyshev's inequality above, but with more, you know, math.

General Moment Inequality. *For any fixed integer $k > 0$, if the variables $X_1, X_2, \ldots, X_n$ are $2k$-wise independent, then $\Pr[(X-\mu)^k \geq z] = O(\mu^k/z)$ for all $z > 0$. The hidden constant in the $O(\cdot)$ notation depends on k.*

This inequality immediately implies the following asymptotic tail bounds for all positive real numbers Δ and δ; again, the hidden $O(\cdot)$ constants depend on the fixed independence parameter k.

$$\boxed{\begin{array}{ll} \Pr[X \geq \mu + \Delta] = O\left(\left(\dfrac{\mu}{\Delta^2}\right)^k\right) & \Pr[X \geq (1+\delta)\mu] = O\left(\left(\dfrac{1}{\delta^2\mu}\right)^k\right) \\[2ex] \Pr[X \leq \mu - \Delta] = O\left(\left(\dfrac{\mu}{\Delta^2}\right)^k\right) & \Pr[X \geq (1-\delta)\mu] = O\left(\left(\dfrac{1}{\delta^2\mu}\right)^k\right) \end{array}}$$

In short, the more independence we can assume about the indicator variables X_i, the higher the degree of the polynomial tails in the distribution of X, and the more likely that X stays close to its expected value.

4.5 Chernoff Bounds

For *fully* independent random variables, even tighter tail bounds were developed in the early 1950s by Herman Chernoff, who was then a mathematics professor at the University of Illinois. However, in strict accordance with Stigler's Law, the first published version of "Chernoff bounds",

which appeared in a 1952 paper by Chernoff that gave the bounds their name, were actually due to his colleague Herman Rubin.[3]

Exponential moment inequality. *If the indicator variables $X_1, X_2, \ldots, X_n$ are fully independent, then $E[\alpha^X] \leq e^{(\alpha-1)\mu}$ for any $\alpha \geq 1$.*

Proof: The definition of expectation immediately implies that

$$E[\alpha^{X_i}] \;=\; p_i\alpha^1 + (1-p_i)\alpha^0 \;=\; (\alpha-1)p_i + 1.$$

Thus, by The World's Most Useful Inequality $1 + t \leq e^t$, we have

$$E[\alpha^{X_i}] \;\leq\; e^{(\alpha-1)p_i}.$$

Full independence of the X_i's now immediately implies

$$E[\alpha^X] \;=\; \prod_i E[\alpha^{X_i}] \;\leq\; \prod_i e^{(\alpha-1)p_i} \;=\; e^{(\alpha-1)\mu}$$

and we're done. □

Chernoff bound (upper tail). *If the indicator variables $X_1, X_2, \ldots, X_n$ are fully independent, then $\Pr[X \geq x] \leq e^{x-\mu}(\mu/x)^x$ for all $x \geq \mu$.*

Proof: Consider any fixed value $x \geq \mu$. The function $t \mapsto (x/\mu)^t$ is monotonically increasing, so

$$\Pr[X \geq x] = \Pr\left[\left(\frac{x}{\mu}\right)^X \geq \left(\frac{x}{\mu}\right)^x\right].$$

Markov's inequality implies that

$$\Pr\left[\left(\frac{x}{\mu}\right)^X \geq \left(\frac{x}{\mu}\right)^x\right] \;\leq\; \frac{E[(x/\mu)^X]}{(x/\mu)^x}.$$

Finally, applying the exponential moment inequality at $\alpha = x/\mu$ gives us

$$E[(x/\mu)^X] \leq e^{((x/\mu)-1)\mu} = e^{x-\mu},$$

which completes the proof. □

A nearly identical argument implies a similar bound on the lower tail:

Chernoff bound (lower tail). *If the indicator variables $X_1, X_2, \ldots, X_n$ are fully independent, then $\Pr[X \leq x] \leq e^{x-\mu}(\mu/x)^x$ for all $x \leq \mu$.*

Proof: For any $x \leq \mu$, we immediately have $\Pr[X \leq x] = \Pr[(x/\mu)^X \geq (x/\mu)^x]$. (The direction of the inequality changes because $x/\mu \leq 1$.) The remainder of the proof is unchanged. □

[3] "Since that seemed to be a minor lemma in the ensuing paper I published (Chernoff, 1952), I neglected to give [Rubin] credit. I now consider it a serious error in judgment, especially because his result is stronger, for the upper bound, than the asymptotic result I had derived."

The particular value x/μ in these results may seem arbitrary, but in fact it's chosen vary carefully. The same arguments imply that

$$\Pr[X \geq x] \leq \frac{e^{(\alpha-1)\mu}}{\alpha^x} \text{ for } \textit{all } \alpha > 1 \qquad \text{and} \qquad \Pr[X \leq x] \leq \frac{e^{(\alpha-1)\mu}}{\alpha^x} \text{ for } \textit{all } \alpha < 1.$$

A bit of calculus implies that the right sides of these inequalities are minimized when $\alpha = x/\mu$.

Direct substitution now implies the following more traditional forms of Chernoff bounds, for any positive reals Δ and δ. Unlike the polynomial moment bounds we derived earlier, Chernoff bounds for the upper and lower tails of X are asymmetric.

$$\boxed{\begin{array}{ll} \Pr[X \geq \mu + \Delta] \leq e^{\Delta}\left(\dfrac{\mu}{\mu+\Delta}\right)^{\mu+\Delta} & \Pr[X \geq (1+\delta)\mu] \leq \left(\dfrac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu} \\ \Pr[X \leq \mu - \Delta] \leq e^{-\Delta}\left(\dfrac{\mu}{\mu-\Delta}\right)^{\mu-\Delta} & \Pr[X \leq (1-\delta)\mu] \leq \left(\dfrac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{\mu} \end{array}}$$

It is sometimes more convenient to work with the following looser forms of the multiplicative Chernoff bounds, which hold for any $0 < \delta < 1$. Here we use the notation $\exp(t) := e^t$.

$$\boxed{\begin{array}{l} \Pr[X \geq (1+\delta)\mu] \leq \exp\left(-\delta^2\mu/3\right) \\ \Pr[X \leq (1-\delta)\mu] \leq \exp\left(-\delta^2\mu/2\right) \end{array}}$$

These looser bounds can be derived from the more precise forms above using the Taylor-series approximations $1/\ln(1+\delta) \leq 1/2 + 1/\delta$ and $\ln(1-\delta) \geq -\delta + \delta^2/2$. I'll omit the straightforward but grungy details. (You're welcome.)

4.6 Coin Flips

To give an elementary example of applying these tail inequalities, consider an experiment where we independently flip N perfectly fair coins, and we want to bound the probability of getting at least αN heads, for some constant $1/2 < \alpha < 1$.

For each index i, let $X_i = 1$ if the ith flip comes up heads and $X_i = 0$ otherwise, so that $X = \sum_{i=1}^{N} X_i$ is the number of heads. Because the coins are fair, $\Pr[X_i = 1] = 1/2$ for all i. Linearity of expectation immediately implies $\mu = \mathrm{E}[X] = N/2$.

- Markov's inequality implies the nearly trivial bound $\Pr[X \geq \alpha N] \leq 1/2\alpha$. In particular, we have $\Pr[X \geq 3N/4] \leq 2/3$ and $\Pr[X \geq 9N/10] \leq 5/9$.
- Chebyshev's inequality implies $\Pr[X \geq \alpha N] \leq 1/((2\alpha-1)N) = O(1/(\alpha N))$. In particular, we have $\Pr[X \geq 3N/4] \leq 2/N$ and $\Pr[X \geq 9N/10] \leq 5/(4N)$.
- Higher moment inequalities implies smaller inverse-polynomial upper bounds. For example, the fourth-moment inequality implies $\Pr[X \geq \alpha N] \leq O(1/(\alpha N)^2)$.
- The simple version of Chernoff's inequality implies the inverse-exponential upper bound

$$\Pr[X \geq \alpha N] \leq \exp\left(-\frac{(2\alpha-1)^2}{6} \cdot N\right).$$

  In particular, we have $\Pr[X \geq 3N/4] < e^{-N/24}$ and $\Pr[X \geq 9N/10] < e^{-8N/75} < e^{-N/10}$.

- Symmetrically, Chernoff's inequality for the *lower* tail symmetrically bounds the probability of getting *few* heads as follows:

$$\Pr[X \le \alpha N] \le \exp\left(-\frac{(1-2\alpha)^2}{4}N\right).$$

  In particular, we have $\Pr[X \le N/4] < e^{-N/16}$ and $\Pr[X \le N/10] < e^{-4N/25} \le e^{-N/7}$.

- A more specialized argument involving binomial coefficients, which relies on all coins being fair, implies

$$\Pr[X \le \alpha N] \le \left(2\alpha^\alpha(1-\alpha)^{1-\alpha}\right)^{-N} \quad \text{and} \quad \Pr[X \ge \alpha N] \le \left(2\alpha^\alpha(1-\alpha)^{1-\alpha}\right)^{-N}.$$

  In particular, we have

$$\Pr[X \le N/4] \;\le\; \left(2\left(\frac{1}{4}\right)^{1/4}\left(\frac{3}{4}\right)^{3/4}\right)^{-N} \;\approx\; e^{-0.13081N} \;<\; e^{-N/8}$$

  and

$$\Pr[X \le N/10] \;\le\; \left(2\left(\frac{1}{10}\right)^{1/10}\left(\frac{9}{10}\right)^{9/10}\right)^{-N} \;\approx\; e^{-0.36806N} \;<\; e^{-N/3}.$$

4.7 Back to Treaps, Take 1

Consider an arbitrary node k in an n-node treap. We can argue informally that $depth(k) = O(\log n)$ with high probability by following the "good/bad pivot" analysis of randomized quicksort as follows.

For each integer $0 \le i \le depth(k)$, let v_i denote the ancestor of node k at depth i in the treap, and let n_i denote the number of nodes in the subtree rooted at v_i. (Thinking in terms of randomized quicksort, n_i is the size of the subproblem containing k after i levels of recursion.) For any integer $i > 0$, we label level i ***good*** or ***bad*** as follows:

- If $i < depth(k)$, then level i is good if $n_{i-1}/4 \le n_i \le 3n_{i-1}/4$ and bad otherwise.
- If $i \ge depth(k)$, then level i is good or bad according to an independent fair coin flip.

Each level $i \ge 1$ is good with probability at least $1/2$, and the events $[i \text{ is good}]$ are mutually independent. There are at most $\log_{4/3} n$ good levels smaller than $depth(k)$, since otherwise $n_{depth(k)}$ would be less than 1.

Thus, the depth of node k is *at most* the number of fair coin flips required to get $\log_{4/3} n$ heads. Equivalently, for any constant c, the probability that $depth(k) \ge c\log_{4/3} n$ is at most the probability that we see at most $\log_{4/3} n$ heads in a sequence of $c\log_{4/3} n$ independent fair coin flips. Our earlier analysis of coin flips via Chernoff bounds, with $N = c\log_{4/3} n$ and $\alpha = 1/c$, implies

$$\Pr\left[depth(k) \ge c\log_{4/3} n\right] \le \exp\left(-\frac{(1-2/c)^2}{4}c\log_{4/3} n\right) = n^{-(c-2)^2/(4c\ln(4/3))}.$$

For any $c > 2$, we obtain an inverse-polynomial upper bound, which we can make arbitrarily small by increasing the constant c. For example, setting $c = 20\ln(4/3) \approx 5.75364$ gives us $\Pr[depth(k) > 20\ln n] < 1/n^2$. We conclude that $depth(k) = O(\log n)$ with high probability.

Our more specialized argument implies that

$$\Pr\left[depth(k) \ge 10\log_{4/3} n\right] < e^{-10\log_{4/3} n/3} = n^{-10/(3\ln(4/3))} < \frac{1}{n^{11}}.$$

4.8 Back to Treaps: Take 2

In our analysis of randomized treaps, we wrote $\boldsymbol{i \uparrow k}$ to indicate that the node with the ith smallest key ('node i') was a proper ancestor of the node with the kth smallest key ('node k'). We argued that

$$\Pr[i \uparrow k] = \frac{[i \neq k]}{|k-i|+1},$$

and from this we concluded that the expected depth of node k is

$$\mathrm{E}[\mathit{depth}(k)] = \sum_{i=1}^{n} \Pr[i \uparrow k] = H_k + H_{n-k} - 2 < 2 \ln n.$$

To prove a worst-case expected bound on the depth of the tree, we need to argue that the *maximum* depth of any node is small. Chernoff bounds make this argument easy, once we establish that the relevant indicator variables are mutually independent.

Lemma 1. *For any index k, the $k-1$ random variables $[i \uparrow k]$ with $i < k$ are mutually independent, and the $n-k$ random variables $[i \uparrow k]$ with $i > k$ are mutually independent.*

Proof: Fix an arbitrary index k. We explicitly consider only the indicators $[i \uparrow k]$ with $i > k$; the proof for $i < k$ is nearly identical.

To simplify notation, let X_i denote the indicator variable $[i \uparrow k]$. We will prove by induction on n that the $n-k$ indicator variables $X_{k+1}, X_{k+2}, \ldots, X_n$ are mutually independent, starting with the vacuous base case $n = k$. Fix $n-k$ arbitrary indicator values $x_{k+1}, x_{k+2}, \ldots, x_n$. The definition of conditional probability immediately implies that

$$\begin{aligned}
\Pr\left[\bigwedge_{i=k+1}^{n} (X_i = x_i)\right] &= \Pr\left[\bigwedge_{i=k+1}^{n-1} (X_i = x_i) \wedge X_n = x_n\right] \\
&= \Pr\left[\bigwedge_{i=k+1}^{n-1} (X_i = x_i) \,\middle|\, X_n = x_n\right] \cdot \Pr\big[X_n = x_n\big]
\end{aligned}$$

Now recall that $X_n = [n \uparrow k] = 1$ if and only if node n has the smallest priority of all nodes between k and n. The other $n-k-1$ indicator variables X_i depend only on the order of the priorities of nodes k through $n-1$. There are exactly $(n-k-1)!$ permutations of the $n-k$ priorities in which the nth priority is smallest, and each of these permutations is equally likely. Thus,

$$\Pr\left[\bigwedge_{i=k+1}^{n-1} (X_i = x_i) \,\middle|\, X_n = x_n\right] = \Pr\left[\bigwedge_{i=k+1}^{n-1} (X_i = x_i)\right]$$

The inductive hypothesis implies that the variables $X_{k+1}, \ldots, X_{n-1}$ are mutually independent, so

$$\Pr\left[\bigwedge_{i=k+1}^{n-1} (X_i = x_i)\right] = \prod_{i=k+1}^{n-1} \Pr[X_i = x_i].$$

We conclude that

$$\Pr\left[\bigwedge_{i=k+1}^{n} (X_i = x_i)\right] = \Pr\big[X_n = x_n\big] \cdot \prod_{i=k+1}^{n-1} \Pr[X_k = x_k] = \prod_{i=k+1}^{n} \Pr[X_i = x_i],$$

or in other words, that the variables $X_i = [i \uparrow k]$ are indeed mutually independent. □

For purposes of analysis, define a new sequence $Z_2, Z_3, \ldots, Z_n$ of independent indicator variables, where $\Pr[Z_j = 1] = 1/j$ for each j, and let $Z := \sum_{j=2}^{n} Z_j$.

Lemma 2. $\Pr[Z > 4\ln n] < 1/n^2$.

Proof: Let $\mu = \mathrm{E}[Z] = H_n - 1 < \ln n$ and $\delta = c - 1$. Since Z is a sum of independent indicator variables, Chernoff's inequality implies

$$\begin{aligned}
\Pr[Z > 4\ln n] &< \Pr[Z > 4\mu] \\
&= \Pr[Z \geq (1+\delta)\mu] \\
&\leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu \\
&= \left(\frac{e^3}{4^4}\right)^\mu < \left(\frac{e^3}{4^4}\right)^{\ln n} = n^{3-4\ln 4} \approx n^{-2.54518} < \frac{1}{n^2}.
\end{aligned}$$

□

Theorem 3. *An n-node treap has depth $O(\log n)$ with high probability.*

Proof: Specifically, I will bound the probability that the depth is at most $8\ln n$. The union bound $\Pr[A \vee B] \leq \Pr[A] + \Pr[B]$ implies

$$\Pr\left[\max_k depth(k) > 8\ln n\right] = \Pr\left[\bigvee_{k=1}^{n} (depth(k) > 8\ln n)\right] \leq \sum_{k=1}^{n} \Pr[depth(k) > 8\ln n].$$

The depth of any node k is a sum of n indicator variables $[i \uparrow k]$, as i ranges from 1 to n. Lemma **??** partitions these variables into two mutually independent subsets. Define

$$depth_<(k) := \sum_{i=1}^{k-1} [i \uparrow k] \qquad \text{and} \qquad depth_>(k) := \sum_{i=k+1}^{n} [i \uparrow k],$$

so that $depth(k) = depth_<(k) + depth_>(k)$. The union bound implies

$$\Pr[depth(k) > 8\ln n] \leq \Pr[depth_<(k) > 4\ln n] + \Pr[depth_>(k) > 4\ln n].$$

For all indices $i \neq k$, the indicator variable $[i \uparrow k]$ has the same distribution as the variable $Z_{|i-k|+1}$ defined at the top of this page. Thus, for each k we have

$$\Pr[depth_<(k) > 4\ln n] = \Pr\left[\sum_{i=1}^{\ell-1} Z_\ell > 4\ln n\right] \leq \Pr[Z > 4\ln n] < 1/n^2.$$

Symmetrically, $\Pr[depth_<(k) > 4\ln n] \leq \Pr[Z > 4\ln n] < 1/n^2$. We immediately conclude that

$$\Pr\left[\max_k depth(k) > 8\ln n\right] \leq 2n\Pr[Z > 4\ln n] < 2/n,$$

which completes the proof. □

By varying the constants in the previous argument, we can prove that for any constant c, the depth of the treap is greater than $c\ln n$ with probability at most $2/n^{c\ln c - c}$.

We can now finally conclude that any search, insertion, deletion, or merge operation on an n-node treap requires $O(\log n)$ time with high probability. In particular, the expected *worst-case* time for each of these operations is $O(\log n)$.

Exercises

1. (a) Prove that for any index k such that $1 < k < n$, the $n-1$ indicator variables $[i \uparrow k]$ with $i \neq k$ are *not* mutually independent. *[Hint: Consider the case $n = 3$ and $k = 2$.]*

   (b) Prove that for any index k, the $k-1$ random variables $[k \uparrow i]$ with $i < k$ are *not* mutually independent. *[Hint: Consider the case $k = 4$.]*

2. (a) Consider an arbitrary node k in an n-node treap. Prove that the expected number of descendants of node k is *precisely* equal to the expected number of ancestors of node k.

   (b) Why doesn't the Chernoff-bound argument for depth imply that, with high probability, *every* node in a treap has $O(\log n)$ descendants? The conclusion is clearly bogus—Every treap has a node with n descendants!—but what's the hole in the argument?

3. The following algorithm finds the smallest element in an unsorted array:

```
RandomMin(A[1..n]):
  min ← ∞
  for i ← 1 to n in random order
      if A[i] < min
          min ← min1          (⋆)
      return min
```

   (a) Let X_i be the indicator variable that equals 1 if line $(\star)$ is executed in the ith iteration of the main loop. Prove that the variables X_i are mutually independent.

   (b) Prove that line $(\star)$ is executed $O(\log n)$ time with high probability.

4. Let S be a set of n points in the plane. A point p in S is called *Pareto-optimal* if no other point in S is both above and to the right of p. Suppose each point in S is chosen independently and uniformly at random from the unit square $[0,1] \times [0,1]$. Prove that the number of Pareto-optimal points in S is $O(\log n)$ with high probability.

5. Recall from the previous lecture note that a ***heater*** is a sort of anti-treap, in which the priorities of the nodes are given, but their search keys are generated independently and uniformly from the unit interval $[0,1]$. Prove that an n-node heater has depth $O(\log n)$ with high probability.

6. Let $X_1, X_2, \ldots, X_n$ be independent indicator variables, each equal to 1 with probability $1/2$, and let $X = \sum_i X_i$. Clearly $\mathrm{E}[X] = n/2$.

   (a) Let $Y = (X - \mathrm{E}[X])^2$. What is $\mathrm{E}[Y]$?

   (b) Let $Z = (Y - \mathrm{E}[Y])^2$. What is $\mathrm{E}[Z]$?

   (c) Let $\Omega = (Z - \mathrm{E}[Z])^2$. What is $\mathrm{E}[\Omega]$?

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

Insanity is repeating the same mistakes and expecting different results.

— *Narcotics Anonymous* (1981)

Calvin: *There! I finished our secret code!*
Hobbes: *Let's see.*
Calvin: *I assigned each letter a totally random number, so the code will be hard to crack. For letter "A", you write 3,004,572,688. "B" is 28,731,569½.*
Hobbes: *That's a good code all right.*
Calvin: *Now we just commit this to memory.*
Calvin: *Did you finish your map of our neighborhood?*
Hoobes: *Not yet. How many bricks does the front walk have?*

— Bill Watterson, "Calvin and Hobbes" (August 23, 1990)

[RFC 1149.5 specifies 4 as the standard IEEE-vetted random number.]

— Randall Munroe, *xkcd* (http://xkcd.com/221/)
Reproduced under a Creative Commons Attribution-NonCommercial 2.5 License

5 Hash Tables

5.1 Introduction

A ***hash table*** is a data structure for storing a set of items, so that we can quickly determine whether an item is or is not in the set. The basic idea is to pick a ***hash function*** h that maps every possible item x to a small integer $h(x)$. Then we store x in an array at index $h(x)$; the array itself is the hash table.

Let's be a little more specific. We want to store a set of n items. Each item is an element of a fixed set $\mathcal{U}$ called the ***universe***; we use u to denote the size of the universe, which is just the number of items in $\mathcal{U}$. A hash table is an array $T[1..m]$, where m is another positive integer, which we call the *table size*. Typically, m is much smaller than u. A *hash function* is any function of the form

$$h : \mathcal{U} \to \{0, 1, \ldots, m-1\},$$

mapping each possible item in $\mathcal{U}$ to a slot in the hash table. We say that an item x ***hashes*** to the slot $T[h(x)]$.

Of course, if $u = m$, we can always just use the trivial hash function $h(x) = x$; in other words, we can use the item itself as the index into the table. The resulting data structure is called a *direct access table*, or more commonly, an *array*. In most applications, however, this approach requires much more space than we can reasonably allocate. On the other hand, we rarely need need to store more than a tiny fraction of $\mathcal{U}$. Ideally, the table size m should be roughly equal to the number n of items we actually need to store, not the number of items that we might possibly store.

The downside of using a smaller table is that we must deal with *collisions*. We say that two items x and y ***collide*** if their hash values are equal: $h(x) = h(y)$. We are now left with two

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/ for the most recent revision.

different (but interacting) design decisions. First, how to we choose a hash function h that can be evaluated quickly and that results in as few collisions as possible? Second, when collisions do occur, how do we resolve them?

5.2 The Importance of Being Random

If we already knew the precise data set that would be stored in our hash table, it is possible (but not particularly easy) to find a *perfect* hash function that avoids collisions entirely. Unfortunately, for most applications of hashing, we don't know in advance what the user will put into the table. Thus, it is impossible, *even in principle*, to devise a perfect hash function in advance; no matter what hash function we choose, some pair of items from $\mathcal{U}$ *must* collide. In fact, for any fixed hash function, there is a subset of at least $|\mathcal{U}|/m$ items that all hash to the same location. If our input data happens to come from such a subset, either by chance or malicious intent, our code will come to a grinding halt. This is a real security issue with core Internet routers, for example; every router on the Internet backbone survives millions of attacks per day, including timing attacks, from malicious agents.

The *only* way to provably avoid this worst-case behavior is to choose our hash functions *randomly*. Specifically, we will fix a set $\mathcal{MB}^+$ of functions from $\mathcal{U}$ to $\{0, 1, \ldots, m-1\}$, and then at run time, we choose our hash function randomly from the set $\mathcal{MB}^+$ according to some fixed distribution. Different sets $\mathcal{MB}^+$ and different distributions over that set imply different theoretical guarantees. Screw this into your brain:

Input data is *not* random!
So good hash functions *must be* random!

In particular, the simple deterministic hash function $h(x) = x \bmod m$, which is often taught and recommended under the name "the division method", is ***utterly stupid***. Many textbooks correctly observe that this hash function is bad when m is a power of 2, because then $h(x)$ is just the low-order bits of m, but then they bizarrely recommend making m prime to avoid such obvious collisions. But even when m is prime, any pair of items whose difference is an integer multiple of m collide with absolute certainty; for all integers a and x, we have $h(x + am) = h(x)$. Why would anyone use a hash function where they *know* certain pairs of keys *always* collide? That's just crazy!

5.3 ...But Not Too Random

Most theoretical analysis of hashing assumes ***ideal random*** hash functions. Ideal randomness means that the hash function is chosen ***uniformly*** at random from the set of ***all*** functions from $\mathcal{U}$ to $\{0, 1, \ldots, m-1\}$. Intuitively, for each new item x, we roll a new m-sided die to determine the hash value $h(x)$. Ideal randomness is a clean theoretical model, which provides the strongest possible theoretical guarantees.

Unfortunately, ideal random hash functions are a theoretical fantasy; evaluating such a function would require recording values in a separate data structure which we could access using the items in our set, which is exactly what hash tables are for! So instead, we look for families of hash functions with *just enough* randomness to guarantee good performance. Fortunately, most

hashing analysis does not actually *require* ideal random hash functions, but only some weaker consequences of ideal randomness.

One property of ideal random hash functions that seems intuitively useful is ***uniformity***. A family $\mathcal{H}$ of hash functions is uniform if choosing a hash function uniformly at random from $\mathcal{H}$ makes every hash value equally likely for every item in the universe:

$$\textbf{Uniform:} \quad \Pr_{h \in \mathcal{H}}\big[h(x) = i\big] = \frac{1}{m} \quad \text{for all } x \text{ and all } i$$

We emphasize that this condition must hold for *every* item $x \in \mathcal{U}$ and *every* index i. Only the hash function h is random.

In fact, despite its intuitive appeal, uniformity is not terribly important or useful by itself. Consider the family $\mathcal{K}$ of *constant* hash functions defined as follows. For each integer a between 0 and $m-1$, let $const_a$ denote the constant function $const_a(x) = a$ for all x, and let $\mathcal{K} = \{const_a \mid 0 \le a \le m-1\}$ be the set of all such functions. It is easy to see that the set $\mathcal{K}$ is both perfectly uniform and utterly useless!

A much more important goal is to minimize the number of collisions. A family of hash functions is ***universal*** if, for any two items in the universe, the probability of collision is as small as possible:

$$\textbf{Universal:} \quad \Pr_{h \in \mathcal{H}}\big[h(x) = h(y)\big] \le \frac{1}{m} \quad \text{for all } x \neq y$$

(Trivially, if $x = y$, then $\Pr[h(x) = h(y)] = 1$!) Again, we emphasize that this equation must hold for *every* pair of distinct items; only the function h is random. The family of constant functions is uniform but not universal; on the other hand, universal hash families are not necessarily uniform.[1]

Most elementary hashing analysis requires a weaker versions of universality. A family of hash functions is ***near-universal*** if the probability of collision is *close* to ideal:

$$\textbf{Near-universal:} \quad \Pr_{h \in \mathcal{H}}\big[h(x) = h(y)\big] \le \frac{2}{m} \quad \text{for all } x \neq y$$

There's nothing special about the number 2 in this definition; any other explicit constant will do.

On the other hand, some hashing analysis requires reasoning about larger sets of collisions. For any integer k, we say that a family of hash functions is ***strongly k-universal*** or ***k-uniform*** if for any sequence of k disjoint keys and any sequence of k hash values, the probability that each key maps to the corresponding hash value is $1/m^k$:

$$\textbf{k-uniform:} \quad \Pr_{h \in \mathcal{H}}\left[\bigwedge_{j=1}^{k} h(x_j) = i_j\right] = \frac{1}{m^k} \quad \text{for all distinct } x_1, \ldots, x_k \text{ and all } i_1, \ldots, i_k$$

Ideal random hash functions are k-uniform for every positive integer k.

[1] Confusingly, universality is often called the ***uniform hashing assumption***, even though it is not an assumption that the hash function is uniform.

5.4 Chaining

One of the most common methods for resolving collisions in hash tables is called *chaining*. In a *chained* hash table, each entry $T[i]$ is not just a single item, but rather (a pointer to) a linked list of all the items that hash to $T[i]$. Let $\ell(x)$ denote the length of the list $T[h(x)]$. To see if an item x is in the hash table, we scan the entire list $T[h(x)]$. The worst-case time required to search for x is $O(1)$ to compute $h(x)$ plus $O(1)$ for every element in $T[h(x)]$, or $O(1+\ell(x))$ overall. Inserting and deleting x also take $O(1+\ell(x))$ time.

A chained hash table with load factor 1.

Let's compute the expected value of $\ell(x)$ under this assumption; this will immediately imply a bound on the expected time to search for an item x. To be concrete, let's suppose that x is not already stored in the hash table. For all items x and y, we define the indicator variable

$$C_{x,y} = \big[h(x) = h(y)\big].$$

(In case you've forgotten the bracket notation, $C_{x,y} = 1$ if $h(x) = h(y)$ and $C_{x,y} = 0$ if $h(x) \neq h(y)$.) Since the length of $T[h(x)]$ is precisely equal to the number of items that collide with x, we have

$$\ell(x) = \sum_{y \in T} C_{x,y}.$$

Assuming h is chosen from a ***universal*** set of hash functions, we have

$$\mathrm{E}[C_{x,y}] = \Pr[C_{x,y} = 1] = \begin{cases} 1 & \text{if } x = y \\ 1/m & \text{otherwise} \end{cases}$$

Now we just have to grind through the definitions.

$$\mathrm{E}[\ell(x)] = \sum_{y \in T} \mathrm{E}[C_{x,y}] = \sum_{y \in T} \frac{1}{m} = \frac{n}{m}$$

We call this fraction n/m the *load factor* of the hash table. Since the load factor shows up everywhere, we will give it its own symbol α.

$$\boxed{\alpha := \frac{n}{m}}$$

Similarly, if h is chosen from a ***near***-universal set of hash functions, then $\mathrm{E}[\ell(x)] \leq 2\alpha$. Thus, the expected time for an unsuccessful search in a chained hash table, using near-universal hashing, is $\Theta(1+\alpha)$. As long as the number of items n is only a constant factor bigger than the table size m, the search time is a constant. A similar analysis gives the same expected time bound (with a slightly smaller constant) for a successful search.

Obviously, linked lists are not the only data structure we could use to store the chains; any data structure that can store a set of items will work. For example, if the universe $\mathcal{U}$ has a total

ordering, we can store each chain in a balanced binary search tree. This reduces the expected time for any search to $O(1 + \log \ell(x))$, and assuming near-universal hashing, the expected time for any search is $O(1 + \log \alpha)$.

Another natural possibility is to work recursively! Specifically, for each $T[i]$, we maintain a hash table T_i containing all the items with hash value i. Collisions in those secondary tables are resolved recursively, by storing secondary overflow lists in tertiary hash tables, and so on. The resulting data structure is a tree of hash tables, whose leaves correspond to items that (at some level of the tree) are hashed without any collisions. If every hash table in this tree has size m, then the expected time for any search is $O(\log_m n)$. In particular, if we set $m = \sqrt{n}$, the expected time for any search is *constant*. On the other hand, there is no inherent reason to use the same hash table size everywhere; after all, hash tables deeper in the tree are storing fewer items.

Caveat Lector! The preceding analysis does *not* imply that the expected *worst-case* search time is constant! The expected worst-case search time is $O(1 + L)$, where $L = \max_x \ell(x)$. Even with *ideal* random hash functions, the maximum list size L is *very* likely to grow faster than any constant, unless the load factor α is *significantly* smaller than 1. For example, $\mathrm{E}[L] = \Theta(\log n / \log\log n)$ when $\alpha = 1$. We've stumbled on a powerful but counterintuitive fact about probability: When several individual items are distributed independently and uniformly at random, the overall distribution of those items is *almost never* uniform in the traditional sense! Later in this lecture, I'll describe how to achieve constant expected worst-case search time using secondary hash tables.

5.5 Multiplicative Hashing

Arguably the simplest technique for near-universal hashing, first described by Lawrence Carter and Mark Wegman in the late 1970s, is called ***multiplicative hashing***. I'll describe two variants of multiplicative hashing, one using modular arithmetic with prime numbers, the other using modular arithmetic with powers of two. In both variants, a hash function is specified by an integer parameter a, called a *salt*. The salt is chosen uniformly at random when the hash table is created and remains fixed for the entire lifetime of the table. All probabilities are defined with respect to the random choice of salt.

For any non-negative integer n, let $[\![n]\!]$ denote the n-element set $\{0, 1, \ldots, n-1\}$, and let $[\![n]\!]^+$ denote the $(n-1)$-element set $\{1, 2, \ldots, n-1\}$.

5.5.1 Prime multiplicative hashing

The first family of multiplicative hash function is defined in terms of a prime number $p > |\mathcal{U}|$. For any integer $a \in [\![p]\!]^+$, define a function $\mathit{multp}_a : \mathcal{U} \to [\![m]\!]$ by setting

$$\boxed{\mathit{multp}_a(x) = (ax \bmod p) \bmod m}$$

and let

$$\mathcal{MP} := \left\{ \mathit{multp}_a \;\middle|\; a \in [\![p]\!]^+ \right\}$$

denote the set of all such functions. Here, the integer a is the salt for the hash function multp_a. We claim that this family of hash functions is near-universal.

The use of prime modular arithmetic is motivated by the fact that *division* modulo prime numbers is well-defined.

Lemma 1. *For every integer $a \in [\![p]\!]^+$, there is a unique integer $z \in [\![p]\!]^+$ such that $az \bmod p = 1$.*

Proof: Fix an arbitrary integer $a \in [p]^+$.

Suppose $az \bmod p = az' \bmod p$ for some integers $z, z' \in [p]^+$. We immediately have $a(z-z') \bmod p = 0$, which implies that $a(z-z')$ is divisible by p. Because p is prime, the inequality $1 \le a \le p-1$ implies that $z - z'$ must be divisible by p. Similarly, because $1 \le z, z' \le p-1$, we have $2-p \le z-z' \le p-2$, which implies that $z = z'$. It follows that for each integer $h \in [p]^+$, there is *at most* one integer $z \in [p]^+$ such that $az \bmod p = h$.

Similarly, if $az \bmod p = 0$ for some integer $z \in [p]^+$, then because p is prime, either a or z is divisible by p, which is impossible.

We conclude that the set $\{az \bmod p \mid z \in [p]^+\}$ has exactly $p-1$ distinct elements, all non-zero, and therefore is equal to $[p]^+$. In other words, multiplication by a defines a permutation of $[p]^+$. The lemma follows immediately. □

Let $\boldsymbol{a^{-1}}$ denote the multiplicative inverse of a, as guaranteed by the previous lemma. We can now precisely characterize when the hash values of two items collide.

Lemma 2. *For any elements $a, x, y \in [p]^+$, we have a collision $multp_a(x) = multp_a(y)$ if and only if either $x = y$ or $multp_a((x-y) \bmod p) = 0$ or $multp_a((y-x) \bmod p) = 0$.*

Proof: Fix three arbitrary elements $a, x, y \in [p]^+$. There are three cases to consider, depending on whether $ax \bmod p$ is greater than, less than, or equal to $ay \bmod p$.

First, suppose $ax \bmod p = ay \bmod p$. Then $x = a^{-1}ax \bmod p = a^{-1}ay \bmod p = y$, which implies that $x = y$. (This is the only place we need primality.)

Next, suppose $ax \bmod p > ay \bmod p$. We immediately observe that

$$ax \bmod p - ay \bmod p \;=\; (ax - ay) \bmod p \;=\; a(x-y) \bmod p.$$

Straightforward algebraic manipulation now implies that $multp_a(x) = multp_a(y)$ if and only if $multp_a((x-y) \bmod p) = 0$.

$$\begin{aligned} multp_a(x) = multp_a(y) &\iff (ax \bmod p) \bmod m = (ay \bmod p) \bmod m \\ &\iff (ax \bmod p) - (ay \bmod p) \equiv 0 \pmod{m} \\ &\iff a(x-y) \bmod p \equiv 0 \pmod{m} \\ &\iff multp_a((x-y) \bmod p) = 0 \end{aligned}$$

Finally, if $ax \bmod p < ay \bmod p$, an argument similar to the previous case implies that $multp_a(x) = multp_a(y)$ if and only if $multp_a((y-x) \bmod p) = 0$. □

For any distinct integers $x, y \in \mathcal{U}$, Lemma **??** immediately implies that

$$\begin{aligned} &\Pr_a\big[multp_a(x) = multp_a(y)\big] \\ &\qquad \le \Pr_a\big[multp_a((x-y) \bmod p) = 0\big] + \Pr_a\big[multp_a((y-x) \bmod p) = 0\big]. \end{aligned}$$

Thus, to show that $\mathcal{MP}$ is near-universal, it suffices to prove the following lemma.

Lemma 3. *For any integer $z \in [p]^+$, we have $\Pr_a[multp_a(z) = 0] \le 1/m$.*

Proof: Fix an arbitrary integer $z \in [p]^+$. Lemma 1 implies that for any integer $h \in [p]^+$, there is a unique integer $a \in [p]^+$ such that $(az \bmod p) = h$; specifically, $a = h \cdot z^{-1} \bmod p$. There are exactly $\lfloor (p-1)/m \rfloor$ integers k such that $1 \le km \le p-1$. Thus, there are exactly $\lfloor (p-1)/m \rfloor$ salts a such that $multp_a(z) = 0$. □

Our analysis of collision probability can be improved, but only slightly. Carter and Wegman observed that if $p \bmod (m+1) = 1$, then $\Pr_a[multp_a(1) = multp_a(m+1)] = 2/(m+1)$. (For any positive integer m, there are infinitely many primes p such that $p \bmod (m+1) = 1$.) For example, by enumerating all possible values of $multp_a(x)$ when $p = 5$ and $m = 3$, we immediately observe that $\Pr_a[multp_a(1) = multp_a(4)] = 1/2 = 2/(m+1) > 1/3$.

	1	2	3	4
0	0	0	0	0
1	**1**	**2**	**0**	**1**
2	2	1	1	0
3	0	1	1	2
4	**1**	**0**	**2**	**1**

5.5.2 Actually universal hashing

Our first example of a truly universal family of hash functions uses a small modification of the multiplicative method we just considered. For any integers $a \in [p]^+$ and $b \in [p]$, let $h_{a,b} : \mathcal{U} \to [m]$ be the function

$$\boxed{h_{a,b}(x) = ((ax+b) \bmod p) \bmod m}$$

and let

$$\mathcal{MB}^+ := \left\{ h_{a,b} \;\middle|\; a \in [p]^+, b \in [p] \right\}$$

denote the set of all $p(p-1)$ such functions. A function in this family is specified by *two* salt parameters a and b.

Theorem 1. *$\mathcal{MB}^+$ is universal.*

Proof: Fix four integers $r, s, x, y \in [p]$ such that $x \neq y$ and $r \neq s$. The linear system

$$\begin{aligned} ax + b &\equiv r \pmod p \\ ay + b &\equiv s \pmod p \end{aligned}$$

has a unique solution $a, b \in [p]$ with $a \neq 0$, namely

$$\begin{aligned} a &= (r-s)(x-y)^{-1} \bmod p \\ b &= (sx - ry)(x-y)^{-1} \bmod p \end{aligned}$$

where z^{-1} denotes the mod-p multiplicative inverse of z, as guaranteed by Lemma 1. It follows that

$$\Pr_{a,b}\big[(ax+b) \bmod p = r \;\text{ and }\; (ay+b) \bmod p = s\big] = \frac{1}{p(p-1)},$$

and therefore

$$\Pr_{a,b}\big[h_{a,b}(x) = h_{a,b}(y)\big] = \frac{N}{p(p-1)},$$

where N is the number of ordered pairs $(r,s) \in [p]^2$ such that $r \neq s$ but $r \bmod m = s \bmod m$. For each fixed $r \in [p]$, there are at most $\lfloor p/m \rfloor$ integers $s \in [p]$ such that $r \neq s$ but $r \bmod m = s \bmod m$. Because p is prime, we have $\lfloor p/m \rfloor \leq (p-1)/m$. We conclude that $N \leq p(p-1)/m$, which completes the proof. □

More careful analysis implies that the collision probability for any pair of items is exactly

$$\frac{(p - p \bmod m)(p - (m - p \bmod m))}{mp(p-1)}.$$

Because p is prime, we must have $0 < p \bmod m < m$, so this probability is actually *strictly less than* $1/m$. For example, when $p = 5$ and $m = 3$, the collision probability is

$$\frac{(5 - 5 \bmod 3)(5 - (3 - 5 \bmod 3))}{3 \cdot 4 \cdot 5} = \frac{1}{5} < \frac{1}{3},$$

which we can confirm by enumerating all possible values:

$b = 0$

	1	2	3	4
0	0	0	0	0
1	**1**	2	0	**1**
2	2	1	1	0
3	0	1	1	2
4	**1**	0	2	**1**

$b = 1$

	1	2	3	4
1	1	1	1	1
1	2	0	1	0
2	0	0	2	1
3	1	2	0	0
4	0	1	0	2

$b = 2$

	1	2	3	4
0	2	2	2	2
1	0	1	0	1
2	1	1	0	0
3	0	0	1	1
4	1	0	1	0

$b = 3$

	1	2	3	4
0	0	0	0	0
1	1	0	1	2
2	0	2	1	1
3	1	1	2	0
4	2	1	0	1

$b = 4$

	1	2	3	4
0	1	1	1	1
1	**0**	1	2	**0**
2	1	0	0	2
3	2	0	0	1
4	**0**	2	1	**0**

5.5.3 Binary multiplicative hashing

A slightly simpler variant of multiplicative hashing that avoids the need for large prime numbers was first formally analyzed by Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen in 1997, although it was proposed decades earlier. For this variant, we assume that $\mathcal{U} = [2^w]$ and that $m = 2^\ell$ for some integers w and ℓ. Thus, our goal is to hash w-bit integers ("words") to ℓ-bit integers ("labels").

For any *odd* integer $a \in [2^w]$, we define the hash function $multb_a : \mathcal{U} \to [m]$ as follows:

$$\boxed{multb_a(x) := \left\lfloor \frac{(a \cdot x) \bmod 2^w}{2^{w-\ell}} \right\rfloor}$$

Again, the odd integer a is the salt.

Binary multiplicative hashing.

If we think of any w-bit integer z as an array of bits $z[0..w-1]$, where $z[0]$ is the least significant bit, this function has an easy interpretation. The product $a \cdot x$ is $2w$ bits long; the hash value $multb_a(x)$ consists of the top ℓ bits of the bottom half:

$$multb_a(x) := (a \cdot x)[w-1..w-\ell]$$

Most programming languages automatically perform integer arithmetic modulo some power of two. If we are using an integer type with w bits, the function $multb_a(x)$ can be implemented by a single multiplication followed by a single right-shift. For example, in C:

```
#define hash(a,x) ((a)*(x) >> (WORDSIZE-HASHBITS))
```

Now we claim that the family $\mathcal{MB} := \{multb_a \mid a \text{ is odd}\}$ of all such functions is near-universal. To prove this claim, we again need to argue that division is well-defined, at least for a large subset of possible words. Let W denote the set of odd integers in $[2^w]$.

Lemma 4. *For any integers $x, z \in W$, there is exactly one integer $a \in W$ such that $ax \bmod 2^w = z$.*

Proof: Fix an integer $x \in W$. Suppose $ax \bmod 2^w = bx \bmod 2^w$ for some integers $a, b \in W$. Then $(b-a)x \bmod 2^w = 0$, which means $x(b-a)$ is divisible by 2^w. Because x is odd, $b-a$ must be divisible by 2^w. But $-2^w < b-a < 2^w$, so a and b must be equal. Thus, for each $z \in W$, there is *at most* one $a \in W$ such that $ax \bmod 2^w = z$. In other words, the function $f_x : W \to W$ defined by $f_x(a) := ax \bmod 2^w$ is injective. Every injective function from a finite set to itself is a bijection. □

Theorem 2. *$\mathcal{MB}$ is near-universal.*

Proof: Fix two distinct words $x, y \in \mathcal{U}$ such that $x < y$. If $multb_a(x) = multb_a(y)$, then the top ℓ bits of $a(y-x) \bmod 2^w$ are either all 0s (if $ax \bmod 2^w \le ay \bmod 2^w$) or all 1s (otherwise). Equivalently, if $multb_a(x) = multb_a(y)$, then either $multb_a(y-x) = 0$ or $multb_a(y-x) = m-1$. Thus,

$$\Pr[multb_a(x) = multb_a(y)] \;\le\; \Pr[multb_a(y-x) = 0] \;+\; \Pr[multb_a(y-x) = m-1].$$

We separately bound the terms on the right side of this inequality.

Because $x \ne y$, we can write $(y-x) \bmod 2^w = q2^r$ for some odd integer q and some integer $0 \le r \le w-1$. The previous lemma implies that $aq \bmod 2^w$ consists of $w-1$ random bits followed by a 1. Thus, $aq2^r \bmod 2^w$ consists of $w-r-1$ random bits, followed by a 1, followed by r 0s. There are three cases to consider:

- If $r < w-\ell$, then $multb_a(y-x)$ consists of ℓ random bits, so

$$\Pr[multb_a(y-x) = 0] \;=\; \Pr[multb_a(y-x) = m-1] \;=\; 1/2^\ell.$$

- If $r = w-\ell$, then $multb_a(y-x)$ consists of $\ell-1$ random bits followed by a 1, so

$$\Pr[multb_a(y-x) = 0] = 0 \quad \text{and} \quad \Pr[multb_a(y-x) = m-1] = 2/2^\ell.$$

- Finally, if $r < w-\ell$, then $multb_a(y-x)$ consists of zero or more random bits, followed by a 1, followed by one or more 0s, so

$$\Pr[multb_a(y-x) = 0] \;=\; \Pr[multb_a(y-x) = m-1] \;=\; 0.$$

In all cases, we have $\Pr[multb_a(x) = multb_a(y)] \le 2/2^\ell$, as required. □

*5.6 High Probability Bounds: Balls and Bins

Although any particular search in a chained hash tables requires only constant expected time, but what about the *worst* search time? Assuming that we are using ***ideal random*** hash functions, this question is equivalent to the following more abstract problem. Suppose we toss n balls independently and uniformly at random into one of n bins. Can we say anything about the number of balls in the fullest bin?

Lemma 5. *If n balls are thrown independently and uniformly into n bins, then with high probability, the fullest bin contains $O(\log n/\log\log n)$ balls.*

Proof: Let X_j denote the number of balls in bin j, and let $\hat{X} = \max_j X_j$ be the maximum number of balls in any bin. Clearly, $\mathrm{E}[X_j] = 1$ for all j.

Now consider the probability that bin j contains at least k balls. There are $\binom{n}{k}$ choices for those k balls, and the probability of any particular subset of k balls landing in bin j is $1/n^k$, so the union bound ($\Pr[A \vee B] \le \Pr[A] + \Pr[B]$ for any events A and B) implies

$$\Pr[X_j \ge k] \;\le\; \binom{n}{k}\left(\frac{1}{n}\right)^k \;\le\; \frac{n^k}{k!}\left(\frac{1}{n}\right)^k \;=\; \frac{1}{k!}.$$

Setting $k = 2c\lg n/\lg\lg n$, we have

$$k! \ge k^{k/2} = \left(\frac{2c\lg n}{\lg\lg n}\right)^{2c\lg n/\lg\lg n} \ge \left(\sqrt{\lg n}\right)^{2c\lg n/\lg\lg n} = 2^{c\lg n} = n^c,$$

which implies that

$$\Pr\left[X_j \ge \frac{2c\lg n}{\lg\lg n}\right] < \frac{1}{n^c}.$$

This probability bound holds for every bin j. Thus, by the union bound, we conclude that

$$\Pr\left[\max_j X_j > \frac{2c\lg n}{\lg\lg n}\right] \;=\; \Pr\left[X_j > \frac{2c\lg n}{\lg\lg n} \text{ for all } j\right] \;\le\; \sum_{j=1}^{n}\Pr\left[X_j > \frac{2c\lg n}{\lg\lg n}\right] \;<\; \frac{1}{n^{c-1}}. \qquad \square$$

A somewhat more complicated argument implies that if we throw n balls randomly into n bins, then with high probability, the fullest bin contains at least $\Omega(\log n/\log\log n)$ balls.

However, if we make the hash table sufficiently large, we can expect every ball to land in its own bin. Suppose there are m bins. Let C_{ij} be the indicator variable that equals 1 if and only if $i \ne j$ and ball i and ball j land in the same bin, and let $C = \sum_{i<j} C_{ij}$ be the total number of pairwise collisions. Since the balls are thrown uniformly at random, the probability of a collision is exactly $1/m$, so $\mathrm{E}[C] = \binom{n}{2}/m$. In particular, if $m = n^2$, the expected number of collisions is less than $1/2$, and thus by Markov's inequality, the probability of getting *even one* collision is less than $1/2$.

We can give a slightly weaker version of this bound that assumes only near-universal hashing. Suppose we hash n items into a table of size m. Linearity of expectation implies that the expected number of collisions is

$$\sum_{x<y}\Pr[h(x) = h(y)] \le \binom{n}{2}\frac{2}{m} = \frac{n(n-1)}{m}.$$

In particular, if we set $m = 2n^2$, the expected number of collisions is less than $1/2$. Again, Markov's inequality implies that the probability of *even one* collision is less than $1/2$.

If we make the hash table slightly larger, we can even prove a high-probability bound.

Lemma 6. *For any $\varepsilon > 0$, if n balls are thrown independently and uniformly into $n^{2+\varepsilon}$ bins, then with high probability, no bin contains more than one ball.*

Proof: Let X_j denote the number of balls in bin j, as in the previous proof. We can easily bound the probability that bin j is empty, by taking the two most significant terms in a binomial expansion:

$$\Pr[X_j = 0] \;=\; \left(1 - \frac{1}{m}\right)^n \;=\; \sum_{i=1}^{n} \binom{n}{i} \left(\frac{-1}{m}\right)^i \;=\; 1 - \frac{n}{m} + \Theta\left(\frac{n^2}{m^2}\right) \;>\; 1 - \frac{n}{m}$$

We can similarly bound the probability that bin j contains exactly one ball:

$$\Pr[X_j = 1] \;=\; n \cdot \frac{1}{m}\left(1 - \frac{1}{m}\right)^{n-1} \;=\; \frac{n}{m}\left(1 - \frac{n-1}{m} + \Theta\left(\frac{n^2}{m^2}\right)\right) \;>\; \frac{n}{m} - \frac{n(n-1)}{m^2}$$

It follows immediately that $\Pr[X_j > 1] < n(n-1)/m^2$. The union bound now implies that $\Pr[\hat{X} > 1] < n(n-1)/m$. If we set $m = n^{2+\varepsilon}$ for any constant $\varepsilon > 0$, then the probability that no bin contains more than one ball is at least $1 - 1/n^{\varepsilon}$. □

5.7 Perfect Hashing

So far we are faced with two alternatives. If we use a small hash table to keep the space usage down, even if we use ideal random hash functions, the resulting worst-case expected search time is $\Theta(\log n / \log\log n)$ with high probability, which is not much better than a binary search tree. On the other hand, we can get constant worst-case search time, at least in expectation, by using a table of roughly quadratic size, but that seems unduly wasteful.

Fortunately, there is a fairly simple way to combine these two ideas to get a data structure of linear expected size, whose expected worst-case search time is constant. At the top level, we use a hash table of size $m = n$ and a near-universal hash function, but instead of linked lists, we use secondary hash tables to resolve collisions. Specifically, the jth secondary hash table has size $2n_j^2$, where n_j is the number of items whose primary hash value is j. Our earlier analysis implies that with probability at least $1/2$, the secondary hash table has no collisions at all, so the worst-case search time in any secondary hash table is $O(1)$. (If we discover a collision in some secondary hash table, we can simply rebuild that table with a new near-universal hash function.)

Although this data structure apparently needs significantly more memory for each secondary structure, the overall increase in space is insignificant, at least in expectation.

Lemma 7. *Assuming near-universal hashing, we have* $\mathrm{E}\left[\sum_i n_i^2\right] < 3n$.

Proof: let $h(x)$ denote the position of x in the primary hash table. We can rewrite the sum $\sum_i n_i^2$ in terms of the indicator variables $[h(x) = i]$ as follows. The first equation uses the definition of n_i; the rest is just routine algebra.

$$\begin{aligned}\sum_i n_i^2 &= \sum_i \left(\sum_x [h(x) = i]\right)^2 \\ &= \sum_i \left(\sum_x \sum_y [h(x) = i][h(y) = i]\right)\end{aligned}$$

$$
\begin{aligned}
&= \sum_i \left(\sum_x [h(x) = i]^2 \;+\; 2 \sum_{x<y} [h(x) = i][h(y) = i] \right) \\
&= \sum_x \sum_i [h(x) = i]^2 \;+\; 2 \sum_{x<y} \sum_i [h(x) = i][h(y) = i] \\
&= \sum_x \sum_i [h(x) = i] \;+\; 2 \sum_{x<y} [h(x) = h(y)]
\end{aligned}
$$

The first sum is equal to n, because each item x hashes to exactly one index i, and the second sum is just the number of pairwise collisions. Linearity of expectation immediately implies that

$$
\mathrm{E}\left[\sum_i n_i^2 \right] \;=\; n + 2 \sum_{x<y} \Pr[h(x) = h(y)] \;\le\; n + 2 \cdot \frac{n(n-1)}{2} \cdot \frac{2}{n} \;=\; 3n - 2. \qquad \square
$$

This lemma immediately implies that the expected size of our two-level hash table is $O(n)$. By our earlier analysis, the expected worst-case search time is $O(1)$.

5.8 Open Addressing

Another method used to resolve collisions in hash tables is called *open addressing*. Here, rather than building secondary data structures, we resolve collisions by looking elsewhere in the table. Specifically, we have a sequence of hash functions $\langle h_0, h_1, h_2, \ldots, h_{m-1} \rangle$, such that for any item x, the *probe sequence* $\langle h_0(x), h_1(x), \ldots, h_{m-1}(x) \rangle$ is a permutation of $\langle 0, 1, 2, \ldots, m-1 \rangle$. In other words, different hash functions in the sequence always map x to different locations in the hash table.

We search for x using the following algorithm, which returns the array index i if $T[i] = x$, 'absent' if x is not in the table but there is an empty slot, and 'full' if x is not in the table and there no no empty slots.

```
OpenAddressSearch(x):
  for i ← 0 to m − 1
      if T[h_i(x)] = x
          return h_i(x)
      else if T[h_i(x)] = ∅
          return 'absent'
  return 'full'
```

The algorithm for inserting a new item into the table is similar; only the second-to-last line is changed to $T[h_i(x)] \leftarrow x$. Notice that for an open-addressed hash table, the load factor is never bigger than 1.

Just as with chaining, we'd like to pretend that the sequence of hash values is truly random, for purposes of analysis. Specifically, most open-addressed hashing analysis uses the following assumption, which is impossible to enforce in practice, but leads to reasonably predictive results for most applications.

Strong uniform hashing assumption:

> For each item x, the probe sequence $\langle h_0(x), h_1(x), \ldots, h_{m-1}(x) \rangle$ is equally likely to be any permutation of the set $\{0, 1, 2, \ldots, m-1\}$.

Let's compute the expected time for an unsuccessful search in light of this assupmtion. Suppose there are currently n elements in the hash table. The strong uniform hashing assumption has two important consequences:

- **Uniformity:** For each item x and index i, the hash value $h_i(x)$ is equally likely to be any integer in the set $\{0, 1, 2, \ldots, m-1\}$.
- **Indpendence:** For each item x, if we ignore the first probe $h_0(x)$, the remaining probe sequence $\langle h_1(x), h_2(x), \ldots, h_{m-1}(x)\rangle$ is equally likely to be any permutation of the smaller set $\{0, 1, 2, \ldots, m-1\} \setminus \{h_0(x)\}$.

Uniformity implies that the probability that $T[h_0(x)]$ is occupied is exactly n/m. Independence implies that if $T[h_0(x)]$ is occupied, *our search algorithm recursively searches the rest of the hash table!* Since the algorithm will never again probe $T[h_0(x)]$, for purposes of analysis, we might as well pretend that slot in the table no longer exists. Thus, we get the following recurrence for the expected number of probes, as a function of m and n:

$$\mathrm{E}[T(m,n)] = 1 + \frac{n}{m}\,\mathrm{E}[T(m-1,n-1)].$$

The trivial base case is $T(m,0) = 1$; if there's nothing in the hash table, the first probe always hits an empty slot. We can now easily prove by induction that $\mathbf{E[T(m,n)] \le m/(m-n)}$:

$$\begin{aligned}
\mathrm{E}[T(m,n)] &= 1 + \frac{n}{m}\,\mathrm{E}[T(m-1,n-1)] \\
&\le 1 + \frac{n}{m}\cdot\frac{m-1}{m-n} && \text{[induction hypothesis]} \\
&< 1 + \frac{n}{m}\cdot\frac{m}{m-n} && [m-1 < m] \\
&= \frac{m}{m-n}\ \checkmark && \text{[algebra]}
\end{aligned}$$

Rewriting this in terms of the load factor $\alpha = n/m$, we get $\mathbf{E[T(m,n)] \le 1/(1-\alpha)}$. In other words, the expected time for an unsuccessful search is $O(1)$, unless the hash table is almost completely full.

5.9 Linear and Binary Probing

In practice, however, we can't generate ideal random probe sequences, so we must rely on a simpler probing scheme to resolve collisions. Perhaps the simplest scheme is ***linear probing***—use a single hash function $h(x)$ and define

$$\boxed{h_i(x) := (h(x) + i) \bmod m}$$

This strategy has several advantages, in addition to its obvious simplicity. First, because the probing strategy visits consecutive entries in the has table, linear probing exhibits better cache performance than other strategies. Second, as long as the load factor is strictly less than 1, the expected length of any probe sequence is provably constant; moreover, this performance is guaranteed even for hash functions with limited independence. On the other hand, the number or probes grows quickly as the load factor approaches 1, because the occupied cells in the hash table tend to cluster together. On the gripping hand, this clustering is arguably an *advantage* of linear probing, since any access to the hash table loads several nearby entries into the cache.

A simple variant of linear probing called ***binary probing*** is slightly easier to analyze. Assume that $m = 2^\ell$ for some integer ℓ (in a binary multiplicative hashing), and define

$$\boxed{h_i(x) := h(x) \oplus i}$$

where $\oplus$ denotes bitwise exclusive-or. This variant of linear probing has slightly better cache performance, because cache lines (and disk pages) usually cover address ranges of the form $[r2^k \mathinner{..} (r+1)2^k - 1]$; assuming the hash table is aligned in memory correctly, binary probing will scan one entire cache line before loading the next one.

Several more complex probing strategies have been proposed in the literature. Two of the most common are ***quadratic probing***, where we use a single hash function h and set $h_i(x) := (h(x) + i^2) \bmod m$, and ***double hashing***, where we use two hash functions h and h' and set $h_i(x) := (h(x) + i \cdot h'(x)) \bmod m$. These methods have some theoretical advantages over linear and binary probing, but they are not as efficient in practice, primarily due to cache effects.

*5.10 Analysis of Binary Probing

Lemma 8. *In a hash table of size $m = 2^\ell$ containing $n \le m/4$ keys, built using binary probing, the expected time for any search is $O(1)$,* ***assuming ideal random hashing****.*

Proof: The hash table is an array $H[0 \mathinner{..} m-1]$. For each integer k between 0 and ℓ, we partition H into $m/2^k$ ***level-k blocks*** of length 2^k; each level-k block has the form $H[c2^k \mathinner{..} (c+1)2^k - 1]$ for some integer c. Each level-k block contains exactly two level-$(k-1)$ blocks; thus, the blocks implicitly define a complete binary tree of depth ℓ.

Now suppose we want to search for a key x. For any integer k, let $B_k(x)$ denote the range of indices for the level-k block containing $H[h(x)]$:

$$B_k(x) = \left[2^k \lfloor h(x)/2^k \rfloor \mathinner{..} 2^k \lfloor h(x)/2^k \rfloor + 2^k - 1\right]$$

Similarly, let $B'_k(x)$ denote the sibling of $B_k(x)$ in the block tree; that is, $B'_k(x) = B_{k+1}(x) \setminus B_k(x)$. We refer to each $B_k(x)$ as an ***ancestor*** of x and each $B'_k(x)$ as an ***uncle*** of x. The proper ancestors of any uncle of x are also proper ancestors of x.

A conservative view of binary probing.

The binary probing algorithm can be recast conservatively as follows. First the algorithm probes $H[h(x)]$; if that cell contains x or is empty, the algorithm halts. Then for each k from 0 to $\ell - 1$, the algorithm probes *every* cell in the uncle block $B'_k(x)$, and then halts if that block contained either x or an empty cell. The actual binary probing algorithm probes the cells in $B'_k(x)$ in a particular order and stops immediately when it finds either x or an empty cell, but for purposes of proving an upper bound, let's assume that the algorithm probes the entire block in some arbitrary order.

```
LooseBinaryProbe(x):
  if H[h(x)] = x
      return True
  if H[h(x)] is empty
      return False
  first ← Dunno
  for k ← 0 to ℓ − 1
      for each index j ∈ B'_k(x) in arbitrary order
          if first ≠ Dunno
              if H[j] = x
                  first ← True
              if H[j] is empty
                  first ← False
      if first ≠ Dunno
          return first
  return Full
```

For purposes of analysis, suppose the target item x is not in the table; the time to search for an item that is in the table can only be faster.) The expected running time of LooseBinaryProbe(x) can be expressed as follows:

$$\mathrm{E}[T(x)] \le \sum_{k=0}^{\ell-1} O(2^k) \cdot \Pr[B'_k(x) \text{ is full}].$$

Assuming ideal random hashing, all blocks at the same level have equal probability of being full. Let F_k denote the probability that $B'_k(x)$ (or any fixed level-k block) is full. Then we have

$$\mathrm{E}[T(x)] \le \sum_{k=0}^{\ell-1} O(2^k) \cdot F_k.$$

Call a level-k block B ***popular*** if there are at least 2^k items y in the table such that $h(y) \in B$. Every popular block is full, but full blocks are not necessarily popular.

If block $B_k(x)$ is full but not popular, then $B_k(x)$ contains at least one item whose hash value is not in $B_k(x)$. Let y be the first such item inserted into the hash table. When y was inserted, some uncle block $B'_j(x) = B_j(y)$ with $j \ge k$ was already full. Let $B'_j(x)$ be the first uncle of $B_k(x)$ to become full. The only blocks that can overflow into $B_j(y)$ are its uncles, which are all either ancestors or uncles of $B_k(x)$. But when $B_j(y)$ became full, no other uncle of $B_k(x)$ was full. Moreover, $B_k(x)$ was not yet full (because there was still room for y), so no ancestor of $B_k(x)$ was full. It follows that $B'_j(x)$ is popular.

We conclude that if a block is full, then either that block or one of its uncles is popular. Thus, if we write P_k to denote the probability that $B'_k(x)$ (or any fixed level-k block) is popular, we have

$$F_k \le 2P_k + \sum_{j>k} P_j.$$

We can crudely bound the probability P_k as follows. Each of the n items in the table hashes into a fixed level-k block with probability $2^k/m$; thus,

$$P_k = \binom{n}{2^k}\left(\frac{2^k}{m}\right)^{2^k} \le \frac{n^{2^k}}{(2^k)!}\frac{2^{k2^k}}{m^{2^k}} < \left(\frac{en}{m}\right)^{2^k}$$

(The last inequality uses a crude form of Stirling's approximation: $n! > n^n/e^n$.) Our assumption $n \le m/4$ implies the simpler inequality $P_k < (e/4)^{2^k}$. Because $e < 4$, it is easy to see that $\boldsymbol{P_k < 4^{-k}}$ for all sufficiently large k.

It follows that $F_k = O(4^{-k})$, which implies that the expected search time is at most $\sum_{k\ge 0} O(2^k) \cdot O(4^{-k}) = \sum_{k\ge 0} O(2^{-k}) = O(1)$. □

In fact, we can prove the same expected time bound with a much weaker randomness requirement.

Lemma 9. *In a hash table of size $m = 2^\ell$ containing $n \le m/4$ keys, built using binary probing, the expected time for any search is $O(1)$,* ***assuming 5-uniform hashing****.*

Proof: Most of the previous proof carries through without modification; the only change is that we need a different argument to bound the probability that $B'_k(x)$ is popular.

For each element $y \ne x$, we define an indicator variable $P_y := [h(y) \in B'_k(x)]$. The uniformity of h implies that $\mathrm{E}[P_y] = \Pr[h(y) \in B'_k(x)] = 2^k/m$, to simplify notation, let $p = 2^k/m$. Now we define a second indicator variable

$$Q_y = P_y - p = \begin{cases} 1-p & \text{if } h(y) \in B'_k(x) \\ -p & \text{otherwise} \end{cases}$$

Linearity of expectation implies that $\mathrm{E}[Q_y] = 0$. Finally, define $P = \sum_{y\ne x} P_y$ and $Q = \sum_{y\ne x} Q_y = P - \mathrm{E}[P]$; again, linearity of expectation gives us $\mathrm{E}[P] = p(n-1) = 2^k(n-1)/m$. We can bound the probability that $B'_k(x)$ is popular in terms of these variables as follows:

$$\begin{aligned}
\Pr[B'_k(x) \text{ is popular}] &= \Pr[P \ge 2^k - 1] && \text{by definition of "popular"} \\
&= \Pr[Q \ge 2^k - 1 - 2^k(n-1)/m] \\
&= \Pr[Q \ge 2^k(1 - n/m - 1/m) - 1] \\
&\le \Pr[Q \ge 2^k(3/4 - 1/m) - 1] && \text{because } n \le m/4 \\
&\le \Pr[Q \ge 2^{k-1}] && \text{because } m \ge 4n \ge 4.
\end{aligned}$$

Now we do something that looks a little weird; instead of considering the variable Q directly, we consider its fourth power. Because Q^4 is non-negative, Markov's inequality gives us

$$\Pr[Q \ge 2^{k-1}] \;=\; \Pr[Q^4 \ge 2^{4(k-1)}] \;\le\; \frac{E[Q^4]}{2^{4(k-1)}}$$

Linearity of expectation implies

$$\mathrm{E}[Q^4] = \sum_{y\ne x}\sum_{z\ne x}\sum_{y'\ne x}\sum_{z'\ne x} \mathrm{E}[Q_y Q_z Q_{y'} Q_{z'}].$$

Because h is 5-uniform, the random variables Q_y are 4-independent. (We lose one level of independence because Q_y depends on both y and the fixed element x.) It follows that if y, z, y', z' are all distinct, then $\mathrm{E}[Q_y Q_z Q_{y'} Q_{z'}] = \mathrm{E}[Q_y]\mathrm{E}[Q_z]\mathrm{E}[Q_{y'}]\mathrm{E}[Q_{z'}] = 0$. More generally, if any one of y, z, y', z' is different from the other three, then $\mathrm{E}[Q_y Q_z Q_{y'} Q_{z'}] = 0$. The expectation $\mathrm{E}[Q_y Q_z Q_{y'} Q_{z'}]$ is only non-zero when $y = z = y' = z'$, or when the values y, z, y', z' consist of two identical pairs.

$$\mathrm{E}[Q^4] = \sum_y \mathrm{E}[Q_y^4] + 6\sum_{y<z} \mathrm{E}[Q_y^2]\mathrm{E}[Q_z^2]$$

The definition of expectation implies

$$\mathrm{E}[Q_y^2] \;=\; p(1-p)^2 + (1-p)(-p)^2 \;=\; p(1-p) \;<\; p$$

and similarly

$$\mathrm{E}[Q_y^4] \;=\; p(1-p)^4 + (1-p)(-p)^4 \;=\; p(1-p)((1-p)^3 + p^3) \;<\; p.$$

It follows that

$$\begin{aligned}\mathrm{E}[Q^4] &< (n-1)p + 6\binom{n-1}{2}p^2 \\ &< \frac{mp}{4} + 3\left(\frac{mp}{4}\right)^2 \\ &< 2^{k-2} + 3\cdot 2^{2(k-2)} \;<\; 2^{2(k-1)}\end{aligned}$$

Putting all the pieces together, we conclude that $\Pr[B'_k(x) \text{ is popular}] \le 2^{-2(k-1)}$. The rest of the proof is unchanged. □

5.11 Cuckoo Hashing

★★★

Write this.

Exercises

1. Your boss wants you to find a *perfect* hash function for mapping a known set of n items into a table of size m. A hash function is *perfect* if there are *no* collisions; each of the n items is mapped to a different slot in the hash table. Of course, a perfect hash function is only possible if $m \ge n$. (This is a different definition of "perfect" than the one considered in the lecture notes.) After cursing your algorithms instructor for not teaching you about (this kind of) perfect hashing, you decide to try something simple: repeatedly pick ideal random hash functions until you find one that happens to be perfect.

   (a) Suppose you pick an ideal random hash function h. What is the *exact* expected number of collisions, as a function of n (the number of items) and m (the size of the table)? Don't worry about how to resolve collisions; just count them.

   (b) What is the *exact* probability that a random hash function is perfect?

   (c) What is the *exact* expected number of different random hash functions you have to test before you find a perfect hash function?

   (d) What is the *exact* probability that none of the first N random hash functions you try is perfect?

   (e) How many ideal random hash functions do you have to test to find a perfect hash function *with high probability*?

2. (a) Describe a set of hash functions that is uniform but not (near-)universal.

   (b) Describe a set of hash functions that is universal but not (near-)uniform.

   (c) Describe a set of hash functions that is universal but (near-)3-universal.

(d) A family of hash function is ***pairwise independent*** if knowing the hash value of any one item gives us absolutely no information about the hash value of any other item; more formally,

$$\Pr_{h \in \mathcal{MB}^+}[h(x) = i \mid h(y) = j] \;=\; \Pr_{h \in \mathcal{MB}^+}[h(x) = i]$$

or equivalently,

$$\Pr_{h \in \mathcal{MB}^+}[(h(x) = i) \wedge (h(y) = j)] = \Pr_{h \in \mathcal{MB}^+}[h(x) = i] \cdot \Pr_{h \in \mathcal{MB}^+}[h(y) = j]$$

for all distinct items $x \neq y$ and all (possibly equal) hash values i and j.

Describe a set of hash functions that is uniform but not pairwise independent.

(e) Describe a set of hash functions that is pairwise independent but not (near-)uniform.

(f) Describe a set of hash functions that is universal but not pairwise independent.

(g) Describe a set of hash functions that is pairwise independent but not (near-)uniform.

(h) Describe a set of hash functions that is universal and pairwise independent but not uniform, or prove no such set exists.

3. (a) Prove that the set $\mathcal{MB}$ of binary multiplicative hash functions described in Section **??** is not uniform. *[Hint: What is $multb_a(0)$?]*

(b) Prove that $\mathcal{MB}$ is not pairwise independent. *[Hint: Compare $multb_a(0)$ and $multb_a(2^{w-1})$.]*

(c) Consider the following variant of multiplicative hashing, which uses slightly longer salt parameters. For any integers $a, b \in [2^{w+\ell}]$ where a is odd, let

$$h_{a,b}(x) \;:=\; \big((a \cdot x + b) \bmod 2^{w+\ell}\big) \operatorname{div} 2^w \;=\; \left\lfloor \frac{(a \cdot x + b) \bmod 2^{w+\ell}}{2^w} \right\rfloor,$$

and let $\mathcal{MB}^+ = \{h_{a,b} \mid a, b \in [2^{w+\ell}] \text{ and } a \text{ odd}\}$. Prove that the family of hash functions $\mathcal{MB}^+$ is ***strongly near-universal***:

$$\Pr_{h \in \mathcal{MB}^+}\big[(h(x) = i) \wedge (h(y) = j)\big] \leq \frac{2}{m^2}$$

for all items $x \neq y$ and all (possibly equal) hash values i and j.

4. Suppose we are using an *open-addressed* hash table of size m to store n items, where $n \leq m/2$. Assume an ideal random hash function. For any i, let X_i denote the number of probes required for the ith insertion into the table, and let $X = \max_i X_i$ denote the length of the longest probe sequence.

(a) Prove that $\Pr[X_i > k] \leq 1/2^k$ for all i and k.

(b) Prove that $\Pr[X_i > 2 \lg n] \leq 1/n^2$ for all i.

(c) Prove that $\Pr[X > 2 \lg n] \leq 1/n$.

(d) Prove that $\mathrm{E}[X] = O(\log n)$.

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://www.cs.uiuc.edu/~jeffe/teaching/algorithms for the most recent revision.

Casus ubique valet; semper tibi pendeat hamus:
Quo minime credas gurgite, piscis erit.
[Luck affects everything. Let your hook always be cast.
Where you least expect it, there will be a fish.]
— Publius Ovidius Naso [Ovid], *Ars Amatoria*, Book III (2 AD)

There is no sense being precise
when you don't even know what you're talking about.
— Attributed to John von Neumann

6 Filtering and Streaming

The randomized algorithms and data structures we have seen so far *always* produce the correct answer but have a small probability of being slow. In this lecture, we will consider randomized algorithms that are always fast, but return the wrong answer with some small probability.[1] More generally, we are interested in tradeoffs between the (likely) efficiency of the algorithm and the (likely) quality of its output.

Specifically, we introduce an ***error rate*** δ and analyze the running time required to guarantee the output is correct with probability $1-\delta$. For "high probability" correctness, we need $\delta < 1/n^c$ for some constant c. In practice, it may be sufficient (or even necessary) to set δ to a small constant; for example, setting $\delta = 1/1000$ means the algorithm produces correct results at least 99.9% of the time.

6.1 Bloom Filters

Bloom filters are a natural variant of hashing proposed by Burton Bloom in 1970 as a mechanism for supporting membership queries in sets. In strict accordance with Stigler's Law of Autonomy, Bloom filters are identical to ***Zatocoding***, a coding system for library cards developed by Calvin Mooers in 1947. (Mooers was the person who coined the phrase "information retrieval".) A probabilistic analysis of Zatocoding appears in the personal notes of cybernetics pioneer W. Ross Ashby from 1960.

A Bloom filter (or Zatocode) for a set X of n items from some universe $\mathcal{U}$ allows one to test whether a given item $x \in \mathcal{U}$ is an element of X. Of course we can already do this with hash tables in $O(1)$ expected time, using $O(n)$ space. Bloom (and Mooers) observed that by allowing false positives—occasionally reporting $x \in X$ when in fact $x \notin X$—we can still answer queries in $O(1)$ expected time using considerably less space. False positives make Bloom filters unsuitable as an exact membership data structure, but because of their speed and low false positive rate, they are commonly used as filters or sanity checks for more complex data structures.

A Bloom filter consists of an array $B[0..m-1]$ of bits, together with k hash functions $h_1, h_2, \ldots, h_k : \mathcal{U} \to \{0, 1, \ldots, m-1\}$. For purposes of theoretical analysis, we assume the hash functions h_i are mutually independent, ideal random functions. This assumption is of course unsupportable in practice, but may be necessary to guarantee theoretical performance. Unlike many other types of hashing, nobody knows whether the same theoretical guarantees can be achieved using practical hash functions with more limited independence.[2] Fortunately, the actual

[1] Some textbooks (and Wikipedia) use the terms "Las Vegas" and "Monte Carlo" algorithms to respectively describe these two types of randomized algorithms. Most algorithms researchers don't.

[2] PhD thesis, anyone?

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

real-world behavior of Bloom filters appears to be consistent with this unrealistic theoretical analysis.

A Bloom filter for a set $X = \{x_1, x_2, \ldots, x_n\}$ is initialized by setting the bit $B[h_j(x_i)]$ to 1 for all indices i and j. Because of collisions, some bits may be set more than once, but that's fine.

```
MakeBloomFilter(X):
  for h ← 0 to m − 1
      B[h] ← 0
  for i ← 1 to n
      for j ← 1 to k
          B[h_j(x_i)] ← 1
  return B
```

Given a new item y, the Bloom filter determines whether $y \in X$ by checking each bit $B[h_j(y)]$. If any of those bits is 0, the Bloom filter correctly reports that $y \notin X$ However, if all bits are 1, the Bloom filter reports that $y \in X$, although this is not necessarily correct.

```
BloomMembership(B, y):
  for j ← 1 to k
      if B[h_j(y)] = 0
          return False
  return Maybe
```

One nice feature of Bloom filters is that the various hash functions h_i can be evaluated in parallel on a multicore machine.

6.2 False Positive Rate

Let's estimate the probability of a false positive, as a function of the various parameters n, m, and k. For all indices h, i, and j, we have $\Pr[h_j(x_i) = h] = 1/m$, so ideal randomness gives us

$$Pr[B[h] = 0] \;=\; \left(1 - \frac{1}{m}\right)^{kn} \;\approx\; e^{-kn/m}$$

for every index h. Using this exact probability is rather unwieldy; to keep things sane, we will use the close approximation $\boldsymbol{p := e^{-kn/m}}$ instead.

The expected number of 0-bits in the Bloom filter is approximately mp; moreover, Chernoff bounds imply that the number of 0-bits is close to mp with very high probability. Thus, the probabililty of a false positive is very close[3] to

$$(1-p)^k = (1 - e^{-kn/m})^k.$$

If all other parameters are held constant, then the false positive rate increases with n (the number of items) and decreases with m (the number of bits). The dependence on k (the number of hash functions) is a bit more complicated, but we can derive the best value of k for given n and m as follows. Consider the logarithm of the false-positive rate:

$$\ln\left((1-p)^k\right) = k\ln(1-p) = -\frac{m}{n}\ln p \ln(1-p).$$

By symmetry, this expression is minimized when $\boldsymbol{p = 1/2}$. We conclude that, the optimal number of hash functions is $\boldsymbol{k = \ln 2 \cdot (m/n)}$, which would give us the false positive rate

[3]This analysis, originally due to Bloom, assumes that certain events are independent even though they are not; as a result, the estimate given here is slightly below the true false positive rate.

$(1/2)^{\ln 2(m/n)} \approx (0.61850)^{m/n}$. Of course, in practice, k must be an integer, so we cannot achieve precisely this rate, but we can get reasonably close (at least if $m \gg n$).

Finally, the previous analysis implies that we can achieve any desired false positive rate $\delta > 0$ using a Bloom filter of size

$$m = \left\lceil \frac{\lg(1/\delta)}{\ln 2} n \right\rceil = \Theta(n \log(1/\delta))$$

that uses with $k = \lceil \lg(1/\delta) \rceil$ hash functions. For example, we can achieve a 1% false-positive rate using a Bloom filter of size $10n$ bits with 7 hash functions; in practice, this is *considerably* fewer bits than we would need to store all the elements of S explicitly. With a $32n$-bit table (equivalent to one integer per item) and 22 hash functions, we get a false positive rate of just over $2 \cdot 10^{-7}$.

★★★ Deletions via counting filters? Key-value pairs via Bloomier filters? Other extensions?

6.3 Streaming Algorithms

A ***data stream*** is an extremely long sequence S of items from some universe $\mathcal{U}$ that can be read only once, in order. Good examples of data streams include the sequence of packets that pass through a network router, the sequence of searches at google.com, the sequence of all bids on the New York Stock Exchange, and the sequence of humans passing through the Shinjuku Railway Station in Tokyo. Standard algorithms are not appropriate for data streams; there is simply too much data to store, and it arrives too quickly for any complex computations.

A ***streaming algorithm*** processes each item in a data stream stream as it arrives, maintaining some summary information in a local data structure. The basic structure of every streaming algorithm is the following:

```
DoSomething(S):
  ⟨⟨initialize⟩⟩
  while S is not done
      x ← next item in S
      ⟨⟨do something fast with x⟩⟩
  return ⟨⟨something⟩⟩
```

Ideally, neither the running time per item nor the space used by the data structure depends on the overall length of the stream; viewed as algorithms in the traditional sense, all streaming algorithms run in constant time! Somewhat surprisingly, even within this very restricted model, it is possible to compute interesting properties of the stream using randomization, provided we are willing to tolerate some errors in the output.

6.4 The Count-Min Sketch

As an example, consider the following problem: At any point during the stream, we want to estimate the number of times that an arbitrary item $x \in \mathcal{U}$ has appeared in the stream so far. This problem can be solved with a variant of Bloom filters called the ***count-min sketch***, first published by Graham Cormode and S. Muthu Muthukrishnan in 2005.

The count-min sketch consists of a $w \times d$ array of counters (all initially zero) and d hash functions $h_1, h_2, \ldots, h_d : \mathcal{U} \to [m]$, drawn independently and uniformly at random from a 2-uniform family of hash functions. Each time an item x arrives in the stream, we call CMIncrement(x). Whenever we want to estimate the number of occurrences of an item x so far, we call CMEstimate(x).

```
CMIncrement(x):
  for i ← 1 to d
      j ← h_i(x)
      Count[i, j] ← Count[i, j] + 1
```

```
CMEstimate(x):
  est ← ∞
  for i ← 1 to d
      j ← h_i(x)
      est ← min {est, Count[i, j]}
  return est
```

If we set $w := \lceil e/\varepsilon \rceil$ and $d := \lceil \ln(1/\delta) \rceil$, then the data structure uses $O(\frac{1}{\varepsilon} \log \frac{1}{\delta})$ space and processes updates and queries in $O(\log \frac{1}{\delta})$ time.

Let f_x be the true frequency (number of occurrences) of x, and let $\hat{f}_x$ be the value returned by CMEstimate. It is easy to see that $f_x \leq \hat{f}_x$; we can never return a value smaller than the actual number of occurrences of x. We claim that $\mathbf{Pr}[\hat{f}_x > f_x + \varepsilon N] < \delta$, where N is the total length of the stream. In other words, our estimate is never too small, and with high probability, it isn't a significant overestimate either. (Notice that the error here is additive; the estimates or truly infrequent items may be much larger than their true frequencies.)

For any items $x \neq y$ and any index j, we define an indicator variable $X_{i,x,y} = [h_i(x) = h_i(y)]$; because the hash functions h_i are universal, we have

$$\mathrm{E}[X_{i,x,y}] = \Pr[h_i(x) = h_i(y)] = \frac{1}{w}.$$

Let $X_{i,x} := \sum_{y \neq x} X_{i,x,y} \cdot f_y$ denote the total number of collisions with x in row i of the table. Then we immediately have

$$Count[i, h_i(x)] \;=\; f_x + X_{i,x} \;\geq\; f_x.$$

On the other hand, linearity of expectation implies

$$\mathrm{E}[X_{i,x}] \;=\; \sum_{y \neq x} \mathrm{E}[X_{i,x,y}] \cdot f_y \;=\; \frac{1}{w} \sum_{y \neq x} f_y \;\leq\; \frac{N}{w}.$$

Now Markov's inequality implies

$$\begin{aligned}
\Pr[\hat{f}_x > f_x + \varepsilon N] &= \Pr[X_{i,x} > \varepsilon N \text{ for all i}] && \text{[definition]} \\
&= \Pr[X_{1,x} > \varepsilon N]^d && \text{[independence of } h_i\text{'s]} \\
&\leq \left(\frac{\mathrm{E}[X_{1,x}]}{\varepsilon N} \right)^d && \text{[Markov's inequality]} \\
&\leq \left(\frac{N/w}{\varepsilon N} \right)^d = \left(\frac{1}{w\varepsilon} \right)^d && \text{[derived earlier]}
\end{aligned}$$

Now setting $w = \lceil e/\varepsilon \rceil$ and $d = \lceil \ln(1/\delta) \rceil$ gives us $\Pr[\hat{a}_x > a_x + \varepsilon N] \leq (1/e)^{\ln(1/\delta)} = \delta$, as claimed.

6.5 Estimating Distinct Items

★★★

Write this, possibly as a simpler replacement for the count-min sketch. AMS estimator: $2^{z+1/2}$ where $z = \max\{zeros(x) \mid x \in S\}$? Or stick with the Flajolet-Martin/Bar-Yossef-et-al estimator in the exercises? Median amplification.

Estimating larger moments?

Exercises

1. ***Reservoir sampling*** is a method for choosing an item uniformly at random from an arbitrarily long stream of data; for example, the sequence of packets that pass through a router, or the sequence of IP addresses that access a given web page. Like all data stream algorithms, this algorithm must process each item in the stream quickly, using very little memory.

```
GetOneSample(stream S):
  ℓ ← 0
  while S is not done
      x ← next item in S
      ℓ ← ℓ + 1
      if Random(ℓ) = 1
          sample ← x        (⋆)
  return sample
```

At the end of the algorithm, the variable ℓ stores the length of the input stream S; this number is *not* known to the algorithm in advance. If S is empty, the output of the algorithm is (correctly!) undefined. In the following, consider an arbitrary non-empty input stream S, and let n denote the (unknown) length of S.

(a) Prove that the item returned by GetOneSample(S) is chosen uniformly at random from S.

(b) What is the *exact* expected number of times that GetOneSample(S) executes line ($\star$)?

(c) What is the *exact* expected value of ℓ when GetOneSample(S) executes line ($\star$) for the *last* time?

(d) What is the *exact* expected value of ℓ when either GetOneSample(S) executes line ($\star$) for the *second* time (or the algorithm ends, whichever happens first)?

(e) Describe and analyze an algorithm that returns a subset of k distinct items chosen uniformly at random from a data stream of length at least k. The integer k is given as part of the input to your algorithm. Prove that your algorithm is correct.

For example, if $k = 2$ and the stream contains the sequence $\langle \spadesuit, \heartsuit, \diamondsuit, \clubsuit \rangle$, the algorithm should return the subset $\{\diamondsuit, \spadesuit\}$ with probability $1/6$.

2. In this problem, we will derive a simple streaming algorithm (first published by Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan in 2002) to estimate the number of distinct items in a data stream S.

Suppose S contains n unique items (but possibly several copies of each item); as usual, the algorithm does *not* know n in advance. Given an accuracy parameter $0 < \varepsilon < 1$ and a confidence parameter $0 < \delta < 1$ as part of the input, our final algorithm will compute an estimate $\widetilde{N}$ such that $\Pr[|\widetilde{N} - n| > \varepsilon n] < \delta$.

As a first step, fix a positive integer m that is large enough that we don't have to worry about round-off errors in the analysis. Our first algorithm chooses a hash function $h\colon \mathcal{U} \to [m]$ at random from a ***2-uniform*** family, computes the minimum hash value $\hbar = \min\{h(x) \mid x \in S\}$, and finally returns the estimate $\widetilde{n} = m/\hbar$.

(a) Prove that $\Pr[\widetilde{n} > (1+\varepsilon)n] \leq 1/(1+\varepsilon)$. *[Hint: Markov's inequality]*

(b) Prove that $\Pr[\widetilde{n} < (1-\varepsilon)n] \leq 1-\varepsilon$. *[Hint: Chebyshev's inequality]*

(c) We can improve this estimator by maintaining the k smallest hash values, for some integer $k > 1$. Let $\widetilde{n}_k = k \cdot m/\hbar_k$, where $\hbar_k$ is the kth smallest element of $\{h(x) \mid x \in S\}$.

Estimate the smallest value of k (as a function of the accuracy parameter ε) such that $\Pr[|\widetilde{n}_k - n| > \varepsilon n] \leq 1/4$.

(d) Now suppose we run d copies of the previous estimator in parallel to generate d independent estimates $\widetilde{n}_{k,1}, \widetilde{n}_{k,2}, \ldots, \widetilde{n}_{k,d}$, for some integer $d > 1$. Each copy uses its own independently chosen hash function, but they all use the same value of k that you derived in part (c). Let $\widetilde{N}$ be the *median* of these d estimates.

Estimate the smallest value of d (as a function of the confidence parameter δ) such that $\Pr[|\widetilde{N} - n| > \varepsilon n] \leq \delta$.

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

Philosophers gathered from far and near
To sit at his feet and hear and hear,
Though he never was heard
To utter a word
But "Abracadabra, abracadab,
Abracada, abracad,
Abraca, abrac, abra, ab!"
'Twas all he had,
'Twas all they wanted to hear, and each
Made copious notes of the mystical speech,
Which they published next –
A trickle of text
In the meadow of commentary.
Mighty big books were these,
In a number, as leaves of trees;
In learning, remarkably – very!

— Jamrach Holobom, quoted by Ambrose Bierce,
The Devil's Dictionary (1911)

Why are our days numbered and not, say, lettered?

— Woody Allen, "Notes from the Overfed", *The New Yorker* (March 16, 1968)

7 String Matching

7.1 Brute Force

The basic object that we consider in this lecture note is a *string*, which is really just an array. The elements of the array come from a set Σ called the *alphabet*; the elements themselves are called *characters*. Common examples are ASCII text, where each character is an seven-bit integer, strands of DNA, where the alphabet is the set of nucleotides $\{A, C, G, T\}$, or proteins, where the alphabet is the set of 22 amino acids.

The problem we want to solve is the following. Given two strings, a ***text*** $T[1..n]$ and a ***pattern*** $P[1..m]$, find the first *substring* of the text that is the same as the pattern. (It would be easy to extend our algorithms to find *all* matching substrings, but we will resist.) A substring is just a contiguous subarray. For any *shift* s, let T_s denote the substring $T[s..s+m-1]$. So more formally, we want to find the smallest shift s such that $T_s = P$, or report that there is no match. For example, if the text is the string 'AMANAPLANACATACANALPANAMA'[1] and the pattern is 'CAN', then the output should be 15. If the pattern is 'SPAM', then the answer should be NONE.

[1]Dan Hoey (or rather, his computer program) found the following 540-word palindrome in 1984. We have better online dictionaries now, so I'm sure you could do better.

A man, a plan, a caret, a ban, a myriad, a sum, a lac, a liar, a hoop, a pint, a catalpa, a gas, an oil, a bird, a yell, a vat, a caw, a pax, a wag, a tax, a nay, a ram, a cap, a yam, a gay, a tsar, a wall, a car, a luger, a ward, a bin, a woman, a vassal, a wolf, a tuna, a nit, a pall, a fret, a watt, a bay, a daub, a tan, a cab, a datum, a gall, a hat, a fag, a zap, a say, a jaw, a lay, a wet, a gallop, a tug, a trot, a trap, a tram, a torr, a caper, a top, a tonk, a toll, a ball, a fair, a sax, a minim, a tenor, a bass, a passer, a capital, a rut, an amen, a ted, a cabal, a tang, a sun, an ass, a maw, a sag, a jam, a dam, a sub, a salt, an axon, a sail, an ad, a wadi, a radian, a room, a rood, a rip, a tad, a pariah, a revel, a reel, a reed, a pool, a plug, a pin, a peek, a parabola, a dog, a pat, a cud, a nu, a fan, a pal, a rum, a nod, an eta, a lag, an eel, a batik, a mug, a mot, a nap, a maxim, a mood, a leek, a grub, a gob, a gel, a drab, a citadel, a total, a cedar, a tap, a gag, a rat, a manor, a bar, a gal, a cola, a pap, a yaw, a tab, a raj, a gab, a nag, a pagan, a bag, a jar, a bat, a way, a papa, a local, a gar, a baron, a mat, a rag, a gap, a tar, a decal, a tot, a led, a tic, a bard, a leg, a bog, a burg, a keel, a doom, a mix, a map, an atom, a gum, a kit, a baleen, a gala, a ten, a don, a mural, a pan, a faun, a ducat, a pagoda, a lob, a rap, a keep, a nip, a gulp, a loop, a deer, a leer, a lever, a hair, a pad, a tapir, a door, a moor, an aid, a raid, a wad, an alias, an ox, an atlas, a bus, a madam, a jag, a saw, a mass, an anus, a gnat, a lab, a cadet, an em, a natural, a tip, a caress, a pass, a baronet, a minimax, a sari, a fall, a ballot, a knot, a pot, a rep, a carrot, a mart, a part, a tort, a gut, a poll, a gateway, a law, a jay, a sap, a zag, a fat, a hall, a gamut, a dab, a can, a tabu, a day, a batt, a waterfall, a patina, a nut, a flow, a lass, a van, a mow, a nib, a draw, a regular, a call, a war, a stay, a gam, a yap, a cam, a ray, an ax, a tag, a wax, a paw, a cat, a valley, a drib, a lion, a saga, a plat, a catnip, a pooh, a rail, a calamus, a dairyman, a bater, a canal—Panama!

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/ for the most recent revision.

In most cases the pattern is much smaller than the text; to make this concrete, I'll assume that $m < n/2$.

Here's the 'obvious' brute force algorithm, but with one immediate improvement. The inner while loop compares the substring T_s with P. If the two strings are not equal, this loop stops at the first character mismatch.

```
AlmostBruteForce(T[1..n], P[1..m]):
  for s ← 1 to n − m + 1
      equal ← True
      i ← 1
      while equal and i ≤ m
          if T[s + i − 1] ≠ P[i]
              equal ← False
          else
              i ← i + 1
      if equal
          return s
  return None
```

In the worst case, the running time of this algorithm is $O((n-m)m) = O(nm)$, and we can actually achieve this running time by searching for the pattern `AAA...AAAB` with $m-1$ `A`'s, in a text consisting of n `A`'s.

In practice, though, breaking out of the inner loop at the first mismatch makes this algorithm quite practical. We can wave our hands at this by assuming that the text and pattern are both random. Then on average, we perform a constant number of comparisons at each position i, so the total expected number of comparisons is $O(n)$. Of course, neither English nor DNA is really random, so this is only a heuristic argument.

7.2 Strings as Numbers

For the moment, let's assume that the alphabet consists of the ten digits `0` through `9`, so we can interpret any array of characters as either a string or a decimal number. In particular, let p be the numerical value of the pattern P, and for any shift s, let t_s be the numerical value of T_s:

$$p = \sum_{i=1}^{m} 10^{m-i} \cdot P[i] \qquad t_s = \sum_{i=1}^{m} 10^{m-i} \cdot T[s+i-1]$$

For example, if $T = \texttt{31415926535897932}\underline{\texttt{3846}}\texttt{26433832795028841971}$ and $m = 4$, then $t_{17} = 2384$.

Clearly we can rephrase our problem as follows: Find the smallest s, if any, such that $p = t_s$. We can compute p in $O(m)$ arithmetic operations, without having to explicitly compute powers of ten, using *Horner's rule*:

$$p = P[m] + 10\big(P[m-1] + 10\big(P[m-2] + \cdots + 10\big(P[2] + 10 \cdot P[1]\big)\cdots\big)\big)$$

We could also compute any t_s in $O(m)$ operations using Horner's rule, but this leads to essentially the same brute-force algorithm as before. But once we know t_s, we can actually compute t_{s+1} in constant time just by doing a little arithmetic — subtract off the most significant digit $T[s] \cdot 10^{m-1}$, shift everything up by one digit, and add the new least significant digit $T[r+m]$:

$$t_{s+1} = 10\big(t_s - 10^{m-1} \cdot T[s]\big) + T[s+m]$$

To make this fast, we need to precompute the constant 10^{m-1}. (And we know how to do that quickly, right?) So at least intuitively, it looks like we can solve the string matching problem in $O(n)$ worst-case time using the following algorithm:

```
NumberSearch(T[1..n], P[1..m]):
  σ ← 10^(m−1)
  p ← 0
  t_1 ← 0
  for i ← 1 to m
      p ← 10 · p + P[i]
      t_1 ← 10 · t_1 + T[i]
  for s ← 1 to n − m + 1
      if p = t_s
          return s
      t_{s+1} ← 10 · (t_s − σ · T[s]) + T[s + m]
  return None
```

Unfortunately, the most we can say is that the number of *arithmetic operations* is $O(n)$. These operations act on numbers with up to m digits. Since we want to handle arbitrarily long patterns, we can't assume that each operation takes only constant time! In fact, if we want to avoid expensive multiplications in the second-to-last line, we should represent each number as a string of decimal digits, which brings us back to our original brute-force algorithm!

7.3 Karp-Rabin Fingerprinting

To make this algorithm efficient, we will make one simple change, proposed by Richard Karp and Michael Rabin in 1981:

$$\boxed{\text{Perform all arithmetic modulo some prime number } q.}$$

We choose q so that the value $10q$ fits into a standard integer variable, so that we don't need any fancy long-integer data types. The values $(p \bmod q)$ and $(t_s \bmod q)$ are called the *fingerprints* of P and T_s, respectively. We can now compute $(p \bmod q)$ and $(t_1 \bmod q)$ in $O(m)$ *time* using Horner's rule:

$$p \bmod q = P[m] + \big(\cdots + \big(10 \cdot \big(P[2] + \big(10 \cdot P[1] \bmod q\big) \bmod q\big) \bmod q\big) \cdots\big)\big) \bmod q.$$

Similarly, given $(t_s \bmod q)$, we can compute $(t_{s+1} \bmod q)$ in constant time as follows:

$$t_{s+1} \bmod q = \big(10 \cdot \big(t_s - \big(\big(10^{m-1} \bmod q\big) \cdot T[s] \bmod q\big) \bmod q\big) \bmod q\big) + T[s+m] \bmod q.$$

Again, we have to precompute the value $(10^{m-1} \bmod q)$ to make this fast.

If $(p \bmod q) \neq (t_s \bmod q)$, then certainly $P \neq T_s$. However, if $(p \bmod q) = (t_s \bmod q)$, we can't tell whether $P = T_s$ or not. All we know for sure is that p and t_s differ by some integer multiple of q. If $P \neq T_s$ in this case, we say there is a *false match* at shift s. To test for a false match, we simply do a brute-force string comparison. (In the algorithm below, $\tilde{p} = p \bmod q$ and $\tilde{t}_s = t_s \bmod q$.) The overall running time of the algorithm is $O(n + Fm)$, where F is the number of false matches.

Intuitively, we expect the fingerprints t_s to jump around between 0 and $q-1$ more or less at random, so the 'probability' of a false match 'ought' to be $1/q$. This intuition implies that

$F = n/q$ "on average", which gives us an 'expected' running time of $O(n + nm/q)$. If we always choose $q \geq m$, this bound simplifies to $O(n)$.

But of course all this intuitive talk of probabilities is meaningless hand-waving, since we haven't actually done anything random yet! There are two simple methods to formalize this intuition.

Random Prime Numbers

The algorithm that Karp and Rabin actually proposed chooses the prime modulus q *randomly* from a sufficiently large range.

```
KarpRabin(T[1..n], P[1..m]):
  q ← a random prime number between 2 and ⌈m² lg m⌉
  σ ← 10^(m−1) mod q
  p̃ ← 0
  t̃_1 ← 0
  for i ← 1 to m
      p̃ ← (10 · p̃ mod q) + P[i] mod q
      t̃_1 ← (10 · t̃_1 mod q) + T[i] mod q
  for s ← 1 to n − m + 1
      if p̃ = t̃_s
          if P = T_s        ⟨⟨brute-force O(m)-time comparison⟩⟩
              return s
      t̃_{s+1} ← (10 · (t̃_s − (σ · T[s] mod q) mod q) mod q) + T[s + m] mod q
  return None
```

For any positive integer u, let $\pi(u)$ denote the number of prime numbers less than u. There are $\pi(m^2 \log m)$ possible values for q, each with the same probability of being chosen. Our analysis needs two results from number theory. I won't even try to prove the first one, but the second one is quite easy.

Lemma 1 (The Prime Number Theorem). $\pi(u) = \Theta(u/\log u)$.

Lemma 2. *Any integer x has at most $\lfloor \lg x \rfloor$ distinct prime divisors.*

Proof: If x has k distinct prime divisors, then $x \geq 2^k$, since every prime number is bigger than 1. □

Suppose there are no true matches, since a true match can only end the algorithm early, so $p \neq t_s$ for all s. There is a false match at shift s if and only if $\tilde{p} = \tilde{t}_s$, or equivalently, if q is one of the prime divisors of $|p - t_s|$. Because $p < 10^m$ and $t_s < 10^m$, we must have $|p - t_s| < 10^m$. Thus, Lemma 2 implies that $|p - t_s|$ has at most $O(m)$ prime divisors. We chose q randomly from a set of $\pi(m^2 \log m) = \Omega(m^2)$ prime numbers, so the probability of a false match at shift s is $O(1/m)$. Linearity of expectation now implies that the expected number of false matches is $O(n/m)$. We conclude that KarpRabin runs in $O(n + \mathrm{E}[F]m) = \boldsymbol{O(n)}$ ***expected time***.

Actually choosing a random prime number is not particularly easy; the best method known is to repeatedly generate a random integer and test whether it's prime. The Prime Number Theorem implies that we will find a prime number after $O(\log m)$ iterations. Testing whether a number x is prime by brute force requires roughly $O(\sqrt{x})$ divisions, each of which require $O(\log^2 x)$ time if we use standard long division. So the total time to choose q using this brute-force method

is about $O(m \log^3 m)$. There are faster algorithms to test primality, but they are considerably more complex. In practice, it's enough to choose a random *probable* prime. Unfortunately, even describing what the phrase "probable prime" means is beyond the scope of this note.

Polynomial Hashing

A much simpler method relies on a classical string-hashing technique proposed by Lawrence Carter and Mark Wegman in the late 1970s. Instead of generating the prime modulus randomly, we generate *the radix of our number representation* randomly. Equivalently, we treat each string as the coefficient vector of a polynomial of degree $m-1$, and we evaluate that polynomial at some random number.

```
CarterWegmanKarpRabin(T[1..n], P[1..m]):
  q ← an arbitrary prime number larger than m²
  b ← Random(q) − 1          ⟨⟨uniform between 0 and q − 1⟩⟩
  σ ← b^(m−1) mod q
  p̃ ← 0
  t̃_1 ← 0
  for i ← 1 to m
      p̃ ← (b · p̃ mod q) + P[i] mod q
      t̃_1 ← (b · t̃_1 mod q) + T[i] mod q
  for s ← 1 to n − m + 1
      if p̃ = t̃_s
          if P = T_s          ⟨⟨brute-force O(m)-time comparison⟩⟩
              return s
      t̃_{s+1} ← (b · (t̃_s − (σ · T[s] mod q) mod q) mod q) + T[s + m] mod q
  return None
```

Fix an arbitrary prime number $q \geq m^2$, and choose b uniformly at random from the set $\{0, 1, \ldots, q-1\}$. We redefine the numerical values p and t_s using b in place of the alphabet size:

$$p(b) = \sum_{i=1}^{m} b^i \cdot P[m-i] \qquad t_s(b) = \sum_{i=1}^{m} b^i \cdot T[s-1+m-i],$$

Now define $\tilde{p}(b) = p(b) \bmod q$ and $\tilde{t}_s(b) = t_s(b) \bmod q$.

The function $f(b) = \tilde{p}(b) - \tilde{t}_s(b)$ is a polynomial of degree $m-1$ over the *variable* b. Because q is prime, the set $\mathbb{Z}_q = \{0, 1, \ldots, q-1\}$ with addition and multiplication modulo q defines a *field*. A standard theorem of abstract algebra states that any polynomial with degree $m-1$ over a field has at most $m-1$ roots in that field. Thus, there are at most $m-1$ elements $b \in \mathbb{Z}_q$ such that $f(b) = 0$.

It follows that if $P \neq T_s$, the probability of a false match at shift s is $\Pr_b[\tilde{p}(b) = \tilde{t}_s(b)] \leq (m-1)/q < 1/m$. Linearity of expectation now implies that the expected number of false positives is $O(n/m)$, so the modified Rabin-Karp algorithm also runs in ***$O(n)$ expected time***.

7.4 Redundant Comparisons

Let's go back to the character-by-character method for string matching. Suppose we are looking for the pattern 'ABRACADABRA' in some longer text using the (almost) brute force algorithm described in the previous lecture. Suppose also that when $s = 11$, the substring comparison fails at the fifth position; the corresponding character in the text (just after the vertical line below) is

not a C. At this point, our algorithm would increment s and start the substring comparison from scratch.

```
HOCUSPOCUSABRA|BRACADABRA...
        ABRA|~~CADABRA~~
         ABR|ACADABRA
```

If we look carefully at the text and the pattern, however, we should notice right away that there's no point in looking at $s = 12$. We already know that the next character is a B — after all, it matched $P[2]$ during the previous comparison — so why bother even looking there? Likewise, we already know that the next two shifts $s = 13$ and $s = 14$ will also fail, so why bother looking there?

```
HOCUSPOCUSABRA|BRACADABRA...
        ABRA|~~CADABRA~~
         ~~A~~~~BR~~|~~ACADABRA~~
           ~~A~~~~B~~|~~RACADABRA~~
             A|BRACADABRA
```

Finally, when we get to $s = 15$, we can't immediately rule out a match based on earlier comparisons. However, for precisely the same reason, we shouldn't start the substring comparison over from scratch — we already know that $T[15] = P[4] = \mathsf{A}$. Instead, we should start the substring comparison at the *second* character of the pattern, since we don't yet know whether or not it matches the corresponding text character.

If you play with this idea long enough, you'll notice that the character comparisons should always advance through the text. **Once we've found a match for a text character, we never need to do another comparison with that text character again.** In other words, we should be able to optimize the brute-force algorithm so that it always *advances* through the text.

You'll also eventually notice a good rule for finding the next 'reasonable' shift s. A *prefix* of a string is a substring that includes the first character; a *suffix* is a substring that includes the last character. A prefix or suffix is *proper* if it is not the entire string. Suppose we have just discovered that $T[i] \neq P[j]$. **The next reasonable shift is the smallest value of s such that $T[s\,..\,i-1]$, which is a suffix of the previously-read text, is also a proper prefix of the pattern.**

in 1977, Donald Knuth, James Morris, and Vaughn Pratt published a string-matching algorithm that implements both of these ideas.

7.5 Finite State Machines

We can interpret any string matching algorithm that always advance through the text as feeding the text through a special type of ***finite-state machine***. A finite state machine is a directed graph. Each node (or ***state***) in the string-matching machine is labeled with a character from the pattern, except for two special nodes labeled ⓢ and ⓘ. Each node has two outgoing edges, a ***success*** edge and a ***failure*** edge. The success edges define a path through the characters of the pattern in order, starting at ⓢ and ending at ⓘ. Failure edges always point to earlier characters in the pattern.

We use the finite state machine to search for the pattern as follows. At all times, we have a current text character $T[i]$ and a current node in the graph, which is usually labeled by some pattern character $P[j]$. We iterate the following rules:

- If $T[i] = P[j]$, or if the current label is ⓢ, follow the success edge to the next node and increment i. (So there is no failure edge from the start node ⓢ.)

A finite state machine for the string ABRACADABRA.
Thick arrows are the success edges; thin arrows are the failure edges.

- If $T[i] \neq P[j]$, follow the failure edge back to an earlier node, but do not change i.

For the moment, let's simply assume that the failure edges are defined correctly—we'll see how to do that later. If we ever reach the node labeled ⓘ, then we've found an instance of the pattern in the text, and if we run out of text characters ($i > n$) before we reach ⓘ, then there is no match.

The finite state machine is really just a (very!) convenient metaphor. In a real implementation, we would not construct the entire graph. Since the success edges always traverse the pattern characters in order, and each state has exactly one outgoing failure edge, we only have to remember the targets of the failure edges. We can encode this ***failure function*** in an array $fail[1..n]$, where for each index j, the failure edge from node j leads to node $fail[j]$. Following a failure edge back to an earlier state corresponds exactly, in our earlier formulation, to shifting the pattern forward. The failure function $fail[j]$ tells us how far to shift after a character mismatch $T[i] \neq P[j]$. Here's the actual algorithm:

```
KnuthMorrisPratt(T[1..n], P[1..m]):
  j ← 1
  for i ← 1 to n
      while j > 0 and T[i] ≠ P[j]
          j ← fail[j]
      if j = m        ⟨⟨Found it!⟩⟩
          return i − m + 1
      j ← j + 1
  return None
```

Before we discuss computing the failure function, let's analyze the running time of KnuthMorrisPratt under the assumption that a correct failure function is already known. At each character comparison, either we increase i and j by one, or we decrease j and leave i alone. We can increment i at most $n-1$ times before we run out of text, so there are at most $n-1$ successful comparisons. Similarly, there can be at most $n-1$ failed comparisons, since the number of times we decrease j cannot exceed the number of times we increment j. In other words, we can amortize character mismatches against earlier character matches. Thus, the total number of character comparisons performed by KnuthMorrisPratt in the worst case is $O(n)$.

7.6 Computing the Failure Function

We can now rephrase our second intuitive rule about how to choose a reasonable shift after a character mismatch $T[i] \neq P[j]$:

$$\boxed{P[1\,..\,fail[j]-1] \text{ is the longest proper prefix of } P[1\,..\,j-1] \text{ that is also a suffix of } T[1\,..\,i-1].}$$

Notice, however, that if we are comparing $T[i]$ against $P[j]$, then we must have already matched the first $j-1$ characters of the pattern. In other words, we already know that $P[1\,..\,j-1]$ is a suffix of $T[1\,..\,i-1]$. Thus, we can rephrase the prefix-suffix rule as follows:

$$\boxed{P[1\,..\,fail[j]-1] \text{ is the longest proper prefix of } P[1\,..\,j-1] \text{ that is also a suffix of } P[1\,..\,j-1].}$$

This is the definition of the Knuth-Morris-Pratt failure function $fail[j]$ for all $j > 1$. By convention we set $fail[1] = 0$; this tells the KMP algorithm that if the first pattern character doesn't match, it should just give up and try the next text character.

$P[i]$	A	B	R	A	C	A	D	A	B	R	A
$fail[i]$	0	1	1	1	2	1	2	1	2	3	4

Failure function for the string ABRACADABRA
(Compare with the finite state machine on the previous page.)

We could easily compute the failure function in $O(m^3)$ time by checking, for each j, whether every prefix of $P[1\,..\,j-1]$ is also a suffix of $P[1\,..\,j-1]$, but this is not the fastest method. The following algorithm essentially uses the KMP search algorithm to look for the pattern inside itself!

```
ComputeFailure(P[1..m]):
  j ← 0
  for i ← 1 to m
      fail[i] ← j        (*)
      while j > 0 and P[i] ≠ P[j]
          j ← fail[j]
      j ← j + 1
```

Here's an example of this algorithm in action. In each line, the current values of i and j are indicated by superscripts; \$ represents the beginning of the string. (You should imagine pointing at $P[j]$ with your left hand and pointing at $P[i]$ with your right hand, and moving your fingers according to the algorithm's directions.)

Just as we did for KnuthMorrisPratt, we can analyze ComputeFailure by amortizing character mismatches against earlier character matches. Since there are at most m character matches, ComputeFailure runs in $O(m)$ time.

Let's prove (by induction, of course) that ComputeFailure correctly computes the failure function. The base case $fail[1] = 0$ is obvious. Assuming inductively that we correctly computed $fail[1]$ through $fail[i-1]$ in line $(*)$, we need to show that $fail[i]$ is also correct. Just after the ith iteration of line $(*)$, we have $j = fail[i]$, so $P[1\,..\,j-1]$ is the longest proper prefix of $P[1\,..\,i-1]$ that is also a suffix.

Let's define the iterated failure functions $fail^c[j]$ inductively as follows: $fail^0[j] = j$, and

$$fail^c[j] = fail[fail^{c-1}[j]] = \overbrace{fail[fail[\cdots[fail}^{c}[j]]\cdots]].$$

$j \leftarrow 0, i \leftarrow 1$	$\$^j$	$\mathbf{A}^i$	B	R	A	C	A	D	A	B	R	X	...
$fail[i] \leftarrow j$		**0**											...
$j \leftarrow j+1, i \leftarrow i+1$	$	$\mathbf{A}^j$	$\mathbf{B}^i$	R	A	C	A	D	A	B	R	X	...
$fail[i] \leftarrow j$		0	**1**										...
$j \leftarrow fail[j]$	$\$^j$	A	$\mathbf{B}^i$	R	A	C	A	D	A	B	R	X	...
$j \leftarrow j+1, i \leftarrow i+1$	$	$\mathbf{A}^j$	B	$\mathbf{R}^i$	A	C	A	D	A	B	R	X	...
$fail[i] \leftarrow j$		0	1	**1**									...
$j \leftarrow fail[j]$	$\$^j$	A	B	$\mathbf{R}^i$	A	C	A	D	A	B	R	X	...
$j \leftarrow j+1, i \leftarrow i+1$	$	$\mathbf{A}^j$	B	R	$\mathbf{A}^i$	C	A	D	A	B	R	X	...
$fail[i] \leftarrow j$		0	1	1	**1**								...
$j \leftarrow j+1, i \leftarrow i+1$	$	A	$\mathbf{B}^j$	R	A	$\mathbf{C}^i$	A	D	A	B	R	X	...
$fail[i] \leftarrow j$		0	1	1	1	**2**							...
$j \leftarrow fail[j]$	$	$\mathbf{A}^j$	B	R	A	$\mathbf{C}^i$	A	D	A	B	R	X	...
$j \leftarrow fail[j]$	$\$^j$	A	B	R	A	$\mathbf{C}^i$	A	D	A	B	R	X	...
$j \leftarrow j+1, i \leftarrow i+1$	$	$\mathbf{A}^j$	B	R	A	C	$\mathbf{A}^i$	D	A	B	R	X	...
$fail[i] \leftarrow j$		0	1	1	1	2	**1**						...
$j \leftarrow j+1, i \leftarrow i+1$	$	A	$\mathbf{B}^j$	R	A	C	A	$\mathbf{D}^i$	A	B	R	X	...
$fail[i] \leftarrow j$		0	1	1	1	2	1	**2**					...
$j \leftarrow fail[j]$	$	$\mathbf{A}^j$	B	R	A	C	A	$\mathbf{D}^i$	A	B	R	X	...
$j \leftarrow fail[j]$	$\$^j$	A	B	R	A	C	A	$\mathbf{D}^i$	A	B	R	X	...
$j \leftarrow j+1, i \leftarrow i+1$	$	$\mathbf{A}^j$	B	R	A	C	A	D	$\mathbf{A}^i$	B	R	X	...
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	**1**				...
$j \leftarrow j+1, i \leftarrow i+1$	$	A	$\mathbf{B}^j$	R	A	C	A	D	A	$\mathbf{B}^i$	R	X	...
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	1	**2**			...
$j \leftarrow j+1, i \leftarrow i+1$	$	A	B	$\mathbf{R}^j$	A	C	A	D	A	B	$\mathbf{R}^i$	X	...
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	1	2	**3**		...
$j \leftarrow j+1, i \leftarrow i+1$	$	A	B	R	$\mathbf{A}^j$	C	A	D	A	B	R	$\mathbf{X}^i$	...
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	1	2	3	**4**	...
$j \leftarrow fail[j]$	$	$\mathbf{A}^j$	B	R	A	C	A	D	A	B	R	$\mathbf{X}^i$	...
$j \leftarrow fail[j]$	$\$^j$	A	B	R	A	C	A	D	A	B	R	$\mathbf{X}^i$	...

ComputeFailure in action. Do this yourself by hand!

In particular, if $fail^{c-1}[j] = 0$, then $fail^c[j]$ is undefined. We can easily show by induction that every string of the form $P[1..fail^c[j]-1]$ is both a proper prefix and a proper suffix of $P[1..i-1]$, and in fact, these are the only examples. Thus, the longest proper prefix/suffix of $P[1..i]$ must be the longest string of the form $P[1..fail^c[j]]$—the one with smallest c—such that $P[fail^c[j]] = P[i]$. This is exactly what the while loop in ComputeFailure computes; the $(c+1)$th iteration compares $P[fail^c[j]] = P[fail^{c+1}[i]]$ against $P[i]$. ComputeFailure is actually a *dynamic programming* implementation of the following recursive definition of $fail[i]$:

$$fail[i] = \begin{cases} 0 & \text{if } i = 0, \\ \max\limits_{c \geq 1} \left\{ fail^c[i-1]+1 \;\middle|\; P[i-1] = P[fail^c[i-1]] \right\} & \text{otherwise.} \end{cases}$$

7.7 Optimizing the Failure Function

We can speed up KnuthMorrisPratt slightly by making one small change to the failure function. Recall that after comparing $T[i]$ against $P[j]$ and finding a mismatch, the algorithm compares $T[i]$ against $P[fail[j]]$. With the current definition, however, it is possible that $P[j]$ and $P[fail[j]]$ are actually the same character, in which case the next character comparison will automatically fail. So why do the comparison at all?

We can optimize the failure function by 'short-circuiting' these redundant comparisons with some simple post-processing:

```
OptimizeFailure(P[1..m], fail[1..m]):
  for i ← 2 to m
      if P[i] = P[fail[i]]
          fail[i] ← fail[fail[i]]
```

We can also compute the optimized failure function directly by adding three new lines (in bold) to the ComputeFailure function.

```
ComputeOptFailure(P[1..m]):
  j ← 0
  for i ← 1 to m
      if P[i] = P[j]
          fail[i] ← fail[j]
      else
          fail[i] ← j
      while j > 0 and P[i] ≠ P[j]
          j ← fail[j]
      j ← j + 1
```

This optimization slows down the preprocessing slightly, but it may significantly decrease the number of comparisons at each text character. The worst-case running time is still $O(n)$; however, the constant is about half as big as for the unoptimized version, so this could be a significant improvement in practice. Several examples of this optimization are given on the next page.

★★★

Feb 2017: Manacher's palindrome-substring algorithm and the Aho-Corasick dictionary-matching algorithm use similar ideas. Knuth-Morris-Pratt was published in 1977, two years after Manacher and Aho-Corasick, but was available as a Stanford tech report in 1974. See also: Gusfield's algorithm Z (which is essentially just KMP preprocessing on $P \bullet T$).

$P[i]$	A	B	R	A	C	A	D	A	B	R	A
unoptimized $fail[i]$	0	1	1	1	2	1	2	1	2	3	4
optimized $fail[i]$	0	1	1	**0**	2	**0**	2	**0**	**1**	**1**	**1**

Optimized finite state machine and failure function for the string 'ABRADACABRA'

$P[i]$	A	N	A	N	A	B	A	N	A	N	A	N	A
unoptimized $fail[i]$	0	1	1	2	3	4	1	2	3	4	5	6	5
optimized $fail[i]$	0	1	0	1	0	4	0	1	0	1	0	6	0

$P[i]$	A	B	A	B	C	A	B	A	B	C	A	B	C
unoptimized $fail[i]$	0	1	1	2	3	1	2	3	4	5	6	7	8
optimized $fail[i]$	0	1	0	1	3	0	1	0	1	3	0	1	8

$P[i]$	A	B	B	A	B	B	A	B	A	B	B	A	B
unoptimized $fail[i]$	0	1	1	1	2	3	4	5	6	2	3	4	5
optimized $fail[i]$	0	1	1	0	1	1	0	1	6	1	1	0	1

Failure functions for four more example strings.

Exercises

1. Describe and analyze a two-dimensional variant of KarpRabin that searches for a given two-dimensional pattern $P[1..p][1..q]$ within a given two-dimensional "text" $T[1..m][1..,n]$. Your algorithm should report *all* index pairs (i, j) such that the subarray $T[i..i+p-1][j..j+q-1]$ is identical to the given pattern, in $O(pq+mn)$ expected time.

2. A *palindrome* is any string that is the same as its reversal, such as `X`, `ABBA`, or `REDIVIDER`. Describe and analyze an algorithm that computes the longest palindrome that is a (not necessarily proper) prefix of a given string $T[1..n]$. Your algorithm should run in $O(n)$ time (either expected or worst-case).

*3. How important is the requirement that the fingerprint modulus q is prime in the original Karp-Rabin algorithm? Specifically, suppose q is chosen uniformly at random in the range $1..N$. If $t_s \neq p$, what is the probability that $\tilde{t}_s = \tilde{p}$? What does this imply about the expected number of false matches? How large should N be to guarantee expected running time $O(m+n)$? *[Hint: This will require some additional number theory.]*

4. Describe a modification of KnuthMorrisPratt in which the pattern can contain any number of *wildcard* symbols `*`, each of which matches an arbitrary string. For example, the pattern `ABR*CAD*BRA` appears in the text `SCHABRAINCADBRANCH`; in this case, the second `*` matches the empty string. Your algorithm should run in $O(m+n)$ time, where m is the length of the pattern and n is the length of the text.

5. Describe a modification of KnuthMorrisPratt in which the pattern can contain any number of *wildcard* symbols `?`, each of which matches an arbitrary single character. For example, the pattern `ABR?CAD?BRA` appears in the text `SCHABRUCADIBRANCH`. Your algorithm should run in $O(m+qn)$ time, where m is the length of the pattern, n is the length of the text., and q is the number of `?`s in the pattern.

*6. Describe another algorithm for the previous problem that runs in time $O(m+kn)$, where k is the number of runs of consecutive non-wildcard characters in the pattern. For example, the pattern `?FISH???B??IS????CUIT?` has $k = 4$ runs.

7. Describe a modification of KnuthMorrisPratt in which the pattern can contain any number of *wildcard* symbols `=`, each of which matches *the same* arbitrary single character. For example, the pattern `=HOC=SPOC=S` appears in the texts `WHUHOCUSPOCUSOT` and `ABRAHOCASPOCASCADABRA`, but *not* in the text `FRISHOCUSPOCESTIX`. Your algorithm should run in $O(m+n)$ time, where m is the length of the pattern and n is the length of the text.

8. This problem considers the maximum length of a *failure chain* $j \to fail[j] \to fail[fail[j]] \to fail[fail[fail[j]]] \to \cdots \to 0$, or equivalently, the maximum number of iterations of the inner loop of KnuthMorrisPratt. This clearly depends on which failure function we use: unoptimized or optimized. Let m be an arbitrary positive integer.

(a) Describe a pattern $A[1..m]$ whose longest *unoptimized* failure chain has length m.

(b) Describe a pattern $B[1..m]$ whose longest *optimized* failure chain has length $\Theta(\log m)$.

*(c) Describe a pattern $C[1..m]$ *containing only two different characters*, whose longest optimized failure chain has length $\Theta(\log m)$.

*(d) Prove that for any pattern of length m, the longest optimized failure chain has length at most $O(\log m)$.

9. Suppose we want to search for a string inside a labeled rooted tree. Our input consists of a *pattern string* $P[1..m]$ and a rooted *text tree* T with n nodes, each labeled with a single character. Nodes in T can have any number of children. Our goal is to either return a downward path in T whose labels match the string P, or report that there is no such path.

The string SEARCH appears on a downward path in the tree.

(a) Describe and analyze a variant of KarpRabin that solves this problem in $O(m+n)$ expected time.

(b) Describe and analyze a variant of KnuthMorrisPratt that solves this problem in $O(m+n)$ expected time.

10. Suppose we want to search a rooted binary tree for subtrees of a certain shape. The input consists of a *pattern tree* P with m nodes and a *text tree* T with n nodes. Every node in both trees has a left subtree and a right subtree, either or both of which may be empty. We want to report *all* nodes v in T such that the subtree rooted at v is structurally identical to P, ignoring all search keys, labels, or other data in the nodes—only the left/right pointer structure matters.

The pattern tree (left) appears exactly twice in the text tree (right).

(a) Describe and analyze a variant of KarpRabin that solves this problem in $O(m+n)$ expected time.

(b) Describe and analyze a variant of KnuthMorrisPratt that solves this problem in $O(m+n)$ expected time.

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://www.cs.uiuc.edu/~jeffe/teaching/algorithms for the most recent revision.

Jaques: *But, for the seventh cause; how did you find the quarrel on the seventh cause?*

Touchstone: *Upon a lie seven times removed:–bear your body more seeming, Audrey:–as thus, sir. I did dislike the cut of a certain courtier's beard: he sent me word, if I said his beard was not cut well, he was in the mind it was: this is called the Retort Courteous. If I sent him word again 'it was not well cut,' he would send me word, he cut it to please himself: this is called the Quip Modest. If again 'it was not well cut,' he disabled my judgment: this is called the Reply Churlish. If again 'it was not well cut,' he would answer, I spake not true: this is called the Reproof Valiant. If again 'it was not well cut,' he would say I lied: this is called the Counter-cheque Quarrelsome: and so to the Lie Circumstantial and the Lie Direct.*

Jaques: *And how oft did you say his beard was not well cut?*

Touchstone: *I durst go no further than the Lie Circumstantial, nor he durst not give me the Lie Direct; and so we measured swords and parted.*

— William Shakespeare, *As You Like It*, Act V, Scene 4 (1600)

13 Randomized Minimum Cut

13.1 Setting Up the Problem

This lecture considers a problem that arises in robust network design. Suppose we have a connected multigraph[1] G representing a communications network like the UIUC telephone system, the Facebook social network, the internet, or Al-Qaeda. In order to disrupt the network, an enemy agent plans to remove some of the edges in this multigraph (by cutting wires, placing police at strategic drop-off points, or paying street urchins to 'lose' messages) to separate it into multiple components. Since his country is currently having an economic crisis, the agent wants to remove as few edges as possible to accomplish this task.

More formally, a *cut* partitions the nodes of G into two nonempty subsets. The *size* of the cut is the number of *crossing edges*, which have one endpoint in each subset. Finally, a *minimum* cut in G is a cut with the smallest number of crossing edges. The same graph may have several minimum cuts.

A multigraph whose minimum cut has three edges.

This problem has a long history. The classical deterministic algorithms for this problem rely on *network flow* techniques, which are discussed in another lecture. The fastest such algorithms (that we will discuss) run in $O(n^3)$ time and are fairly complex; we will see some of these later in the semester. Here I'll describe a relatively simple randomized algorithm discovered by David Karger when he was a Ph.D. student.[2]

[1] A multigraph allows multiple edges between the same pair of nodes. Everything in this lecture could be rephrased in terms of simple graphs where every edge has a non-negative weight, but this would make the algorithms and analysis slightly more complicated.

[2] David R. Karger*. Random sampling in cut, flow, and network design problems. Proc. 25th STOC, 648–657, 1994.

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/ for the most recent revision.

Karger's algorithm uses a primitive operation called ***edge contraction***. Suppose u and v are vertices that are connected by an edge in some multigraph G. To contract the edge $\{u, v\}$, we create a new node called uv, replace any edge of the form $\{u, w\}$ or $\{v, w\}$ with a new edge $\{uv, w\}$, and then delete the original vertices u and v. Equivalently, contracting the edge shrinks the edge down to nothing, pulling the two endpoints together. The new contracted graph is denoted $G/\{u, v\}$. We don't allow self-loops in our multigraphs; if there are multiple edges between u and v, contracting any one of them deletes them all.

A graph G and two contracted graphs $G/\{b, e\}$ and $G/\{c, d\}$.

Any edge in an n-vertex graph can be contracted in $O(n)$ time, assuming the graph is represented as an adjacency list; I'll leave the precise implementation details as an easy exercise.

The correctness of our algorithms will eventually boil down the following simple observation: For any cut in $G/\{u, v\}$, there is cut in G with exactly the same number of crossing edges. In fact, in some sense, the 'same' edges form the cut in both graphs. The converse is not necessarily true, however. For example, in the picture above, the original graph G has a cut of size 1, but the contracted graph $G/\{c, d\}$ does not.

This simple observation has two immediate but important consequences. First, contracting an edge cannot decrease the minimum cut size. More importantly, contracting an edge increases the minimum cut size if and only if that edge is part of *every* minimum cut.

13.2 Blindly Guessing

Let's start with an algorithm that tries to *guess* the minimum cut by randomly contracting edges until only two vertices remain.

```
GuessMinCut(G):
  for i ← n downto 2
      pick a random edge e in G
      G ← G/e
  return the only cut in G
```

Because each contraction requires $O(n)$ time, this algorithm runs in $O(n^2)$ time. Our earlier observations imply that as long as we never contract an edge that lies in every minimum cut, our algorithm will actually guess correctly. But how likely is that?

Suppose G has only one minimum cut—if it actually has more than one, just pick your favorite—and this cut has size k. Every vertex of G must lie on at least k edges; otherwise, we could separate that vertex from the rest of the graph with an even smaller cut. Thus, the number of incident vertex-edge pairs is at least kn. Since every edge is incident to exactly two vertices, G must have at least $kn/2$ edges. That implies that if we pick an edge in G uniformly at random, the probability of picking an edge in the minimum cut is at most $2/n$. In other words, the probability that we don't screw up on the very first step is at least $1 - 2/n$.

Once we've contracted the first random edge, the rest of the algorithm proceeds recursively (with independent random choices) on the remaining $(n-1)$-node graph. So the overall probability $P(n)$ that GuessMinCut returns the true minimum cut is given by the recurrence

$$P(n) \geq \frac{n-2}{n} \cdot P(n-1)$$

with base case $P(2) = 1$. We can expand this recurrence into a product, most of whose factors cancel out immediately.

$$P(n) \geq \prod_{i=3}^{n} \frac{i-2}{i} = \frac{\prod_{i=3}^{n}(i-2)}{\prod_{i=3}^{n} i} = \frac{\prod_{j=1}^{n-2} j}{\prod_{i=3}^{n} i} = \boxed{\frac{2}{n(n-1)}}$$

13.3 Blindly Guessing Over and Over

That's not very good. Fortunately, there's a simple method for increasing our chances of finding the minimum cut: run the guessing algorithm many times and return the smallest guess. Randomized algorithms folks like to call this idea ***amplification***.

```
KargerMinCut(G):
  mink ← ∞
  for i ← 1 to N
      X ← GuessMinCut(G)
      if |X| < mink
          mink ← |X|
          minX ← X
  return minX
```

Both the running time and the probability of success will depend on the number of iterations N, which we haven't specified yet.

First let's figure out the probability that KargerMinCut returns the actual minimum cut. The only way for the algorithm to return the wrong answer is if GuessMinCut fails N times in a row. Since each guess is independent, our probability of success is at least

$$1 - \left(1 - \frac{2}{n(n-1)}\right)^N \leq 1 - e^{-2N/n(n-1)},$$

by The World's Most Useful Inequality $1 + x \leq e^x$. By making N larger, we can make this probability arbitrarily close to 1, but never equal to 1. In particular, if we set $\boldsymbol{N = c\binom{n}{2}\ln n}$ for some constant c, then KargerMinCut is correct with probability at least

$$1 - e^{-c \ln n} = 1 - \frac{1}{n^c}.$$

When the failure probability is a polynomial fraction, we say that the algorithm is correct *with high probability*. Thus, KargerMinCut computes the minimum cut of any n-node graph in $\boldsymbol{O(n^4 \log n)}$ time with high probability.

If we make the number of iterations even larger, say $N = n^2(n-1)/2$, the success probability becomes $1 - e^{-n}$. When the failure probability is exponentially small like this, we say that the algorithm is correct with *very* high probability. In practice, very high probability is usually overkill; high probability is enough. (Remember, there is a small but non-zero probability that your computer will transform itself into a kitten before your program is finished.)

13.4 Not-So-Blindly Guessing

The $O(n^4 \log n)$ running time is actually comparable to some of the simpler flow-based algorithms, but it's nothing to get excited about. But we can improve our guessing algorithm, and thus decrease the number of iterations in the outer loop, by exploiting the observation we made earlier:

> *As the graph shrinks, the probability of contracting an edge in the minimum cut increases.*

At first the probability is quite small, only $2/n$, but near the end of execution, when the graph has only three vertices, we have a $2/3$ chance of screwing up!

A simple technique for working around this increasing probability of error was developed by David Karger and Cliff Stein.[3] Their idea was to group the first several random contractions a "safe" phase, so that the cumulative probability of screwing up is relatively small and a "dangerous" phase, which is much more likely to screw up.

The safe phase shrinks the graph from n nodes to about $n/2$ nodes, using a sequence of about $n/2$ random contractions.[4] Following our earlier analysis, the probability that *none* of these safe contractions touches the minimum cut is at least

$$\prod_{i=n/2+1}^{n} \frac{i-2}{i} = \frac{(n/2)(n/2-1)}{n(n-1)} = \frac{n-2}{4(n-1)} \approx \frac{1}{4}.$$

To get around the danger of the dangerous phase, we use amplification: we run the dangerous phase *four times* and keep the best of the four answers. Naturally, we treat the dangerous phase recursively, so we actually obtain a recursion tree with degree 4, which expands as we get closer to the base case, instead of a single path. More formally, the algorithm looks like this:

```
CONTRACT(G, m):
  for i ← n downto m
      pick a random edge e in G
      G ← G/e
  return G
```

```
BETTERGUESS(G):
  if n < 1000
      use brute force
  else
      H ← CONTRACT(G, n/α)
      X1 ← BETTERGUESS(H, α)
      X2 ← BETTERGUESS(H, α)
      X3 ← BETTERGUESS(H, α)
      X4 ← BETTERGUESS(H, α)
      return min{X1, X2, X3, X4}
```

At first glance, it may look like we are performing exactly the same BETTERGUESS computation four times in a row, but remember that CONTRACT (and therefore BETTERGUESS) is randomized. Each recursive call to BETTERGUESS contracts a different, independently chosen random set of edges. Thus, X_1, X_2, X_3, and X_4 are almost always four completely different cuts!

Why four recursive calls, and not two or six? We're facing a tradeoff between the speed of the algorithm and its probability of success. More recursive calls makes the algorithm more likely to succeed but slower; fewer recursive calls makes the algorithm faster but *considerably* more likely to fail. Four recursive calls turns out to be the right balance, both guaranteeing a fast algorithm and a reasonably large success probability.

[3] David R. Karger* and Cliff Stein. An $\tilde{O}(n^2)$ algorithm for minimum cuts. Proc. 25th STOC, 757–765, 1993.

[4] I'm deliberately simplifying the analysis here to expose more intuition. More formally, if we contract from n to $\lceil n/2 \rceil + 1$ nodes, the probability that no minimum cut edge is contracted is strictly greater than $1/2$.

BetterGuess correctly returns the minimum cut unless (1) none of the edges of the minimum cut are Contracted and (2) *all four* recursive calls return the incorrect cut for H. Let $P(n)$ denote the probability that BetterGuess returns a minimum cut of an n-node graph. Then for each index i, the probability that X_i is the minimum cut of H is $P(n/2)$. Because the four recursive calls are independent, we obtain the following recurrence for $P(n)$, with base case $P(n) = 1$ for all $n \leq 8$.

$$\begin{aligned}
P(n) &= \Pr[\text{mincut in } H \text{ is mincut in } G] \cdot \Pr[\text{some } X_i \text{ is mincut in } H] \\
&\geq \frac{1}{4}\big(1 - \Pr[\text{no } X_i \text{ is mincut in } H]\big) \\
&\geq \frac{1}{4}\big(1 - \Pr[X_1 \text{ is not mincut in } H]^4\big) \\
&\geq \frac{1}{4}\Big(1 - \big(1 - \Pr[X_1 \text{ is mincut in } H]\big)^4\Big) \\
&\geq \frac{1}{4}\left(1 - \left(1 - P\left(\frac{n}{2}\right)\right)^4\right) \\
&= \frac{1}{4} - \frac{1}{4}\left(1 - P\left(\frac{n}{2}\right)\right)^4
\end{aligned}$$

Using a series of transformations, Karger and Stein prove that $P(n) = \Omega(1/\log n)$. I've included a proof at the end of this note.

For the running time, we get a simple recurrence that is easily solved using recursion trees:

$$T(n) = O(n^2) + 4T\left(\frac{n}{2}\right) = O(n^2 \log n)$$

So all this splitting and recursing has slowed down the guessing algorithm slightly, but the probability of failure is *exponentially* smaller!

Let's express the lower bound $P(n) = \Omega(1/\log n)$ explicitly as $P(n) \geq \alpha/\ln n$ for some constant α. (Karger and Stein's proof implies $\alpha > 2$). If we call BetterGuess $N = c\ln^2 n$ times, for some new constant c, the overall probability of success is at least

$$1 - \left(1 - \frac{\alpha}{\ln n}\right)^{c\ln^2 n} \geq 1 - e^{-(c/\alpha)\ln n} = 1 - \frac{1}{n^{c/\alpha}}.$$

By setting c sufficiently large, we can bound the probability of failure by an arbitrarily small polynomial function of n. In other words, we now have an algorithm that computes the minimum cut with high probability in only $\boldsymbol{O(n^2 \log^3 n)}$ time!

*13.5 Solving the Karger-Stein recurrence

Recall the following recursive inequality for the probability that BetterGuess successfully finds a minimum cut of an n-node graph:

$$P(n) \geq \frac{1}{4} - \frac{1}{4}\left(1 - P\left(\frac{n}{2}\right)\right)^4$$

Let $\overline{P}(n)$ be the function that satisfies this recurrence with equality; clearly, $P(n) \geq \overline{P}(n)$.

We can solve this rather ugly recurrence for $\overline{P}(n)$ through a series of functional transformations. First, consider the function $p(k) = \overline{P}(2^k)$; this function satisfies the recurrence

$$p(k) = \frac{1}{4} - \frac{1}{4}(1 - p(k-1))^4,$$

Next, define $d(k) = 1/p(k)$; this new function obeys the reciprocal recurrence

$$d(k) \;=\; \frac{4}{1-\left(1-\frac{1}{d(k-1)}\right)^4} \;=\; \frac{4\,d(k-1)^4}{d(k-1)^4-(d(k-1)-1)^4}$$

Straightforward algebraic manipulation[5] implies the inequalities

$$x < \frac{4x^4}{x^4-(x-1)^4} \le x+3$$

for all $x \ge 1$. Thus, as long as $d(k-1) \ge 1$, we have the simple recursive inequalities

$$d(k-1) < d(k) \le d(k-1)+3.$$

Our original base case $P(2) = 1$ implies the base case $d(1) = 1$. It immediately follows by induction that $d(k) \le 3k-2$, and therefore

$$P(n) \ge \overline{P}(n) = p(\lg n) = \frac{1}{d(\lg n)} \ge \frac{1}{3\lg n - 2} = \Omega\left(\frac{1}{\log n}\right),$$

as promised. Whew!

★★★ Galton-Watson process?

Exercises

1. (a) Suppose you had an algorithm to compute the minimum spanning tree of a graph in $O(m)$ time, where m is the number of edges in the input graph.[6] Use this algorithm as a subroutine to improve the running time of GuessMinCut from $O(n^2)$ to $O(m)$.

   (b) Describe and analyze an algorithm to compute the *maximum-weight edge* in the minimum spanning tree of a graph with m edges in $O(m)$ time. *[Hint: We can compute the median edge weight in $O(m)$ time. How do you tell quickly if that's too big or too small?]*

   (c) Use your algorithm from part (b) to improve the running time of GuessMinCut from $O(n^2)$ to $O(m)$.

2. Suppose you are given a graph G with weighted edges, and your goal is to find a cut whose total weight (not just number of edges) is smallest.

   (a) Describe and analyze an algorithm to select a random edge of G, where the probability of choosing edge e is proportional to the weight of e.

   (b) Prove that if you use the algorithm from part (a), instead of choosing edges uniformly at random, the probability that GuessMinCut returns a minimum-weight cut is still $\Omega(1/n^2)$.

[5] otherwise known as Wolfram Alpha

[6] In fact, there *is* a randomized algorithm—due to Philip Klein, David Karger, and Robert Tarjan—that computes the minimum spanning tree of any graph in $O(m)$ *expected* time. The fastest deterministic algorithm known in 2015 runs in $O(m\alpha(m))$ time, where $\alpha(\cdot)$ is the inverse Ackermann function.

(c) What is the running time of your modified GuessMinCut algorithm?

3. Prove that GuessMinCut returns the *second* smallest cut in its input graph with probability $\Omega(1/n^3)$. (The second smallest cut could be significantly larger than the minimum cut.)

4. Consider the following generalization of the BetterGuess algorithm, where we pass in a real parameter $\alpha > 1$ in addition to the graph G.

```
BetterGuess(G, α):
  if n < 1000
      use brute force
  else
      H ← Contract(G, n/α)
      X1 ← BetterGuess(H, α)
      X2 ← BetterGuess(H, α)
      X3 ← BetterGuess(H, α)
      X4 ← BetterGuess(H, α)
      return min{X1, X2, X3, X4}
```

Assume for this question that the input graph G has a unique minimum cut.

(a) What is the running time of the modified algorithm, as a function of n and α? *[Hint: Consider the cases $\alpha < 2$, $\alpha = 2$, and $\alpha > 2$ separately.]*

(b) What is the probability that Contract$(G, n/\alpha)$ does not contract any edge in the minimum cut in G? Give both an exact expression involving both n and α, and a simple approximation in terms of just α. *[Hint: When $\alpha = 2$, the probability is approximately $1/4$.]*

(c) Estimate the probability that BetterGuess(G, α) returns the minimum cut in G, by adapting the solution to the Karger-Stein recurrence. *[Hint: Consider the cases $\alpha < 2$, $\alpha = 2$, and $\alpha > 2$ separately.]*

(d) Suppose we iterate BetterGuess(G, α) until we are guaranteed to see the minimum cut with high probability. What is the running time of the resulting algorithm? For which value of α is this running time minimized?

(e) Suppose we modify BetterGuess(G, α) further, to make k recursive calls instead of four. What is the best choice of α, as a function of k? What is the resulting running time? What is the best choice for k and α?

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://www.cs.uiuc.edu/~jeffe/teaching/algorithms for the most recent revision.

The goode workes that men don whil they ben in good lif
al amortised by synne folwyng.
— Geoffrey Chaucer, "The Persones [Parson's] Tale" (c.1400)

I will gladly pay you Tuesday for a hamburger today.
— J. Wellington Wimpy, "Thimble Theatre" (1931)

I want my two dollars!
— Johnny Gasparini [Demian Slade], "Better Off Dead" (1985)

A dollar here, a dollar there. Over time, it adds up to two dollars.
— Jarod Kintz, *The Titanic Would Never Have Sunk if It Were Made out of a Sink* (2012)

9 Amortized Analysis

9.1 Incrementing a Binary Counter

It is a straightforward exercise in induction, which often appears on Homework 0, to prove that any non-negative integer n can be represented as the sum of distinct powers of 2. Although some students correctly use induction on the number of bits—pulling off either the least significant bit or the most significant bit in the binary representation and letting the Recursion Fairy convert the remainder—the most commonly submitted proof uses induction on the value of the integer, as follows:

Proof: The base case $n = 0$ is trivial. For any $n > 0$, the inductive hypothesis implies that there is set of distinct powers of 2 whose sum is $n-1$. If we add 2^0 to this set, we obtain a *multiset* of powers of two whose sum is n, which might contain two copies of 2^0. Then as long as there are two copies of any 2^i in the multiset, we remove them both and insert 2^{i+1} in their place. The sum of the elements of the multiset is unchanged by this replacement, because $2^{i+1} = 2^i + 2^i$. Each iteration decreases the size of the multiset by 1, so the replacement process must eventually terminate. When it does terminate, we have a *set* of distinct powers of 2 whose sum is n. □

This proof is describing an algorithm to increment a binary counter from $n-1$ to n. Here's a more formal (and shorter!) description of the algorithm to add 1 to a binary counter. The input B is an (infinite) array of bits, where $B[i] = 1$ if and only if 2^i appears in the sum.

```
INCREMENT(B[0..∞]):
  i ← 0
  while B[i] = 1
      B[i] ← 0
      i ← i + 1
  B[i] ← 1
```

We've already argued that INCREMENT must terminate, but how quickly? Obviously, the running time depends on the array of bits passed as input. If the first k bits are all 1s, then INCREMENT takes $\Theta(k)$ time. The binary representation of any positive integer n is exactly $\lfloor \lg n \rfloor + 1$ bits long. Thus, if B represents an integer between 0 and n, INCREMENT takes $\Theta(\log n)$ time in the worst case.

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

9.2 Counting from 0 to n

Now suppose we call INCREMENT n times, starting with a zero counter. How long does it take to count from 0 to n? If we only use the worst-case running time for each INCREMENT, we get an upper bound of $O(n \log n)$ on the total running time. Although this bound is correct, we can do better; in fact, the total running time is only $\Theta(n)$. This section describes several general methods for deriving, or at least proving, this linear time bound. Many (perhaps even all) of these methods are logically equivalent, but different formulations are more natural for different problems.

9.2.1 Summation

Perhaps the simplest way to derive a tighter bound is to observe that INCREMENT doesn't flip $\Theta(\log n)$ bits *every* time it is called. The least significant bit $B[0]$ does flip in every iteration, but $B[1]$ only flips every other iteration, $B[2]$ flips every 4th iteration, and in general, $B[i]$ flips every 2^ith iteration. Because we start with an array full of 0's, a sequence of n INCREMENTS flips each bit $B[i]$ exactly $\lfloor n/2^i \rfloor$ times. Thus, the total number of bit-flips for the entire sequence is

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < \sum_{i=0}^{\infty} \frac{n}{2^i} = 2n.$$

(More precisely, the number of flips is exactly $2n - \#1(n)$, where $\#1(n)$ is the number of 1 bits in the binary representation of n.) Thus, *on average*, each call to INCREMENT flips just less than two bits, and therefore runs in constant time.

This sense of "on average" is quite different from the averaging we consider with randomized algorithms. There is no probability involved; we are averaging over a sequence of operations, not the possible running times of a single operation. This averaging idea is called ***amortization***—the ***amortized*** time for each INCREMENT is $O(1)$. Amortization is a ~~sleazy~~ clever trick used by accountants to average large one-time costs over long periods of time; the most common example is calculating uniform payments for a loan, even though the borrower is paying interest on less and less capital over time. For this reason, it is common to use "*cost*" as a synonym for running time in the context of amortized analysis. Thus, the worst-case *cost* of INCREMENT is $O(\log n)$, but the amortized *cost* is only $O(1)$.

Most textbooks call this particular technique "the aggregate method", or "aggregate analysis", but these are just fancy names for computing the total cost of all operations and then dividing by the number of operations.

> **The Summation Method.** *Let $T(n)$ be the worst-case running time for a sequence of n operations. The amortized time for each operation is $T(n)/n$.*

9.2.2 Taxation

A second method we can use to derive amortized bounds is called either the *accounting* method or the *taxation* method. Suppose it costs us a dollar to toggle a bit, so we can measure the running time of our algorithm in dollars. Time is money!

Instead of paying for each bit flip when it happens, the Increment Revenue Service charges a two-dollar *increment tax* whenever we want to set a bit from zero to one. One of those dollars is spent changing the bit from zero to one; the other is stored away as *credit* until we need to reset the same bit to zero. The key point here is that we always have enough credit saved up to pay for

the next INCREMENT. The amortized cost of an INCREMENT is the total tax it incurs, which is exactly 2 dollars, since each INCREMENT changes just one bit from 0 to 1.

It is often useful to distribute the tax income to specific pieces of the data structure. For example, for each INCREMENT, we could store one of the two dollars on the single bit that is set for 0 to 1, so that *that* bit can pay to reset itself back to zero later on.

> **Taxation Method 1.** *Certain steps in the algorithm charge you taxes, so that the total cost incurred by the algorithm is never more than the total tax you pay. The amortized cost of an operation is the overall tax charged to you during that operation.*

A different way to schedule the taxes is for *every* bit to charge us a tax at *every* operation, regardless of whether the bit changes of not. Specifically, each bit $B[i]$ charges a tax of $1/2^i$ dollars for each INCREMENT. The total tax we are charged during each INCREMENT is $\sum_{i\geq 0} 2^{-i} = 2$ dollars. Every time a bit $B[i]$ actually needs to be flipped, it has collected exactly \$1, which is just enough for us to pay for the flip.

> **Taxation Method 2.** *Certain portions of the data structure charge you taxes at each operation, so that the total cost of maintaining the data structure is never more than the total taxes you pay. The amortized cost of an operation is the overall tax you pay during that operation.*

In both of the taxation methods, our task as algorithm analysts is to come up with an appropriate 'tax schedule'. Different 'schedules' can result in different amortized time bounds. The tightest bounds are obtained from tax schedules that *just barely* stay in the black.

9.2.3 Charging

Another common method of amortized analysis involves *charging* the cost of some steps to some other, earlier steps. The method is similar to taxation, except that we focus on where each unit of tax is (or will be) *spent*, rather than where is it *collected*. By charging the cost of some operations to earlier operations, we are overestimating the total cost of any sequence of operations, since we pay for some charges from future operations that may never actually occur.

> **The Charging Method.** *Charge the cost of some steps of the algorithm to earlier steps, or to steps in some earlier operation. The amortized cost of the algorithm is its actual running time, minus its total charges* ***to*** *past operations, plus its total charge* ***from*** *future operations.*

For example, in our binary counter, suppose we charge the cost of clearing a bit (changing its value from 1 to 0) to the previous operation that sets that bit (changing its value from 0 to 1). If we flip k bits during an INCREMENT, we charge $k-1$ of those bit-flips to earlier bit-flips. Conversely, the single operation that sets a bit receives at most one unit of charge from the next time that bit is cleared. So instead of paying for k bit-flips, we pay for at most two: one for actually setting a bit, plus at most one charge from the future for clearing that same bit. Thus, the total amortized cost of the INCREMENT is at most two bit-flips.

We can visualize this charging scheme as follows. For each integer i, we represent the running time of the ith INCREMENT as a stack of blocks, one for each bit flip. The jth block in the ith stack is white if the ith INCREMENT changes $B[j]$ from 0 to 1, and shaded if the ith INCREMENT changes $B[j]$ from 1 to 0. If we moved each shaded block onto the white block directly to its left, there would at most two blocks in each stack.

Charging scheme for a binary counter.

9.2.4 Potential

The most powerful method (and the hardest to use) builds on a physics metaphor of 'potential energy'. Instead of associating costs or taxes with particular operations or pieces of the data structure, we represent prepaid work as *potential* that can be spent on later operations. The potential is a function of the entire data structure.

Let D_i denote our data structure after i operations have been performed, and let Φ_i denote its potential. Let c_i denote the actual cost of the ith operation (which changes D_{i-1} into D_i). Then the *amortized* cost of the ith operation, denoted a_i, is defined to be the actual cost plus the increase in potential:

$$\boxed{a_i = c_i + \Phi_i - \Phi_{i-1}}$$

So the *total* amortized cost of n operations is the actual total cost plus the total increase in potential:

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0.$$

A potential function is *valid* if $\Phi_i - \Phi_0 \geq 0$ for all i. If the potential function is valid, then the total *actual* cost of any sequence of operations is always less than the total *amortized* cost:

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} a_i - \Phi_n \leq \sum_{i=1}^{n} a_i.$$

For our binary counter example, we can define the potential Φ_i after the ith INCREMENT to be the number of bits with value 1. Initially, all bits are equal to zero, so $\Phi_0 = 0$, and clearly $\Phi_i > 0$ for all $i > 0$, so this is a valid potential function. We can describe both the actual cost of an INCREMENT and the change in potential in terms of the number of bits set to 1 and reset to 0.

$$c_i = \#\text{bits changed from 0 to 1} + \#\text{bits changed from 1 to 0}$$
$$\Phi_i - \Phi_{i-1} = \#\text{bits changed from 0 to 1} - \#\text{bits changed from 1 to 0}$$

Thus, the amortized cost of the ith INCREMENT is

$$a_i = c_i + \Phi_i - \Phi_{i-1} = 2 \times \#\text{bits changed from 0 to 1}$$

Since INCREMENT changes only *one* bit from 0 to 1, the amortized cost INCREMENT is 2.

> **The Potential Method.** *Define a potential function for the data structure that is initially equal to zero and is always non-negative. The amortized cost of an operation is its actual cost plus the change in potential.*

For this particular example, the potential is precisely the total unspent taxes paid using the taxation method, so it should be no surprise that we obtain precisely the same amortized cost. In general, however, there may be no natural way to interpret change in potential as "taxes" or "charges". Taxation and charging are useful when there is a convenient way to distribute costs to specific steps in the algorithm or components of the data structure. Potential arguments allow us to argue more globally when a local distribution is difficult or impossible.

Different potential functions can lead to different amortized time bounds. The trick to using the potential method is to come up with the best possible potential function. A good potential function goes up a little during any cheap/fast operation, and goes down a lot during any expensive/slow operation. Unfortunately, there is no general technique for finding good potential functions, except to play around with the data structure and try lots of possibilities (most of which won't work).

9.3 Incrementing and Decrementing

Now suppose we wanted a binary counter that we can both increment and decrement efficiently. A standard binary counter won't work, even in an amortized sense; if we alternate between 2^k and $2^k - 1$, *every* operation costs $\Theta(k)$ time.

A nice alternative is represent each integer as a *pair* (P, N) of bit strings, subject to the invariant $\boldsymbol{P \wedge N = 0}$ where $\wedge$ represents bit-wise AND. In other words,

For every index i, at most one of the bits $P[i]$ and $N[i]$ is equal to 1.

If we interpret P and N as binary numbers, the actual value of the counter is $P - N$; thus, intuitively, P represents the "positive" part of the pair, and N represents the "negative" part. Unlike the standard binary representation, this new representation is not unique, except for zero, which can only be represented by the pair $(0, 0)$. In fact, every positive or negative integer can be represented has an *infinite* number of distinct representations.

We can increment and decrement our double binary counter as follows. Intuitively, the INCREMENT algorithm increments P, and the DECREMENT algorithm increments N; however, in both cases, we must change the increment algorithm slightly to maintain the invariant $P \wedge N = 0$.

```
INCREMENT(P, N):
  i ← 0
  while P[i] = 1
      P[i] ← 0
      i ← i + 1
  if N[i] = 1
      N[i] ← 0
  else
      P[i] ← 1
```

```
DECREMENT(P, N):
  i ← 0
  while N[i] = 1
      N[i] ← 0
      i ← i + 1
  if P[i] = 1
      P[i] ← 0
  else
      N[i] ← 1
```

$$\begin{array}{ccccccccccccc}
P = 10001 & & P = 10010 & & P = 10011 & & P = 10000 & & P = 10000 & & P = 10001 \\
N = 01100 & \xrightarrow{++} & N = 01100 & \xrightarrow{++} & N = 01100 & \xrightarrow{++} & N = 01000 & \xrightarrow{--} & N = 01001 & \xrightarrow{--} & N = 01010 & \xrightarrow{++} & N = 01010 \\
P - N = 5 & & P - N = 6 & & P - N = 7 & & P - N = 8 & & P - N = 7 & & P - N = 6 & & P - N = 7
\end{array}$$

Incrementing and decrementing a double-binary counter.

Now suppose we start from $(0, 0)$ and apply a sequence of n INCREMENTS and DECREMENTS. In the worst case, each operation takes $\Theta(\log n)$ time, but what is the amortized cost? We can't

use the aggregate method here, because we don't know what the sequence of operations looks like.

What about taxation? It's not hard to prove (by induction, of course) that after either $P[i]$ or $N[i]$ is set to 1, there must be at least 2^i operations, either INCREMENTS or DECREMENTS, before that bit is reset to 0. So if each bit $P[i]$ and $N[i]$ pays a tax of 2^{-i} at each operation, we will always have enough money to pay for the next operation. Thus, the amortized cost of each operation is at most $\sum_{i\geq 0} 2 \cdot 2^{-i} = 4$.

We can get even better amortized time bounds using the potential method. Define the potential Φ_i to be the number of 1-bits in both P and N after i operations. Just as before, we have

$$\begin{aligned} c_i &= \#\text{bits changed from 0 to 1} + \#\text{bits changed from 1 to 0} \\ \Phi_i - \Phi_{i-1} &= \#\text{bits changed from 0 to 1} - \#\text{bits changed from 1 to 0} \\ \Longrightarrow \qquad a_i &= 2 \times \#\text{bits changed from 0 to 1} \end{aligned}$$

Since each operation changes *at most* one bit to 1, the ith operation has amortized cost $a_i \leq 2$.

*9.4 Gray Codes

An attractive alternate solution to the increment/decrement problem was independently suggested by several students. *Gray codes* (named after Frank Gray, who discovered them in the 1950s) are methods for representing numbers as bit strings so that successive numbers differ by only one bit. For example, here is the four-bit *binary reflected* Gray code for the integers 0 through 15:

0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000

The general rule for incrementing a binary reflected Gray code is to invert the bit that would be set from 0 to 1 by a normal binary counter. In terms of bit-flips, this is the perfect solution; each increment of decrement *by definition* changes only one bit. Unfortunately, the naïve algorithm to *find* the single bit to flip still requires $\Theta(\log n)$ time in the worst case. Thus, so the total cost of maintaining a Gray code, using the obvious algorithm, is the same as that of maintaining a normal binary counter.

Fortunately, this is only true of the naïve algorithm. The following algorithm, discovered by Gideon Ehrlich[1] in 1973, maintains a Gray code counter in constant *worst-case* time per increment! The algorithm uses a separate 'focus' array $F[0..n]$ in addition to a Gray-code bit array $G[0..n-1]$.

```
EhrlichGrayInit(n):
  for i ← 0 to n − 1
      G[i] ← 0
  for i ← 0 to n
      F[i] ← i
```

```
EhrlichGrayIncrement(n):
  j ← F[0]
  F[0] ← 0
  if j = n
      G[n − 1] ← 1 − G[n − 1]
  else
      G[j] = 1 − G[j]
      F[j] ← F[j + 1]
      F[j + 1] ← j + 1
```

[1] Gideon Ehrlich. Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *J. Assoc. Comput. Mach.* 20:500–513, 1973.

The EhrlichGrayIncrement algorithm obviously runs in $O(1)$ time, even in the worst case. Here's the algorithm in action with $n = 4$. The first line is the Gray bit-vector G, and the second line shows the focus vector F, both in reverse order:

$$G : 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000$$
$$F : 3210, 3211, 3220, 3212, 3310, 3311, 3230, 3213, 4210, 4211, 4220, 4212, 3410, 3411, 3240, 3214$$

Voodoo! I won't explain in detail how Ehrlich's algorithm works, except to point out the following invariant. Let $B[i]$ denote the ith bit in the *standard* binary representation of the current number. ***If $B[j] = 0$ and $B[j-1] = 1$, then $F[j]$ is the smallest integer $k > j$ such that $B[k] = 1$; otherwise, $F[j] = j$.*** Got that?

But wait — this algorithm only handles increments; what if we also want to decrement? Sorry, I don't have a clue. Extra credit, anyone?

9.5 Generalities and Warnings

Although computer scientists usually apply amortized analysis to understand the efficiency of maintaining and querying data structures, you should remember that amortization can be applied to *any* sequence of numbers. Banks have been using amortization to calculate fixed payments for interest-bearing loans for centuries. The IRS allows taxpayers to amortize business expenses or gambling losses across several years for purposes of computing income taxes. Some cell phone contracts let you to apply amortization to calling time, by rolling unused minutes from one month into the next month.

It's also important to remember that ***amortized time bounds are not unique***. For a data structure that supports multiple operations, different amortization schemes can assign different costs to *exactly the same* algorithms. For example, consider a generic data structure that can be modified by three algorithms: Fold, Spindle, and Mutilate. One amortization scheme might imply that Fold and Spindle each run in $O(\log n)$ amortized time, while Mutilate runs in $O(n)$ amortized time. Another scheme might imply that Fold runs in $O(\sqrt{n})$ amortized time, while Spindle and Mutilate each run in $O(1)$ amortized time. These two results are not necessarily inconsistent! Moreover, there is no general reason to prefer one of these sets of amortized time bounds over the other; our preference may depend on the context in which the data structure is used.

Exercises

1. Suppose we are maintaining a data structure under a series of n operations. Let $f(k)$ denote the actual running time of the kth operation. For each of the following functions f, determine the resulting amortized cost of a single operation. (For practice, try *all* of the methods described in this note.)

   (a) $f(k)$ is the largest integer i such that 2^i divides k.

   (b) $f(k)$ is the largest power of 2 that divides k.

   (c) $f(k) = n$ if k is a power of 2, and $f(k) = 1$ otherwise.

   (d) $f(k) = n^2$ if k is a power of 2, and $f(k) = 1$ otherwise.

   (e) $f(k)$ is the index of the largest Fibonacci number that divides k.

   (f) $f(k)$ is the largest Fibonacci number that divides k.

(g) $f(k) = k$ if k is a Fibonacci number, and $f(k) = 1$ otherwise.

(h) $f(k) = k^2$ if k is a Fibonacci number, and $f(k) = 1$ otherwise.

(i) $f(k)$ is the largest integer whose square divides k.

(j) $f(k)$ is the largest perfect square that divides k.

(k) $f(k) = k$ if k is a perfect square, and $f(k) = 1$ otherwise.

(l) $f(k) = k^2$ if k is a perfect square, and $f(k) = 1$ otherwise.

(m) Let T be a *complete* binary search tree, storing the integer keys 1 through n. $f(k)$ is the number of ancestors of node k.

(n) Let T be a *complete* binary search tree, storing the integer keys 1 through n. $f(k)$ is the number of descendants of node k.

(o) Let T be a *complete* binary search tree, storing the integer keys 1 through n. $f(k)$ is the *square* of the number of ancestors of node k.

(p) Let T be a *complete* binary search tree, storing the integer keys 1 through n. $f(k) = \mathit{size}(k) \lg \mathit{size}(k)$, where $\mathit{size}(k)$ is the number of descendants of node k.

(q) Let T be an *arbitrary* binary search tree, storing the integer keys 0 through n. $f(k)$ is the length of the path in T from node $k-1$ to node k.

(r) Let T be an *arbitrary* binary search tree, storing the integer keys 0 through n. $f(k)$ is the *square* of the length of the path in T from node $k-1$ to node k.

(s) Let T be a *complete* binary search tree, storing the integer keys 0 through n. $f(k)$ is the *square* of the length of the path in T from node $k-1$ to node k.

2. Consider the following modification of the standard algorithm for incrementing a binary counter.

```
INCREMENT(B[0..∞]):
  i ← 0
  while B[i] = 1
      B[i] ← 0
      i ← i + 1
  B[i] ← 1
  SOMETHINGELSE(i)
```

The only difference from the standard algorithm is the function call at the end, to a black-box subroutine called SomethingElse.

Suppose we call Increment n times, starting with a counter with value 0. The amortized time of each Increment clearly depends on the running time of SomethingElse. Let $T(i)$ denote the worst-case running time of SomethingElse(i). For example, we proved in class that Increment algorithm runs in $O(1)$ amortized time when $T(i) = 0$.

(a) What is the amortized time per Increment if $T(i) = 42$?

(b) What is the amortized time per Increment if $T(i) = 2^i$?

(c) What is the amortized time per Increment if $T(i) = 4^i$?

(d) What is the amortized time per Increment if $T(i) = \sqrt{2}^i$?

(e) What is the amortized time per Increment if $T(i) = 2^i/(i+1)$?

3. An ***extendable array*** is a data structure that stores a sequence of items and supports the following operations.

   - AddToFront(x) adds x to the *beginning* of the sequence.
   - AddToEnd(x) adds x to the *end* of the sequence.
   - Lookup(k) returns the kth item in the sequence, or Null if the current length of the sequence is less than k.

   Describe a ***simple*** data structure that implements an extendable array. Your AddToFront and AddToBack algorithms should take $O(1)$ *amortized* time, and your Lookup algorithm should take $O(1)$ *worst-case* time. The data structure should use $O(n)$ space, where n is the ***current*** length of the sequence.

4. An ***ordered stack*** is a data structure that stores a sequence of items and supports the following operations.

   - OrderedPush(x) removes all items smaller than x from the beginning of the sequence and then adds x to the beginning of the sequence.
   - Pop deletes and returns the first item in the sequence (or Null if the sequence is empty).

   Suppose we implement an ordered stack with a simple linked list, using the obvious OrderedPush and Pop algorithms. Prove that if we start with an empty data structure, the amortized cost of each OrderedPush or Pop operation is $O(1)$.

5. A *multistack* consists of an infinite series of stacks $S_0, S_1, S_2, \ldots$, where the ith stack S_i can hold up to 3^i elements. The user always pushes and pops elements from the smallest stack S_0. However, before any element can be pushed onto any full stack S_i, we first pop all the elements off S_i and push them onto stack S_{i+1} to make room. (Thus, if S_{i+1} is already full, we first recursively move all its members to S_{i+2}.) Similarly, before any element can be popped from any empty stack S_i, we first pop 3^i elements from S_{i+1} and push them onto S_i to make room. (Thus, if S_{i+1} is already empty, we first recursively fill it by popping elements from S_{i+2}.) Moving a single element from one stack to another takes $O(1)$ time.

   Here is pseudocode for the multistack operations MSPush and MSPop. The internal stacks are managed with the subroutines Push and Pop.

```
MPush(x):
  i ← 0
  while S_i is full
      i ← i + 1
  while i > 0
      i ← i − 1
      for j ← 1 to 3^i
          Push(S_{i+1}, Pop(S_i))
  Push(S_0, x)
```

```
MPop(x):
  i ← 0
  while S_i is empty
      i ← i + 1
  while i > 0
      i ← i − 1
      for j ← 1 to 3^i
          Push(S_i, Pop(S_{i+1}))
  return Pop(S_0)
```

   (a) In the worst case, how long does it take to push one more element onto a multistack containing n elements?

Making room in a multistack, just before pushing on a new element.

(b) Prove that if the user never pops anything from the multistack, the amortized cost of a push operation is $O(\log n)$, where n is the maximum number of elements in the multistack during its lifetime.

(c) Prove that in any intermixed sequence of pushes and pops, each push or pop operation takes $O(\log n)$ amortized time, where n is the maximum number of elements in the multistack during its lifetime.

6. Recall that a standard (FIFO) queue maintains a sequence of items subject to the following operations.

- PUSH(x): Add item x to the end of the sequence.
- PULL(): Remove and return the item at the beginning of the sequence.

It is easy to implement a queue using a doubly-linked list and a counter, so that the entire data structure uses $O(n)$ space (where n is the number of items in the queue) and the worst-case time per operation is $O(1)$.

(a) Now suppose we want to support the following operation instead of PULL:

- MULTIPULL(k): Remove the first k items from the front of the queue, and return the kth item removed.

Suppose we use the obvious algorithm to implement MULTIPULL:

```
MultiPull(k):
  for i ← 1 to k
      x ← Pull()
  return x
```

Prove that in any intermixed sequence of PUSH and MULTIPULL operations, the amortized cost of each operation is $O(1)$

(b) Now suppose we *also* want to support the following operation instead of PUSH:

- MULTIPUSH(x, k): Insert k copies of x into the back of the queue.

Suppose we use the obvious algorithm to implement MULTIPUUSH:

```
MultiPush(k, x):
  for i ← 1 to k
      Push(x)
```

Prove that for any integers ℓ and n, there is a sequence of ℓ MultiPush and MultiPull operations that require $\Omega(n\ell)$ time, where n is the maximum number of items in the queue at any time. Such a sequence implies that the amortized cost of each operation is $\Omega(n)$.

(c) Describe a data structure that supports arbitrary intermixed sequences of MultiPush and MultiPull operations in $O(1)$ amortized cost per operation. Like a standard queue, your data structure should use only $O(1)$ space per item.

7. Recall that a standard (FIFO) queue maintains a sequence of items subject to the following operations.

   - Push(x): Add item x to the end of the sequence.
   - Pull(): Remove and return the item at the beginning of the sequence.
   - Size(): Return the current number of items in the sequence.

   It is easy to implement a queue using a doubly-linked list, so that it uses $O(n)$ space (where n is the number of items in the queue) and the worst-case time for each of these operations is $O(1)$.

   Consider the following new operation, which removes every tenth element from the queue, starting at the beginning, in $\Theta(n)$ worst-case time.

```
Decimate():
  n ← Size()
  for i ← 0 to n − 1
      if i mod 10 = 0
          Pull()    ⟨⟨result discarded⟩⟩
      else
          Push(Pull())
```

   Prove that in any intermixed sequence of Push, Pull, and Decimate operations, the amortized cost of each operation is $O(1)$.

8. Chicago has many tall buildings, but only some of them have a clear view of Lake Michigan. Suppose we are given an array $A[1..n]$ that stores the height of n buildings on a city block, indexed from west to east. Building i has a good view of Lake Michigan if and only if every building to the east of i is shorter than i.

   Here is an algorithm that computes which buildings have a good view of Lake Michigan. What is the running time of this algorithm?

```
GoodView(A[1..n]):
  initialize a stack S
  for i ← 1 to n
      while (S not empty and A[i] > A[Top(S)])
          Pop(S)
      Push(S, i)
  return S
```

9. Suppose we can insert or delete an element into a hash table in $O(1)$ time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

   - After an insertion, if the table is more than 3/4 full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
   - After a deletion, if the table is less than 1/4 full, we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

   Show that for any sequence of insertions and deletions, the amortized time per operation is still $O(1)$. *[Hint: Do not use potential functions.]*

10. Professor Pisano insists that the size of any hash table used in his class must always be a Fibonacci number. He insists on the following variant of the previous global rebuilding strategy. Suppose the current hash table has size F_k.

    - After an insertion, if the number of items in the table is F_{k-1}, we allocate a new hash table of size F_{k+1}, insert everything into the new table, and then free the old table.
    - After a deletion, if the number of items in the table is F_{k-3}, we allocate a new hash table of size F_{k-1}, insert everything into the new table, and then free the old table.

    Show that for any sequence of insertions and deletions, the amortized time per operation is still $O(1)$. *[Hint: Do not use potential functions.]*

11. Remember the difference between stacks and queues? Good.

    (a) Describe how to implement a queue using two stacks and $O(1)$ additional memory, so that the amortized time for any enqueue or dequeue operation is $O(1)$. The *only* access you have to the stacks is through the standard subroutines PUSH and POP.

    (b) A *quack* is a data structure combining properties of both stacks and queues. It can be viewed as a list of elements written left to right such that three operations are possible:

    - QUACKPUSH(x): add a new item x to the left end of the list;
    - QUACKPOP(): remove and return the item on the left end of the list;
    - QUACKPULL(): remove the item on the right end of the list.

    Implement a quack using *three* stacks and $O(1)$ additional memory, so that the amortized time for any QUACKPUSH, QUACKPOP, or QUACKPULL operation is $O(1)$. In particular, each element in the quack must be stored in *exactly one* of the three stacks. Again, you ***cannot*** access the component stacks except through the interface functions PUSH and POP.

12. Let's glom a whole bunch of earlier problems together. Yay! An ***random-access double-ended multi-queue*** or ***radmuque*** (pronounced "rad muck") stores a sequence of items and supports the following operations.

    - MULTIPUSH(x, k) adds k copies of item x to the *beginning* of the sequence.

- MultiPoke(x, k) adds k copies of item x to the *end* of the sequence.
- MultiPop(k) removes k items from the *beginning* of the sequence and retrns the last item removed. (If there are less than k items in the sequence, remove them all and return Null.)
- MultiPull(k) removes k items from the *end* of the sequence and retrns the last item removed. (If there are less than k items in the sequence, remove them all and return Null.)
- Lookup(k) returns the kth item in the sequence. (If there are less than k items in the sequence, return Null.)

Describe and analyze a simple data structure that supports these operations using $O(n)$ space, where n is the current number of items in the sequence. Lookup should run in $O(1)$ *worst-case* time; all other operations should run in $O(1)$ *amortized* time.

13. Suppose you are faced with an infinite number of counters x_i, one for each integer i. Each counter stores an integer mod m, where m is a fixed global constant. All counters are initially zero. The following operation increments a single counter x_i; however, if x_i overflows (that is, wraps around from m to 0), the adjacent counters x_{i-1} and x_{i+1} are incremented recursively.

```
Nudgeₘ(i):
  xᵢ ← xᵢ + 1
  while xᵢ ≥ m
      xᵢ ← xᵢ − m
      Nudgeₘ(i − 1)
      Nudgeₘ(i + 1)
```

(a) Prove that Nudge_3 runs in $O(1)$ amortized time. *[Hint: Prove that Nudge_3 always halts!]*

(b) What is the worst-case total time for n calls to Nudge_2, if all counters are initially zero?

14. Now suppose you are faced with an infinite two-dimensional grid of modular counters, one counter $x_{i,j}$ for every pair of integers (i, j). Again, all counters are initially zero. The counters are modified by the following operation, where m is a fixed global constant:

```
2dNudgeₘ(i, j):
  xᵢ,ⱼ ← xᵢ + 1
  while xᵢ,ⱼ ≥ m
      xᵢ,ⱼ ← xᵢ,ⱼ − m
      2dNudgeₘ(i − 1, j)
      2dNudgeₘ(i, j + 1)
      2dNudgeₘ(i + 1, j)
      2dNudgeₘ(i, j − 1)
```

(a) Prove that 2dNudge_5 runs in $O(1)$ amortized time.

★(b) Prove or disprove: 2dNudge_4 also runs in $O(1)$ amortized time.

★(c) Prove or disprove: $\textsc{2dNudge}_3$ always halts.

*15. Suppose instead of powers of two, we represent integers as the sum of Fibonacci numbers. In other words, instead of an array of bits, we keep an array of *fits*, where the ith least significant fit indicates whether the sum includes the ith Fibonacci number F_i. For example, the fitstring 101110_F represents the number $F_6+F_4+F_3+F_2 = 8+3+2+1 = 14$. Describe algorithms to increment and decrement a single fitstring in constant amortized time. *[Hint: Most numbers can be represented by more than one fitstring!]*

*16. A *doubly lazy binary counter* represents any number as a weighted sum of powers of two, where each weight is one of *four* values: -1, 0, 1, or 2. (For succinctness, I'll write $\bar{1}$ instead of -1.) Every integer—positive, negative, or zero—has an infinite number of doubly lazy binary representations. For example, the number 13 can be represented as 1101 (the standard binary representation), or $2\bar{1}01$ (because $2\cdot 2^3 - 2^2 + 2^0 = 13$) or $10\bar{1}1\bar{1}$ (because $2^4 - 2^2 + 2^1 - 2^0 = 13$) or $\bar{1}1200010\bar{1}1\bar{1}$ (because $-2^{10} + 2^9 + 2\cdot 2^8 + 2^4 - 2^2 + 2^1 - 2^0 = 13$).

To increment a doubly lazy binary counter, we add 1 to the least significant bit, then carry the rightmost 2 (if any). To decrement, we subtract 1 from the lest significant bit, and then borrow the rightmost $\bar{1}$ (if any).

```
LazyIncrement(B[0..n]):
  B[0] ← B[0] + 1
  for i ← 1 to n − 1
      if B[i] = 2
          B[i] ← 0
          B[i + 1] ← B[i + 1] + 1
          return
```

```
LazyDecrement(B[0..n]):
  B[0] ← B[0] − 1
  for i ← 1 to n − 1
      if B[i] = −1
          B[i] ← 1
          B[i + 1] ← B[i + 1] − 1
          return
```

For example, here is a doubly lazy binary count from zero up to twenty and then back down to zero. The bits are written with the least significant bit $B[0]$ on the *right*, omitting all leading 0's.

$$0 \xrightarrow{++} 1 \xrightarrow{++} 10 \xrightarrow{++} 11 \xrightarrow{++} 20 \xrightarrow{++} 101 \xrightarrow{++} 110 \xrightarrow{++} 111 \xrightarrow{++} 120 \xrightarrow{++} 201 \xrightarrow{++} 210$$
$$\xrightarrow{++} 1011 \xrightarrow{++} 1020 \xrightarrow{++} 1101 \xrightarrow{++} 1110 \xrightarrow{++} 1111 \xrightarrow{++} 1120 \xrightarrow{++} 1201 \xrightarrow{++} 1210 \xrightarrow{++} 2011 \xrightarrow{++} 2020$$
$$\xrightarrow{--} 2011 \xrightarrow{--} 2010 \xrightarrow{--} 2001 \xrightarrow{--} 2000 \xrightarrow{--} 20\bar{1}1 \xrightarrow{--} 2\bar{1}10 \xrightarrow{--} 2\bar{1}01 \xrightarrow{--} 1100 \xrightarrow{--} 11\bar{1}1 \xrightarrow{--} 1010$$
$$\xrightarrow{--} 1001 \xrightarrow{--} 1000 \xrightarrow{--} 10\bar{1}1 \xrightarrow{--} 1\bar{1}10 \xrightarrow{--} 1\bar{1}01 \xrightarrow{--} 100 \xrightarrow{--} 1\bar{1}1 \xrightarrow{--} 10 \xrightarrow{--} 1 \xrightarrow{--} 0$$

Prove that for any intermixed sequence of increments and decrements of a doubly lazy binary number, starting with 0, the amortized time for each operation is $O(1)$. Do *not* assume, as in the example above, that all the increments come before all the decrements.

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms for the most recent revision.

Everything was balanced before the computers went off line. Try and adjust something, and you unbalance something else. Try and adjust that, you unbalance two more and before you know what's happened, the ship is out of control.

— Blake, *Blake's 7*, "Breakdown" (March 6, 1978)

A good scapegoat is nearly as welcome as a solution to the problem.

— Anonymous

Let's play.

— El Mariachi [Antonio Banderas], *Desperado* (1992)

```
CAPTAIN: TAKE OFF EVERY 'ZIG'!!
CAPTAIN: YOU KNOW WHAT YOU DO-
ING.
CAPTAIN: MOVE 'ZIG'.
CAPTAIN: FOR GREAT JUSTICE.
```

— *Zero Wing* (1992)

10 Scapegoat and Splay Trees

10.1 Definitions

★★★

Move intro paragraphs to earlier treap notes, or maybe to new appendix on basic data structures (arrays, stacks, queues, heaps, binary search trees).

I'll assume that everyone is already familiar with the standard terminology for binary search trees—node, search key, edge, root, internal node, leaf, right child, left child, parent, descendant, sibling, ancestor, subtree, preorder, postorder, inorder, etc.—as well as the standard algorithms for searching for a node, inserting a node, or deleting a node. Otherwise, consult your favorite data structures textbook.

For this lecture, we will consider only *full* binary trees—where every internal node has *exactly* two children—where only the *internal* nodes actually store search keys. In practice, we can represent the leaves with null pointers.

Recall that the *depth* of a node is its distance from the root, and its *height* is the distance to the farthest leaf in its subtree. The height (or depth) of the tree is just the height of the root. The *size* of a node is the number of nodes in its subtree. The size n of the whole tree is just the total number of nodes.

A tree with height h has at most 2^h leaves, so the minimum height of an n-leaf binary tree is $\lceil \lg n \rceil$. In the worst case, the time required for a search, insertion, or deletion to the height of the tree, so in general we would like keep the height as close to $\lg n$ as possible. The best we can possibly do is to have a *perfectly balanced* tree, in which each subtree has as close to half the leaves as possible, and both subtrees are perfectly balanced. The height of a perfectly balanced tree is $\lceil \lg n \rceil$, so the worst-case search time is $O(\log n)$. However, even if we started with a perfectly balanced tree, a malicious sequence of insertions and/or deletions could make the tree arbitrarily unbalanced, driving the search time up to $\Theta(n)$.

To avoid this problem, we need to periodically modify the tree to maintain 'balance'. There are several methods for doing this, and depending on the method we use, the search tree is given a different name. Examples include AVL trees, red-black trees, height-balanced trees, weight-balanced trees, bounded-balance trees, path-balanced trees, B-trees, treaps, randomized

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

binary search trees, skip lists,[1] and jumplists. Some of these trees support searches, insertions, and deletions, in $O(\log n)$ *worst-case* time, others in $O(\log n)$ *amortized* time, still others in $O(\log n)$ *expected* time.

In this lecture, I'll discuss three binary search tree data structures with good *amortized* performance. The first two are variants of *lazy* balanced trees: *lazy weight-balanced trees*, developed by Mark Overmars* in the early 1980s, [14] and *scapegoat trees*, discovered by Arne Andersson* in 1989 [1, 2] and independently[2] by Igal Galperin* and Ron Rivest in 1993 [11]. The third structure is the *splay tree*, discovered by Danny Sleator and Bob Tarjan in 1981 [19, 16].

10.2 Lazy Deletions: Global Rebuilding

First let's consider the simple case where we start with a perfectly-balanced tree, and we only want to perform searches and deletions. To get good search and delete times, we can use a technique called *global rebuilding*. When we get a delete request, we locate and mark the node to be deleted, *but we don't actually delete it*. This requires a simple modification to our search algorithm—we still use marked nodes to guide searches, but if we search for a marked node, the search routine says it isn't there. This keeps the tree more or less balanced, but now the search time is no longer a function of the amount of data currently stored in the tree. To remedy this, we also keep track of how many nodes have been marked, and then apply the following rule:

> **Global Rebuilding Rule.** *As soon as half the nodes in the tree have been marked, rebuild a new perfectly balanced tree containing only the unmarked nodes.*[3]

With this rule in place, a search takes $O(\log n)$ time in the worst case, where n is the number of unmarked nodes. Specifically, since the tree has at most n marked nodes, or $2n$ nodes altogether, we need to examine at most $\lg n + 1$ keys. There are several methods for rebuilding the tree in $O(n)$ time, where n is the size of the new tree. (Homework!) So a single deletion can cost $\Theta(n)$ time in the worst case, but only if we have to rebuild; most deletions take only $O(\log n)$ time.

We spend $O(n)$ time rebuilding, but only after $\Omega(n)$ deletions, so the *amortized* cost of rebuilding the tree is $O(1)$ per deletion. (Here I'm using a simple version of the 'taxation method'. For each deletion, we charge a \$1 tax; after n deletions, we've collected $\$n$, which is just enough to pay for rebalancing the tree containing the remaining n nodes.) Since we also have to find and mark the node being 'deleted', the total amortized time for a deletion is $\boldsymbol{O(\log n)}$.

10.3 Insertions: Partial Rebuilding

Now suppose we only want to support searches and insertions. We can't 'not really insert' new nodes into the tree, since that would make them unavailable to the search algorithm.[4] So instead, we'll use another method called *partial rebuilding*. We will insert new nodes normally, but whenever a *subtree* becomes unbalanced enough, we rebuild it. The definition of 'unbalanced enough' depends on an arbitrary constant $\alpha > 1$.

Each node v will now also store $height(v)$ and $size(v)$. We now modify our insertion algorithm with the following rule:

[1] Yeah, yeah. Skip lists aren't really binary search trees. Whatever you say, Mr. Picky.

[2] The claim of independence is Andersson's [2]. The two papers actually describe very slightly different rebalancing algorithms. The algorithm I'm using here is closer to Andersson's, but my analysis is closer to Galperin and Rivest's.

[3] Alternately: When the number of unmarked nodes is one less than an exact power of two, rebuild the tree. This rule ensures that the tree is always *exactly* balanced.

[4] Well, we could use the Bentley-Saxe* logarithmic method [3], but that would raise the query time to $O(\log^2 n)$.

> **Partial Rebuilding Rule.** *After we insert a node, walk back up the tree updating the heights and sizes of the nodes on the search path. If we encounter a node v where $height(v) > \alpha \cdot \lg(size(v))$, rebuild its subtree into a perfectly balanced tree (in $O(size(v))$ time).*

If we always follow this rule, then after an insertion, the height of the tree is at most $\alpha \cdot \lg n$. Thus, since α is a constant, the worst-case search time is $O(\log n)$. In the worst case, insertions require $\Theta(n)$ time—we might have to rebuild the entire tree. However, the *amortized* time for each insertion is again only $O(\log n)$. Not surprisingly, the proof is a little bit more complicated than for deletions.

Define the *imbalance* $I(v)$ of a node v to be the absolute difference between the sizes of its two subtrees:

$$Imbal(v) := |size(left(v)) - size(right(v))|$$

A simple induction proof implies that $Imbal(v) \leq 1$ for every node v in a perfectly balanced tree. In particular, immediately after we rebuild the subtree of v, we have $Imbal(v) \leq 1$. On the other hand, each insertion into the subtree of v increments either $size(left(v))$ or $size(right(v))$, so $Imbal(v)$ changes by at most 1.

The whole analysis boils down to the following lemma.

Lemma 1. *Just before we rebuild v's subtree, $Imbal(v) = \Omega(size(v))$.*

Before we prove this lemma, let's first look at what it implies. If $Imbal(v) = \Omega(size(v))$, then $\Omega(size(v))$ keys have been inserted in the v's subtree since the last time it was rebuilt from scratch. On the other hand, rebuilding the subtree requires $O(size(v))$ time. Thus, if we amortize the rebuilding cost across all the insertions since the previous rebuild, v is charged *constant* time for each insertion into its subtree. Since each new key is inserted into at most $\alpha \cdot \lg n = O(\log n)$ subtrees, the total amortized cost of an insertion is $\boldsymbol{O(\log n)}$.

Proof: Since we're about to rebuild the subtree at v, we must have $height(v) > \alpha \cdot \lg size(v)$. Without loss of generality, suppose that the node we just inserted went into v's left subtree. Either we just rebuilt this subtree or we didn't have to, so we also have $height(left(v)) \leq \alpha \cdot \lg size(left(v))$. Combining these two inequalities with the recursive definition of height, we get

$$\alpha \cdot \lg size(v) \;<\; height(v) \;\leq\; height(left(v)) + 1 \;\leq\; \alpha \cdot \lg size(left(v)) + 1.$$

After some algebra, this simplifies to $size(left(v)) > size(v)/2^{1/\alpha}$. Combining this with the identity $size(v) = size(left(v)) + size(right(v)) + 1$ and doing some more algebra gives us the inequality

$$size(right(v)) < \left(1 - 1/2^{1/\alpha}\right) size(v) - 1.$$

Finally, we combine these two inequalities using the recursive definition of imbalance.

$$Imbal(v) \geq size(left(v)) - size(right(v)) - 1 \;>\; \left(2/2^{1/\alpha} - 1\right) size(v)$$

Since α is a constant bigger than 1, the factor in parentheses is a positive constant. □

10.4 Scapegoat (Lazy Height-Balanced) Trees

Finally, to handle both insertions and deletions efficiently, *scapegoat trees* use both of the previous techniques. We use partial rebuilding to re-balance the tree after insertions, and global rebuilding to re-balance the tree after deletions. Each search takes $O(\log n)$ time in the worst case, and the amortized time for any insertion or deletion is also $O(\log n)$. There are a few small technical details left (which I won't describe), but no new ideas are required.

Once we've done the analysis, we can actually simplify the data structure. It's not hard to prove that at most one subtree (the *scapegoat*) is rebuilt during any insertion. Less obviously, we can even get the same amortized time bounds (except for a small constant factor) if we only maintain the three integers in addition to the actual tree: the size of the entire tree, the height of the entire tree, and the number of marked nodes. Whenever an insertion causes the tree to become unbalanced, we can compute the sizes of all the subtrees on the search path, starting at the new leaf and stopping at the scapegoat, in time proportional to the size of the scapegoat subtree. Since we need that much time to re-balance the scapegoat subtree, this computation increases the running time by only a small constant factor! Thus, unlike almost every other kind of balanced trees, scapegoat trees require only $O(1)$ extra space.

10.5 Rotations, Double Rotations, and Splaying

Another method for maintaining balance in binary search trees is by adjusting the shape of the tree locally, using an operation called a *rotation*. A rotation at a node x decreases its depth by one and increases its parent's depth by one. Rotations can be performed in constant time, since they only involve simple pointer manipulation.

Figure 1. A right rotation at x and a left rotation at y are inverses.

For technical reasons, we will need to use rotations two at a time. There are two types of double rotations, which might be called *zig-zag* and *roller-coaster*. A zig-zag at x consists of two rotations at x, in opposite directions. A roller-coaster at a node x consists of a rotation at x's parent followed by a rotation at x, both in the same direction. Each double rotation decreases the depth of x by two, leaves the depth of its parent unchanged, and increases the depth of its grandparent by either one or two, depending on the type of double rotation. Either type of double rotation can be performed in constant time.

Finally, a *splay* operation moves an arbitrary node in the tree up to the root through a series of double rotations, possibly with one single rotation at the end. Splaying a node v requires time proportional to $depth(v)$. (Obviously, this means the depth *before* splaying, since after splaying v is the root and thus has depth zero!)

10.6 Splay Trees

A *splay tree* is a binary search tree that is kept more or less balanced by splaying. Intuitively, after we access any node, we move it to the root with a splay operation. In more detail:

Figure 2. A zig-zag at x. The symmetric case is not shown.

Figure 3. A right roller-coaster at x and a left roller-coaster at z.

Figure 4. Splaying a node. Irrelevant subtrees are omitted for clarity.

- **Search:** Find the node containing the key using the usual algorithm, or its predecessor or successor if the key is not present. Splay whichever node was found.
- **Insert:** Insert a new node using the usual algorithm, then splay that node.
- **Delete:** Find the node x to be deleted, splay it, and then delete it. This splits the tree into two subtrees, one with keys less than x, the other with keys bigger than x. Find the node w in the left subtree with the largest key (the inorder predecessor of x in the original tree), splay it, and finally join it to the right subtree.

Figure 5. Deleting a node in a splay tree.

Each search, insertion, or deletion consists of a constant number of operations of the form *walk down to a node, and then splay it up to the root*. Since the walk down is clearly cheaper

than the splay up, all we need to get good amortized bounds for splay trees is to derive good amortized bounds for a single splay.

Believe it or not, the easiest way to do this uses the potential method. We define the *rank* of a node v to be $\lfloor \lg size(v) \rfloor$, and the *potential* of a splay tree to be the sum of the ranks of its nodes:

$$\Phi := \sum_v rank(v) = \sum_v \lfloor \lg size(v) \rfloor$$

It's not hard to observe that a perfectly balanced binary tree has potential $\Theta(n)$, and a linear chain of nodes (a perfectly *unbalanced* tree) has potential $\Theta(n \log n)$.

The amortized analysis of splay trees boils down to the following lemma. Here, $rank(v)$ denotes the rank of v before a (single or double) rotation, and $rank'(v)$ denotes its rank afterwards. Recall that the amortized cost is defined to be the number of rotations plus the drop in potential.

The Access Lemma. *The amortized cost of a single rotation at any node v is at most $1 + 3\,rank'(v) - 3\,rank(v)$, and the amortized cost of a double rotation at any node v is at most $3\,rank'(v) - 3\,rank(v)$.*

Proving this lemma is a straightforward but tedious case analysis of the different types of rotations. For the sake of completeness, I'll give a proof (of a generalized version) in the next section.

By adding up the amortized costs of all the rotations, we find that the total amortized cost of splaying a node v is at most $1 + 3\,rank'(v) - 3\,rank(v)$, where $rank'(v)$ is the rank of v after the entire splay. (The intermediate ranks cancel out in a nice telescoping sum.) But after the splay, v is the root! Thus, $rank'(v) = \lfloor \lg n \rfloor$, which implies that the amortized cost of a splay is at most $3 \lg n - 1 = O(\log n)$.

We conclude that every insertion, deletion, or search in a splay tree takes $O(\log n)$ amortized time.

*10.7 Other Optimality Properties

In fact, splay trees are optimal in several other senses. Some of these optimality properties follow easily from the following generalization of the Access Lemma.

Let's arbitrarily assign each node v a non-negative real *weight* $w(v)$. These weights are not actually stored in the splay tree, nor do they affect the splay algorithm in any way; they are only used to help with the analysis. We then redefine the *size* $s(v)$ of a node v to be the sum of the weights of the descendants of v, including v itself:

$$s(v) := w(v) + s(right(v)) + s(left(v)).$$

If $w(v) = 1$ for every node v, then the size of a node is just the number of nodes in its subtree, as in the previous section. As before, we define the *rank* of any node v to be $r(v) = \lg s(v)$, and the *potential* of a splay tree to be the sum of the ranks of all its nodes:

$$\Phi = \sum_v r(v) = \sum_v \lg s(v)$$

In the following lemma, $r(v)$ denotes the rank of v before a (single or double) rotation, and $r'(v)$ denotes its rank afterwards.

The Generalized Access Lemma. *For **any** assignment of non-negative weights to the nodes, the amortized cost of a single rotation at any node x is at most $1+3r'(x)-3r(x)$, and the amortized cost of a double rotation at any node v is at most $3r'(x)-3r(x)$.*

Proof: First consider a single rotation, as shown in Figure 1.

$$\begin{aligned}
1+\Phi'-\Phi &= 1+r'(x)+r'(y)-r(x)-r(y) && [\text{only } x \text{ and } y \text{ change rank}]\\
&\le 1+r'(x)-r(x) && [r'(y)\le r(y)]\\
&\le 1+3r'(x)-3r(x) && [r'(x)\ge r(x)]
\end{aligned}$$

Now consider a zig-zag, as shown in Figure 2. Only w, x, and z change rank.

$$\begin{aligned}
&2+\Phi'-\Phi\\
&\quad= 2+r'(w)+r'(x)+r'(z)-r(w)-r(x)-r(z) && [\text{only } w,x,z \text{ change rank}]\\
&\quad\le 2+r'(w)+r'(x)+r'(z)-2r(x) && [r(x)\le r(w) \text{ and } r'(x)=r(z)]\\
&\quad= 2+(r'(w)-r'(x))+(r'(z)-r'(x))+2(r'(x)-r(x))\\
&\quad= 2+\lg\frac{s'(w)}{s'(x)}+\lg\frac{s'(z)}{s'(x)}+2(r'(x)-r(x))\\
&\quad\le 2+2\lg\frac{s'(x)/2}{s'(x)}+2(r'(x)-r(x)) && [s'(w)+s'(z)\le s'(x), \lg \text{ is concave}]\\
&\quad= 2(r'(x)-r(x))\\
&\quad\le 3(r'(x)-r(x)) && [r'(x)\ge r(x)]
\end{aligned}$$

Finally, consider a roller-coaster, as shown in Figure 3. Only x, y, and z change rank.

$$\begin{aligned}
&2+\Phi'-\Phi\\
&\quad= 2+r'(x)+r'(y)+r'(z)-r(x)-r(y)-r(z) && [\text{only } x,y,z \text{ change rank}]\\
&\quad\le 2+r'(x)+r'(z)-2r(x) && [r'(y)\le r(z) \text{ and } r(x)\ge r(y)]\\
&\quad= 2+(r(x)-r'(x))+(r'(z)-r'(x))+3(r'(x)-r(x))\\
&\quad= 2+\lg\frac{s(x)}{s'(x)}+\lg\frac{s'(z)}{s'(x)}+3(r'(x)-r(x))\\
&\quad\le 2+2\lg\frac{s'(x)/2}{s'(x)}+3(r'(x)-r(x)) && [s(x)+s'(z)\le s'(x), \lg \text{ is concave}]\\
&\quad= 3(r'(x)-r(x))
\end{aligned}$$

This completes the proof. [5] □

Observe that this argument works for *arbitrary* non-negative vertex weights. By adding up the amortized costs of all the rotations, we find that the total amortized cost of splaying a node x is at most $1+3r(root)-3r(x)$. (The intermediate ranks cancel out in a nice telescoping sum.)

This analysis has several immediate corollaries. The first corollary is that the amortized search time in a splay tree is within a constant factor of the search time in the best possible *static*

[5] This proof is essentially taken verbatim from the original Sleator and Tarjan paper. Another proof technique, which may be more accessible, involves maintaining $\lfloor \lg s(v) \rfloor$ tokens on each node v and arguing about the changes in token distribution caused by each single or double rotation. But I haven't yet internalized this approach enough to include it here.

binary search tree. Thus, if some nodes are accessed more often than others, the standard splay algorithm *automatically* keeps those more frequent nodes closer to the root, at least most of the time.

Static Optimality Theorem. *Suppose each node x is accessed at least $t(x)$ times, and let $T = \sum_x t(x)$. The amortized cost of accessing x is $O(\log T - \log t(x))$.*

Proof: Set $w(x) = t(x)$ for each node x. □

For any nodes x and z, let $dist(x,z)$ denote the *rank distance* between x and y, that is, the number of nodes y such that $key(x) \le key(y) \le key(z)$ or $key(x) \ge key(y) \ge key(z)$. In particular, $dist(x,x) = 1$ for all x.

Static Finger Theorem. *For any fixed node f ('the finger'), the amortized cost of accessing x is $O(\lg dist(f,x))$.*

Proof: Set $w(x) = 1/dist(x,f)^2$ for each node x. Then $s(root) \le \sum_{i=1}^{\infty} 2/i^2 = \pi^2/3 = O(1)$, and $r(x) \ge \lg w(x) = -2 \lg dist(f,x)$. □

Here are a few more interesting properties of splay trees, which I'll state without proof.[6] The proofs of these properties (especially the dynamic finger theorem) are considerably more complicated than the amortized analysis presented above.

Working Set Theorem [16]. *The amortized cost of accessing node x is $O(\log D)$, where D is the number of distinct items accessed since the last time x was accessed. (For the first access to x, we set $D = n$.)*

Scanning Theorem [18]. *Splaying all nodes in a splay tree in order, starting from any initial tree, requires $O(n)$ total rotations.*

Dynamic Finger Theorem [7, 6]. *Immediately after accessing node y, the amortized cost of accessing node x is $O(\lg dist(x,y))$.*

*10.8 Splay Tree Conjectures

Splay trees are conjectured to have many interesting properties in addition to the optimality properties that have been proved; I'll describe just a few of the more important ones.

The *Deque Conjecture* [18] considers the cost of dynamically maintaining two fingers l and r, starting on the left and right ends of the tree. Suppose at each step, we can move one of these two fingers either one step left or one step right; in other words, we are using the splay tree as a doubly-ended queue. Sundar* proved that the total cost of m deque operations on an n-node splay tree is $O((m+n)\alpha(m+n))$ [17]. More recently, Pettie later improved this bound to $O(m\alpha^*(n))$ [15]. The Deque Conjecture states that the total cost is actually $O(m+n)$.

The *Traversal Conjecture* [16] states that accessing the nodes in a splay tree, in the order specified by a *preorder* traversal of any other binary tree with the same keys, takes $O(n)$ time. This is generalization of the Scanning Theorem.

The *Unified Conjecture* [13] states that the time to access node x is $O(\lg \min_y (D(y) + d(x,y)))$, where $D(y)$ is the number of *distinct* nodes accessed since the last time y was accessed. This

[6] This list and the following section are taken almost directly from Erik Demaine's lecture notes [5].

would immediately imply both the Dynamic Finger Theorem, which is about spatial locality, and the Working Set Theorem, which is about temporal locality. Two other structures are known that satisfy the unified bound [4, 13].

Finally, the most important conjecture about splay trees, and one of the most important open problems about data structures, is that they are *dynamically optimal* [16]. Specifically, the cost of any sequence of accesses to a splay tree is conjectured to be at most a constant factor more than the cost of the best possible dynamic binary search tree *that knows the entire access sequence in advance*. To make the rules concrete, we consider binary search trees that can undergo *arbitrary* rotations after a search; the cost of a search is the number of key comparisons plus the number of rotations. We do not require that the rotations be on or even near the search path. This is an extremely strong conjecture!

No dynamically optimal binary search tree is known, even in the offline setting. However, three very similar $O(\log\log n)$-competitive binary search trees have been discovered in the last few years: *Tango trees* [9], *multisplay trees* [20], and *chain-splay trees* [12]. A recently-published geometric formulation of dynamic binary search trees [8, 10] also offers significant hope for future progress.

References

[1] Arne Andersson*. Improving partial rebuilding by using simple balance criteria. *Proc. Workshop on Algorithms and Data Structures*, 393–402, 1989. Lecture Notes Comput. Sci. 382, Springer-Verlag.

[2] Arne Andersson. General balanced trees. *J. Algorithms* 30:1–28, 1999.

[3] Jon L. Bentley and James B. Saxe*. Decomposable searching problems I: Static-to-dynamic transformation. *J. Algorithms* 1(4):301–358, 1980.

[4] Mihai Bădiou* and Erik D. Demaine. A simplified and dynamic unified structure. *Proc. 6th Latin American Sympos. Theoretical Informatics*, 466–473, 2004. Lecture Notes Comput. Sci. 2976, Springer-Verlag.

[5] Jeff Cohen* and Erik Demaine. 6.897: Advanced data structures (Spring 2005), Lecture 3, February 8 2005. ⟨http://theory.csail.mit.edu/classes/6.897/spring05/lec.html⟩.

[6] Richard Cole. On the dynamic finger conjecture for splay trees. Part II: The proof. *SIAM J. Comput.* 30(1):44–85, 2000.

[7] Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. Part I: Splay sorting $\log n$-block sequences. *SIAM J. Comput.* 30(1):1–43, 2000.

[8] Erik D. Demaine, Dion Harmon*, John Iacono, Daniel Kane*, and Mihai Pătraşcu. The geometry of binary search trees. *Proc. 20th Ann. ACM-SIAM Symp. Discrete Algorithms*, 496–505, 2009.

[9] Erik D. Demaine, Dion Harmon*, John Iacono, and Mihai Pătraşcu**. Dynamic optimality—almost. *Proc. 45th Annu. IEEE Sympos. Foundations Comput. Sci.*, 484–490, 2004.

[10] Jonathan Derryberry*, Daniel Dominic Sleator, and Chengwen Chris Wang*. A lower bound framework for binary search trees with rotations. Tech. Rep. CMU-CS-05-187, Carnegie Mellon Univ., Nov. 2005. ⟨http://reports-archive.adm.cs.cmu.edu/anon/2005/CMU-CS-05-187.pdf⟩.

[11] Igal Galperin* and Ronald R. Rivest. Scapegoat trees. *Proc. 4th Annu. ACM-SIAM Sympos. Discrete Algorithms*, 165–174, 1993.

[12] George F. Georgakopoulos. Chain-splay trees, or, how to achieve and prove $\log\log N$-competitiveness by splaying. *Inform. Proc. Lett.* 106(1):37–43, 2008.

[13] John Iacono*. Alternatives to splay trees with $O(\log n)$ worst-case access times. *Proc. 12th Annu. ACM-SIAM Sympos. Discrete Algorithms*, 516–522, 2001.

[14] Mark H. Overmars*. *The Design of Dynamic Data Structures*. Lecture Notes Comput. Sci. 156. Springer-Verlag, 1983.

[15] Seth Pettie. Splay trees, Davenport-Schinzel sequences, and the deque conjecture. *Proc. 19th Ann. ACM-SIAM Symp. Discrete Algorithms*, 1115–1124, 2008.

[16] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *J. ACM* 32(3):652–686, 1985.

[17] Rajamani Sundar*. On the Deque conjecture for the splay algorithm. *Combinatorica* 12(1):95–124, 1992.

[18] Robert E. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica* 5(5):367–378, 1985.

[19] Robert Endre Tarjan. *Data Structures and Network Algorithms*. CBMS-NSF Regional Conference Series in Applied Mathematics 44. SIAM, 1983.

[20] Chengwen Chris Wang*, Jonathan Derryberry*, and Daniel Dominic Sleator. $O(\log\log n)$-competitive dynamic binary search trees. *Proc. 17th Annu. ACM-SIAM Sympos. Discrete Algorithms*, 374–383, 2006.

*Starred authors were graduate students at the time that the cited work was published. **Double-starred authors were undergraduates.

Exercises

1. (a) An n-node binary tree is *perfectly balanced* if either $n \leq 1$, or its two subtrees are perfectly balanced binary trees, each with at most $\lfloor n/2 \rfloor$ nodes. Prove that $I(v) \leq 1$ for every node v of any perfectly balanced tree.

   (b) Prove that at most one subtree is rebalanced during a scapegoat tree insertion.

2. In a *dirty* binary search tree, each node is labeled either *clean* or *dirty*. The lazy deletion scheme used for scapegoat trees requires us to *purge* the search tree, keeping all the clean nodes and deleting all the dirty nodes, as soon as half the nodes become dirty. In addition, the purged tree should be perfectly balanced.

   (a) Describe and analyze an algorithm to purge an arbitrary n-node dirty binary search tree in $O(n)$ time. (Such an algorithm is necessary for scapegoat trees to achieve $O(\log n)$ amortized insertion cost.)

   *(b) Modify your algorithm so that is uses only $O(\log n)$ space, in addition to the tree itself. Don't forget to include the recursion stack in your space bound.

   ★(c) Modify your algorithm so that is uses only $O(1)$ additional space. In particular, your algorithm cannot call itself recursively at all.

3. Consider the following simpler alternative to splaying:

```
MoveToRoot(v):
  while parent(v) ≠ Null
      rotate at v
```

   Prove that the amortized cost of MoveToRoot in an n-node binary tree can be $\Omega(n)$. That is, prove that for any integer k, there is a sequence of k MoveToRoot operations that require $\Omega(kn)$ time to execute.

4. Let P be a set of n points in the plane. The *staircase* of P is the set of all points in the plane that have at least one point in P both above and to the right.

A set of points in the plane and its staircase (shaded).

(a) Describe an algorithm to compute the staircase of a set of n points in $O(n \log n)$ time.

(b) Describe and analyze a data structure that stores the staircase of a set of points, and an algorithm ABOVE?(x, y) that returns TRUE if the point (x, y) is above the staircase, or FALSE otherwise. Your data structure should use $O(n)$ space, and your ABOVE? algorithm should run in $O(\log n)$ time.

Two staircase queries.

(c) Describe and analyze a data structure that maintains a staircase as new points are inserted. Specifically, your data structure should support a function INSERT(x, y) that adds the point (x, y) to the underlying point set and returns TRUE or FALSE to indicate whether the staircase of the set has changed. Your data structure should use $O(n)$ space, and your INSERT algorithm should run in $O(\log n)$ amortized time.

Two staircase insertions.

5. Suppose we want to maintain a dynamic set of values, subject to the following operations:

- INSERT(x): Add x to the set (if it isn't already there).
- PRINT&DELETEBETWEEN(a, b): Print every element x in the range $a \leq x \leq b$, in increasing order, and delete those elements from the set.

For example, if the current set is $\{1, 5, 3, 4, 8\}$, then

- PRINT&DELETEBETWEEN$(4, 6)$ prints the numbers 4 and 5 and changes the set to $\{1, 3, 8\}$;
- PRINT&DELETEBETWEEN$(6, 7)$ prints nothing and does not change the set;
- PRINT&DELETEBETWEEN$(0, 10)$ prints the sequence $1, 3, 4, 5, 8$ and deletes everything.

(a) Suppose we store the set in our favorite balanced binary search tree, using the standard INSERT algorithm and the following algorithm for PRINT&DELETEBETWEEN:

```
PRINT&DELETEBETWEEN(a, b):
  x ← SUCCESSOR(a)
  while x ≤ b
      print x
      DELETE(x)
      x ← SUCCESSOR(a)
```

Here, SUCCESSOR(a) returns the smallest element greater than or equal to a (or ∞ if there is no such element), and DELETE is the standard deletion algorithm. Prove that the amortized time for INSERT and PRINT&DELETEBETWEEN is $O(\log N)$, where N is the *maximum* number of items that are ever stored in the tree.

(b) Describe and analyze INSERT and PRINT&DELETEBETWEEN algorithms that run in $O(\log n)$ amortized time, where n is the *current* number of elements in the set.

(c) What is the running time of your INSERT algorithm in the worst case?

(d) What is the running time of your PRINT&DELETEBETWEEN algorithm in the worst case?

6. Say that a binary search tree is *augmented* if every node v also stores $size(v)$, the number of nodes in the subtree rooted at v.

(a) Show that a rotation in an augmented binary tree can be performed in constant time.

(b) Describe an algorithm SCAPEGOATSELECT(k) that selects the kth smallest item in an augmented scapegoat tree in $O(\log n)$ *worst-case* time. (The scapegoat trees presented in these notes are already augmented.)

(c) Describe an algorithm SPLAYSELECT(k) that selects the kth smallest item in an augmented splay tree in $O(\log n)$ *amortized* time.

(d) Describe an algorithm TREAPSELECT(k) that selects the kth smallest item in an augmented treap in $O(\log n)$ *expected* time.

7. Many applications of binary search trees attach a *secondary data structure* to each node in the tree, to allow for more complicated searches. Let T be an arbitrary binary tree. The secondary data structure at any node v stores exactly the same set of items as the subtree of T rooted at v. This secondary structure has size $O(size(v))$ and can be built in $O(size(v))$ time, where $size(v)$ denotes the number of descendants of v.

The primary and secondary data structures are typically defined by different attributes of the data being stored. For example, to store a set of points in the plane, we could define the primary tree T in terms of the x-coordinates of the points, and define the secondary data structures in terms of their y-coordinate.

Maintaining these secondary structures complicates algorithms for keeping the top-level search tree balanced. Specifically, performing a rotation at any node v in the primary tree now requires $O(size(v))$ time, because we have to rebuild one of the secondary structures (at the new child of v). When we insert a new item into T, we must also insert into one or more secondary data structures.

(a) Overall, how much space does this data structure use *in the worst case*?

(b) How much space does this structure use if the primary search tree is perfectly balanced?

(c) Suppose the primary tree is a splay tree. Prove that the *amortized* cost of a splay (and therefore of a search, insertion, or deletion) is $\Omega(n)$. *[Hint: This is easy!]*

(d) Now suppose the primary tree T is a scapegoat tree. How long does it take to rebuild the subtree of T rooted at some node v, as a function of $size(v)$?

(e) Suppose the primary tree and all secondary trees are scapegoat trees. What is the amortized cost of a single insertion?

*(f) Finally, suppose the primary tree and every secondary tree is a treap. What is the worst-case *expected* time for a single insertion?

8. Suppose we want to maintain a collection of strings (sequences of characters) under the following operations:

   - NewString(a) creates a new string of length 1 containing only the character a and returns a pointer to that string.
   - Concat(S, T) removes the strings S and T (given by pointers) from the data structure, adds the concatenated string ST to the data structure, and returns a pointer to the new string.
   - Split(S, k) removes the strings S (given by a pointer) from the data structure, adds the first k characters of S and the rest of S as two new strings in the data structure, and returns pointers to the two new strings.
   - Reverse(S) removes the string S (given by a pointer) from the data structure, adds the reversal of S to the data structure, and returns a pointer to the new string.
   - Lookup(S, k) returns the kth character in string S (given by a pointer), or Null if the length of the S is less than k.

   Describe and analyze a simple data structure that supports NewString and Reverse in $O(1)$ *worst-case* time, supports every other operation in $O(\log n)$ time (either worst-case, expected, or amortized), and uses $O(n)$ space, where n is the sum of the *current* string lengths. *[Hint: Why is this problem here?]*

9. After the Great Academic Meltdown of 2020, you get a job as a cook's assistant at Jumpin' Jack's Flapjack Stack Shack, which sells arbitrarily-large stacks of pancakes for just four bits (50 cents) each. Jumpin' Jack insists that any stack of pancakes given to one of his customers must be sorted, with smaller pancakes on top of larger pancakes. Also, whenever a pancake goes to a customer, at least the top side must not be burned.

   The cook provides you with a unsorted stack of n perfectly round pancakes, of n different sizes, possibly burned on one or both sides. Your task is to throw out the pancakes that are burned on both sides (and *only* those) and sort the remaining pancakes so that their burned sides (if any) face down. Your only tool is a spatula. You can insert the spatula under any pancake and then either *flip* or *discard* the stack of pancakes above the spatula.

   More concretely, we can represent a stack of pancakes by a sequence of distinct integers between 1 and n, representing the sizes of the pancakes, with each number marked to

Flipping the top four pancakes. Again.

indicate the burned side(s) of the corresponding pancake. For example, $\underline{1}\,\overline{4}\,3\,\underline{\overline{2}}$ represents a stack of four pancakes: a one-inch pancake burned on the bottom; a four-inch pancake burned on the top; an unburned three-inch pancake, and a two-inch pancake burned on both sides. We store this sequence in a data structure that supports the following operations:

- Position(x): Return the position of integer x in the current sequence, or 0 if x is not in the sequence.
- Value(k): Return the kth integer in the current sequence, or 0 if the sequence has no kth element. Value is essentially the inverse of Position.
- TopBurned(k): Return True if and only if the top side of the kth pancake in the current sequence is burned.
- Flip(k): Reverse the order and the burn marks of the first k elements of the sequence.
- Discard(k): Discard the first k elements of the sequence.

(a) Describe an algorithm to filter and sort any stack of n burned pancakes using $O(n)$ of the operations listed above. Try to make the big-Oh constant small.

$$\underline{1}\,\overline{4}\,3\,\underline{\overline{2}} \xrightarrow{\text{Flip}(4)} \underline{\overline{2}}\,3\,\underline{4}\,\overline{1} \xrightarrow{\text{Discard}(1)} 3\,\underline{4}\,\overline{1} \xrightarrow{\text{Flip}(2)} \overline{4}\,3\,\overline{1} \xrightarrow{\text{Flip}(3)} \underline{1}\,3\,\underline{4}$$

(b) Describe a data structure that supports each of the operations listed above in $O(\log n)$ amortized time. Together with part (a), such a data structure gives us an algorithm to filter and sort any stack of n burned pancakes in $O(n \log n)$ time.

10. Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ be a sequence of m integers, each from the set $\{1, 2, \ldots, n\}$. We can visualize this sequence as a set of integer points in the plane, by interpreting each element x_i as the point (x_i, i). The resulting point set, which we can also call X, has exactly one point on each row of the $n \times m$ integer grid.

(a) Let Y be an arbitrary set of integer points in the plane. Two points (x_1, y_1) and (x_2, y_2) in Y are *isolated* if (1) $x_1 \neq x_2$ and $y_1 \neq y_2$, and (2) there is no other point $(x, y) \in Y$ with $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$. If the set Y contains no isolated pairs of points, we call Y a *commune*.[7]

Let X be an arbitrary set of points on the $n \times n$ integer grid with exactly one point per row. Show that there is a commune Y that contains X and consists of $O(n \log n)$ points.

[7] Demaine *et al.* [8] refer to communes as *arborally satisfied sets*.

(b) Consider the following model of self-adjusting binary search trees. We interpret X as a sequence of accesses in a binary search tree. Let T_0 denote the initial tree. In the ith round, we traverse the path from the root to node x_i, and then *arbitrarily reconfigure* some subtree S_i of the current search tree T_{i-1} to obtain the next search tree T_i. The only restriction is that the subtree S_i must contain both x_i and the root of T_{i-1}. (For example, in a splay tree, S_i is the search path to x_i.) The *cost* of the ith access is the number of nodes in the subtree S_i.

Prove that the minimum cost of executing an access sequence X in this model is at least the size of the smallest commune containing the corresponding point set X. *[Hint: Lowest common ancestor.]*

*(c) Suppose X is a *random* permutation of the integers $1, 2, \ldots, n$. Use the lower bound in part (b) to prove that the expected minimum cost of executing X is $\Omega(n \log n)$.

★(d) Describe a polynomial-time algorithm to compute (or even approximate up to constant factors) the smallest commune containing a given set X of integer points, with at most one point per row. Alternately, prove that the problem is NP-hard.

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms for the most recent revision.

E pluribus unum (Out of many, one)

— Official motto of the United States of America

John: *Who's your daddy? C'mon, you know who your daddy is!*
Who's your daddy? D'Argo, tell him who his daddy is!
D'Argo: *I'm your daddy.*

— *Farscape*, "Thanks for Sharing" (June 15, 2001)

What rolls down stairs, alone or in pairs, rolls over your neighbor's dog?
What's great for a snack, and fits on your back? It's Log, Log, Log!

It's Log! It's Log! It's big, it's heavy, it's wood!
It's Log! It's Log! It's better than bad, it's good!

— *Ren & Stimpy*, "Stimpy's Big Day/The Big Shot" (August 11, 1991)
lyrics by John Kricfalusi

The thing's hollow - it goes on forever - and - oh my God! - it's full of stars!

— Capt. David Bowman's last words(?)
2001: A Space Odyssey by Arthur C. Clarke (1968)

11 Data Structures for Disjoint Sets

In this lecture, we describe some methods for maintaining a collection of disjoint sets. Each set is represented as a pointer-based data structure, with one node per element. We will refer to the elements as either 'objects' or 'nodes', depending on whether we want to emphasize the set abstraction or the actual data structure. Each set has a unique 'leader' element, which identifies the set. (Since the sets are always disjoint, the same object cannot be the leader of more than one set.) We want to support the following operations.

- $\textsc{MakeSet}(x)$: Create a new set $\{x\}$ containing the single element x. The object x must not appear in any other set in our collection. The leader of the new set is obviously x.
- $\textsc{Find}(x)$: Find (the leader of) the set containing x.
- $\textsc{Union}(A, B)$: Replace two sets A and B in our collection with their union $A \cup B$. For example, $\textsc{Union}(A, \textsc{MakeSet}(x))$ adds a new element x to an existing set A. The sets A and B are specified by arbitrary elements, so $\textsc{Union}(x, y)$ has exactly the same behavior as $\textsc{Union}(\textsc{Find}(x), \textsc{Find}(y))$.

Disjoint set data structures have lots of applications. For instance, Kruskal's minimum spanning tree algorithm relies on such a data structure to maintain the components of the intermediate spanning forest. Another application is maintaining the connected components of a graph as new vertices and edges are added. In both these applications, we can use a disjoint-set data structure, where we maintain a set for each connected component, containing that component's vertices.

11.1 Reversed Trees

One of the easiest ways to store sets is using trees, in which each node represents a single element of the set. Each node points to another node, called its *parent*, except for the leader of each set, which points to itself and thus is the root of the tree. $\textsc{MakeSet}$ is trivial. $\textsc{Find}$ traverses

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

parent pointers up to the leader. UNION just redirects the parent pointer of one leader to the other. Unlike most tree data structures, nodes do *not* have pointers down to their children.

```
MAKESET(x):
  parent(x) ← x
```

```
FIND(x):
  while x ≠ parent(x)
      x ← parent(x)
  return x
```

```
UNION(x, y):
  x̄ ← FIND(x)
  ȳ ← FIND(y)
  parent(ȳ) ← x̄
```

Merging two sets stored as trees. Arrows point to parents. The shaded node has a new parent.

MAKE-SET clearly takes $\Theta(1)$ time, and UNION requires only $O(1)$ time in addition to the two FINDS. The running time of FIND(x) is proportional to the depth of x in the tree. It is not hard to come up with a sequence of operations that results in a tree that is a long chain of nodes, so that FIND takes $\boldsymbol{\Theta(n)}$ ***time*** in the worst case.

However, there is an easy change we can make to our UNION algorithm, called *union by depth*, so that the trees always have logarithmic depth. Whenever we need to merge two trees, we always make the root of the *shallower* tree a child of the *deeper* one. This requires us to also maintain the depth of each tree, but this is quite easy.

```
MAKESET(x):
  parent(x) ← x
  depth(x) ← 0
```

```
FIND(x):
  while x ≠ parent(x)
      x ← parent(x)
  return x
```

```
UNION(x, y)
  x̄ ← FIND(x)
  ȳ ← FIND(y)
  if depth(x̄) > depth(ȳ)
      parent(ȳ) ← x̄
  else
      parent(x̄) ← ȳ
      if depth(x̄) = depth(ȳ)
          depth(ȳ) ← depth(ȳ) + 1
```

With this new rule in place, it's not hard to prove by induction that for any set leader $\overline{x}$, the size of $\overline{x}$'s set is at least $2^{depth(\overline{x})}$, as follows. If $depth(\overline{x}) = 0$, then $\overline{x}$ is the leader of a singleton set. For any $d > 0$, when $depth(\overline{x})$ becomes d for the first time, $\overline{x}$ is becoming the leader of the union of two sets, both of whose leaders had depth $d-1$. By the inductive hypothesis, both component sets had at least 2^{d-1} elements, so the new set has at least 2^d elements. Later UNION operations might add elements to $\overline{x}$'s set without changing its depth, but that only helps us.

Since there are only n elements altogether, the maximum depth of any set is $\lg n$. We conclude that if we use union by depth, both FIND and UNION run in $\boldsymbol{\Theta(\log n)}$ time in the worst case.

11.2 Shallow Threaded Trees

Alternately, we could just have every object keep a pointer to the leader of its set. Thus, each set is represented by a shallow tree, where the leader is the root and all the other elements are its

children. With this representation, MakeSet and Find are completely trivial. Both operations clearly run in constant time. Union is a little more difficult, but not much. Our algorithm sets all the leader pointers in one set to point to the leader of the other set. To do this, we need a method to visit every element in a set; we will 'thread' a linked list through each set, starting at the set's leader. The two threads are merged in the Union algorithm in constant time.

Merging two sets stored as threaded trees.
Bold arrows point to leaders; lighter arrows form the threads. Shaded nodes have a new leader.

```
MakeSet(x):
  leader(x) ← x
  next(x) ← x
```

```
Find(x):
  return leader(x)
```

```
Union(x, y):
  x̄ ← Find(x)
  ȳ ← Find(y)

  y ← ȳ
  leader(y) ← x̄
  while (next(y) ≠ Null)
      y ← next(y)
      leader(y) ← x̄

  next(y) ← next(x̄)
  next(x̄) ← ȳ
```

The worst-case running time of Union is a constant times the size of the *larger* set. Thus, if we merge a one-element set with another n-element set, the running time can be $\Theta(n)$. Generalizing this idea, it is quite easy to come up with a sequence of n MakeSet and $n-1$ Union operations that requires $\Theta(n^2)$ time to create the set $\{1, 2, \ldots, n\}$ from scratch.

```
WorstCaseSequence(n):
  MakeSet(1)
  for i ← 2 to n
      MakeSet(i)
      Union(1, i)
```

We are being stupid in two different ways here. One is the order of operations in WorstCaseSequence. Obviously, it would be more efficient to merge the sets in the other order, or to use some sort of divide and conquer approach. Unfortunately, we can't fix this; we don't get to decide how our data structures are used! The other is that we always update the leader pointers in the larger set. To fix this, we add a comparison inside the Union algorithm to determine which set is smaller. This requires us to maintain the size of each set, but that's easy.

```
MakeWeightedSet(x):
  leader(x) ← x
  next(x) ← x
  size(x) ← 1
```

```
WeightedUnion(x, y)
  x̄ ← Find(x)
  ȳ ← Find(y)
  if size(x̄) > size(ȳ)
      Union(x̄, ȳ)
      size(x̄) ← size(x̄) + size(ȳ)
  else
      Union(ȳ, x̄)
      size(ȳ) ← size(x̄) + size(ȳ)
```

The new WeightedUnion algorithm still takes $\Theta(n)$ time to merge two n-element sets. However, in an amortized sense, this algorithm is much more efficient. Intuitively, before we can merge two large sets, we have to perform a large number of MakeWeightedSet operations.

Theorem 1. *A sequence of m MakeWeightedSet operations and n WeightedUnion operations takes $O(m + n \log n)$ time in the worst case.*

Proof: Whenever the leader of an object x is changed by a WeightedUnion, the size of the set containing x increases by at least a factor of two. By induction, if the leader of x has changed k times, the set containing x has at least 2^k members. After the sequence ends, the largest set contains at most n members. (Why?) Thus, the leader of any object x has changed at most $\lfloor \lg n \rfloor$ times.

Since each WeightedUnion reduces the number of sets by one, there are $m - n$ sets at the end of the sequence, and at most n objects are *not* in singleton sets. Since each of the non-singleton objects had $O(\log n)$ leader changes, the total amount of work done in updating the leader pointers is $O(n \log n)$. □

The aggregate method now implies that each WeightedUnion has ***amortized cost $O(\log n)$***.

11.3 Path Compression

Using unthreaded tress, Find takes logarithmic time and everything else is constant; using threaded trees, Union takes logarithmic amortized time and everything else is constant. A third method allows us to get both of these operations to have *almost* constant running time.

We start with the original unthreaded tree representation, where every object points to a parent. The key observation is that in any Find operation, once we determine the leader of an object x, we can speed up future Finds by redirecting x's parent pointer directly to that leader. In fact, we can change the parent pointers of all the ancestors of x all the way up to the root; this is easiest if we use recursion for the initial traversal up the tree. This modification to Find is called *path compression*.

Path compression during Find(c). Shaded nodes have a new parent.

```
FIND(x)
  if x ≠ parent(x)
      parent(x) ← FIND(parent(x))
  return parent(x)
```

If we use path compression, the 'depth' field we used earlier to keep the trees shallow is no longer correct, and correcting it would take way too long. But this information still ensures that Find runs in $\Theta(\log n)$ time in the worst case, so we'll just give it another name: *rank*. The following algorithm is usually called *union by rank*:

```
MakeSet(x):
  parent(x) ← x
  rank(x) ← 0
```

```
Union(x, y)
  x̄ ← Find(x)
  ȳ ← Find(y)
  if rank(x̄) > rank(ȳ)
      parent(ȳ) ← x̄
  else
      parent(x̄) ← ȳ
      if rank(x̄) = rank(ȳ)
          rank(ȳ) ← rank(ȳ) + 1
```

Find still runs in $O(\log n)$ time in the worst case; path compression increases the cost by only most a constant factor. But we have good reason to suspect that this upper bound is no longer tight. Our new algorithm memoizes the results of each Find, so if we are asked to Find the same item twice in a row, the second call returns in constant time. Splay trees used a similar strategy to achieve their optimal amortized cost, but our up-trees have fewer constraints on their structure than binary search trees, so we should get even better performance.

This intuition is exactly correct, but it takes a bit of work to define precisely *how* much better the performance is. As a first approximation, we will prove below that the amortized cost of a Find operation is bounded by the *iterated logarithm* of n, denoted $\log^* n$, which is the number of times one must take the logarithm of n before the value is less than 1:

$$\lg^* n = \begin{cases} 1 & \text{if } n \leq 2, \\ 1 + \lg^*(\lg n) & \text{otherwise.} \end{cases}$$

Our proof relies on several useful properties of ranks, which follow directly from the Union and Find algorithms.

- If a node x is not a set leader, then the rank of x is smaller than the rank of its parent.
- Whenever $parent(x)$ changes, the new parent has larger rank than the old parent.
- Whenever the leader of x's set changes, the new leader has larger rank than the old leader.
- The size of any set is exponential in the rank of its leader: $size(\overline{x}) \geq 2^{rank(\overline{x})}$. (This is easy to prove by induction, hint, hint.)
- In particular, since there are only n objects, the highest possible rank is $\lfloor \lg n \rfloor$.
- For any integer r, there are at most $n/2^r$ objects of rank r.

Only the last property requires a clever argument to prove. Fix your favorite integer r. Observe that only set leaders can change their rank. Whenever the rank of any set leader $\overline{x}$ changes from $r-1$ to r, mark all the objects in $\overline{x}$'s set. Since leader ranks can only increase over time, each object is marked at most once. There are n objects altogether, and any object with rank r marks at least 2^r objects. It follows that there are at most $n/2^r$ objects with rank r, as claimed.

*11.4 $O(\log^* n)$ Amortized Time

The following analysis of path compression was discovered just a few years ago by Raimund Seidel and Micha Sharir.[1] Previous proofs[2] relied on complicated charging schemes or potential-function arguments; Seidel and Sharir's analysis relies on a comparatively simple recursive decomposition. (Of course, simple is in the eye of the beholder.)

Seidel and Sharir phrase their analysis in terms of two more general operations on set forests. Their more general COMPRESS operation compresses *any* directed path, not just paths that lead to the root. The new SHATTER operation makes every node on a root-to-leaf path into its own parent.

```
COMPRESS(x, y):
⟨⟨y must be an ancestor of x⟩⟩
  if x ≠ y
      COMPRESS(parent(x), y)
      parent(x) ← parent(y)
```

```
SHATTER(x):
  if parent(x) ≠ x
      SHATTER(parent(x))
      parent(x) ← x
```

Clearly, the running time of FIND(x) operation is dominated by the running time of COMPRESS(x, y), where y is the leader of the set containing x. Thus, we can prove the upper bound by analyzing an *arbitrary* sequence of UNION and COMPRESS operations. Moreover, we can assume that the arguments of every UNION operation are set leaders, so that each UNION takes only constant worst-case time.

Finally, since each call to COMPRESS specifies the top node in the path to be compressed, we can reorder the sequence of operations, so that every UNION occurs before any COMPRESS, without changing the number of pointer assignments.

Top row: A COMPRESS followed by a UNION. Bottom row: The same operations in the opposite order.

Each UNION requires only constant time, so we only need to analyze the amortized cost of COMPRESS. The running time of COMPRESS is proportional to the number of parent pointer assignments, plus $O(1)$ overhead, so we will phrase our analysis in terms of pointer assignments. Let $\boldsymbol{T(m, n, r)}$ denote the worst case number of pointer assignments in any sequence of at most m COMPRESS operations, executed on a forest of at most n nodes, in which each node has rank at most r.

The following trivial upper bound will be the base case for our recursive argument.

[1] Raimund Seidel and Micha Sharir. Top-down analysis of path compression. *SIAM J. Computing* 34(3):515–525, 2005.

[2] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. Assoc. Comput. Mach.* 22:215–225, 1975.

Theorem 2. $T(m,n,r) \le nr$

Proof: Each node can change parents at most r times, because each new parent has higher rank than the previous parent. □

Fix a forest F of n nodes with maximum rank r, and a sequence C of m COMPRESS operations on F, and let $\boldsymbol{T(F,C)}$ denote the total number of pointer assignments executed by this sequence.

Let s be an arbitrary positive rank. Partition F into two sub-forests: a 'low' forest F_- containing all nodes with rank at most s, and a 'high' forest F_+ containing all nodes with rank greater than s. Since ranks increase as we follow parent pointers, every ancestor of a high node is another high node. Let n_- and n_+ denote the number of nodes in F_- and F_+, respectively. Finally, let m_+ denote the number of COMPRESS operations that involve any node in F_+, and let $m_- = m - m_+$.

Splitting the forest F (in this case, a single tree) into sub-forests F_+ and F_- at rank s.

Any sequence of COMPRESS operations on F can be decomposed into a sequence of COMPRESS operations on F_+, plus a sequence of COMPRESS and SHATTER operations on F_-, with the same total cost. This requires only one small modification to the code: We forbid any low node from having a high parent. Specifically, if x is a low node and y is a high node, we replace any assignment $parent(x) \leftarrow y$ with $parent(x) \leftarrow x$.

A COMPRESS operation in F splits into a COMPRESS operation in F_+ and a SHATTER operation in F_-

This modification is equivalent to the following reduction:

```
COMPRESS(x, y, F):                    ⟨⟨y is an ancestor of x⟩⟩
  if rank(x) > s
      COMPRESS(x, y, F+)              ⟨⟨in C+⟩⟩
  else if rank(y) ≤ s
      COMPRESS(x, y, F−)              ⟨⟨in C−⟩⟩
  else
      z ← x
      while rank(parent_F(z)) ≤ s
          z ← parent_F(z)
      COMPRESS(parent_F(z), y, F+)    ⟨⟨in C+⟩⟩
      SHATTER(x, z, F−)
      parent(z) ← z                   (?)
```

The pointer assignment in the last line (**?**) looks redundant, but it is actually necessary for the analysis. Each execution of that line mirrors an assignment of the form $parent(z) \leftarrow w$, where z is a low node, w is a high node, and the previous parent of z was also a high node. Each of these 'redundant' assignments happens immediately after a COMPRESS in the top forest, so we perform at most m_+ redundant assignments.

Each node x is touched by at most one SHATTER operation, so the total number of pointer reassignments in all the SHATTER operations is at most n.

Thus, by partitioning the forest F into F_+ and F_-, we have also partitioned the sequence C of COMPRESS operations into subsequences C_+ and C_-, with respective lengths m_+ and m_-, such that the following inequality holds:

$$\boxed{T(F,C) \le T(F_+,C_+) + T(F_-,C_-) + m_+ + n}$$

Since there are only $n/2^i$ nodes of any rank i, we have $n_+ \le \sum_{i>s} n/2^i = n/2^s$. The number of different ranks in F_+ is $r - s < r$. Thus, Theorem 2 implies the upper bound

$$T(F_+,C_+) < rn/2^s.$$

Let us fix $\boldsymbol{s = \lg r}$, so that $T(F_+,C_+) \le n$. We can now simplify our earlier recurrence to

$$T(F,C) \le T(F_-,C_-) + m_+ + 2n,$$

or equivalently,

$$T(F,C) - m \le T(F_-,C_-) - m_- + 2n.$$

Since this argument applies to *any* forest F and *any* sequence C, we have just proved that

$$T'(m,n,r) \le T'(m,n,\lfloor \lg r \rfloor) + 2n,$$

where $T'(m,n,r) = T(m,n,r) - m$. The solution to this recurrence is $\boldsymbol{T'(n,m,r) \le 2n \lg^* r}$. Voilá!

Theorem 3. $T(m,n,r) \le m + 2n \lg^* r$

*11.5 Turning the Crank

There is one place in the preceding analysis where we have significant room for improvement. Recall that we bounded the total cost of the operations on F_+ using the trivial upper bound from Theorem 2. But we just proved a better upper bound in Theorem 3! We can apply precisely the same strategy, using Theorem 3 recursively instead of Theorem 2, to improve the bound even more.

Suppose we fix $s = \lg^* r$, so that $n_+ = n/2^{\lg^* r}$. Theorem 3 implies that

$$T(F_+,C_+) \le m_+ + 2n\frac{\lg^* r}{2^{\lg^* r}} \le m_+ + 2n.$$

This implies the recurrence

$$T(F,C) \le T(F_-,C_-) + 2m_+ + 3n,$$

which in turn implies that

$$T''(m,n,r) \le T''(m,n,\lg^* r) + 3n,$$

where $T''(m,n,r) = T(m,n,r) - 2m$. The solution to this equation is $\boldsymbol{T(m,n,r) \le 2m + 3n \lg^{**} r}$, where $\lg^{**} r$ is the *iterated* iterated logarithm of r:

$$\lg^{**} r = \begin{cases} 1 & \text{if } r \le 2, \\ 1 + \lg^{**}(\lg^{*} r) & \text{otherwise.} \end{cases}$$

Naturally we can apply the same improvement strategy again, and again, as many times as we like, each time producing a tighter upper bound. Applying the reduction c times, for any positive integer c, gives us $\boldsymbol{T(m,n,r) \le cm + (c+1)n \lg^{*^c} r}$, where

$$\lg^{*^c} r = \begin{cases} \lg r & \text{if } c = 0, \\ 1 & \text{if } r \le 2, \\ 1 + \lg^{*^c}(\lg^{*^{c-1}} r) & \text{otherwise.} \end{cases}$$

Each time we 'turn the crank', the dependence on m increases, while the dependence on n and r decreases. For sufficiently large values of c, the cm term dominates the time bound, and further iterations only make things worse. The point of diminishing returns can be estimated by *the minimum number of stars* such that $\lg^{**\cdots*} r$ is smaller than a constant:

$$\alpha(r) = \min\left\{c \ge 1 \;\middle|\; \lg^{*^c} n \le 3\right\}.$$

(The threshold value 3 is used here because $\lg^{*^c} 5 \ge 2$ for all c.) By setting $c = \alpha(r)$, we obtain our final upper bound.

Theorem 4. $T(m,n,r) \le m\alpha(r) + 3n(\alpha(r)+1)$

We can assume without loss of generality that $m \ge n$ by ignoring any singleton sets, so this upper bound can be further simplified to $T(m,n,r) = O(m\alpha(r)) = O(m\alpha(n))$. It follows that if we use union by rank, FIND with path compression runs in $\boldsymbol{O(\alpha(n))}$ ***amortized time***.

Even this upper bound is somewhat conservative if m is larger than n. A closer estimate is given by the function

$$\alpha(m,n) = \min\left\{c \ge 1 \;\middle|\; \lg^{*^c}(\lg n) \le m/n\right\}.$$

It's not hard to prove that if $m = \Theta(n)$, then $\alpha(m,n) = \Theta(\alpha(n))$. On the other hand, if $m \ge n \lg^{*****} n$, for any constant number of stars, then $\alpha(m,n) = O(1)$. So even if the number of FIND operations is only *slightly* larger than the number of nodes, the amortized cost of each FIND is *constant*.

$O(\alpha(m,n))$ is actually a *tight* upper bound for the amortized cost of path compression; there are no more tricks that will improve the analysis further. More surprisingly, this is the best amortized bound we obtain for *any* pointer-based data structure for maintaining disjoint sets; the amortized cost of *every* FIND algorithm is at least $\Omega(\alpha(m,n))$. The proof of the matching lower bound is, unfortunately, far beyond the scope of this class.[3]

[3] Robert E. Tarjan. A class of algorithms which require non-linear time to maintain disjoint sets. *J. Comput. Syst. Sci.* 19:110–127, 1979.

11.6 The Ackermann Function and its Inverse

The iterated logarithms that fell out of our analysis of path compression are the inverses of a hierarchy of recursive functions defined by Wilhelm Ackermann in 1928.[4]

$$\mathbf{2} \uparrow^{\boldsymbol{c}} \boldsymbol{n} := \begin{cases} 2 & \text{if } n = 1 \\ 2n & \text{if } c = 0 \\ 2 \uparrow^{c-1} (2 \uparrow^{c} (n-1)) & \text{otherwise} \end{cases}$$

For each fixed integer c, the function $2 \uparrow^c n$ is monotonically increasing in n, and these functions grow *incredibly* faster as the index c increases. $\mathbf{2} \uparrow \boldsymbol{n}$ is the familiar power function 2^n. $\mathbf{2} \uparrow\uparrow \boldsymbol{n}$ is the *tower* function:

$$2 \uparrow\uparrow n = \underbrace{2 \uparrow 2 \uparrow \ldots \uparrow 2}_{n} = 2^{2^{2^{\cdot^{\cdot^{2}}}}}\Big\}_{n}$$

John Conway named $2 \uparrow\uparrow\uparrow n$ the *wower* function:

$$2 \uparrow\uparrow\uparrow n = \underbrace{2 \uparrow\uparrow 2 \uparrow\uparrow \cdots \uparrow\uparrow 2}_{n}.$$

And so on, *et cetera, ad infinitum*.

For any fixed c, the function $\log^{*^c} n$ is the inverse of the function $2 \uparrow^{c+1} n$, the $(c+1)$th row in the Ackerman hierarchy. Thus, for any remotely reasonable values of n, say $n \le 2^{256}$, we have $\log^* n \le 5$, $\log^{**} n \le 4$, and $\log^{*^c} n \le 3$ for any $c \ge 3$.

The function $\alpha(n)$ is usually called the *inverse Ackerman function*.[5] Our earlier definition is equivalent to $\alpha(n) = \min\{c \ge 1 \mid 2 \uparrow^{c+2} 3 \ge n\}$; in other words, $\alpha(n) + 2$ is the inverse of the third column in the Ackermann hierarchy. The function $\alpha(n)$ grows *much* more slowly than $\log^{*^c} n$ for any fixed c; we have $\alpha(n) \le 3$ for all even *remotely imaginable* values of n. Nevertheless, the function $\alpha(n)$ is eventually larger than any constant, so it is *not* $O(1)$.

$2 \uparrow^c n$	$n = 1$	2	$n = 3$	$n = 4$	$n = 5$
$2n$	2	4	6	8	10
$2 \uparrow n$	2	4	8	16	32
$2 \uparrow\uparrow n$	2	4	16	65536	2^{65536}
$2 \uparrow\uparrow\uparrow n$	2	4	65536	$2^{2^{2^{\cdot^{\cdot^{2}}}}}\Big\}_{65536}$	$2^{2^{2^{\cdot^{\cdot^{2}}}}}\Big\}_{2^{2^{2^{\cdot^{\cdot^{2}}}}}\Big\}_{65536}}$
$2 \uparrow\uparrow\uparrow\uparrow n$	2	4	$2^{2^{2^{\cdot^{\cdot^{2}}}}}\Big\}_{65536}$	$2^{2^{\cdots 2}}\Big\}^{2^{\cdots 2}}\Big\}^{\cdot^{\cdot^{2^{\cdots 2}}\Big\}_{65536}}}\Big\}_{2^{2^{2^{\cdot^{\cdot^{2}}}}}\Big\}_{65536}}$	⟨⟨*Yeah, right.*⟩⟩
$2 \uparrow\uparrow\uparrow\uparrow\uparrow n$	2	4	$2^{2^{\cdots 2}}\Big\}^{2^{\cdots 2}}\Big\}^{\cdot^{\cdot^{2^{\cdots 2}}\Big\}_{65536}}}\Big\}_{2^{2^{2^{\cdot^{\cdot^{2}}}}}\Big\}_{65536}}$	⟨⟨*Very funny.*⟩⟩	⟨⟨*Argh! My eyes!*⟩⟩

Small (!!) values of Ackermann's functions.

[4]Ackermann didn't define his functions this way—I'm actually describing a slightly cleaner hierarchy defined 35 years later by R. Creighton Buck—but the exact details of the definition are surprisingly irrelevant! The mnemonic up-arrow notation for these functions was introduced by Don Knuth in the 1970s.

[5]Strictly speaking, the name 'inverse Ackerman function' is inaccurate. One good formal definition of the true inverse Ackerman function is $\tilde{\alpha}(n) = \min\left\{c \ge 1 \;\middle|\; \lg^{*^c} n \le c\right\} = \min\left\{c \ge 1 \;\middle|\; 2 \uparrow^{c+2} c \ge n\right\}$. However, it's not hard to prove that $\tilde{\alpha}(n) \le \alpha(n) \le \tilde{\alpha}(n) + 1$ for all sufficiently large n, so the inaccuracy is completely forgivable. As I said in the previous footnote, the exact details of the definition are surprisingly irrelevant!

11.7 To infinity... and beyond!

Of course, one can generalize the inverse Ackermann function to functions that grow arbitrarily more slowly, starting with the *iterated* inverse Ackermann function

$$\alpha^*(n) = \begin{cases} 1 & \text{if } n \le 4, \\ 1 + \alpha^*(\alpha(n)) & \text{otherwise,} \end{cases}$$

then the *iterated* iterated iterated inverse Ackermann function

$$\alpha^{*^c}(n) = \begin{cases} \alpha(n) & \text{if } c = 0, \\ 1 & \text{if } n \le 4, \\ 1 + \alpha^{*^c}(\alpha^{*^{c-1}}(n)) & \text{otherwise,} \end{cases}$$

and then the diagonalized inverse Ackermann function

$$\textit{Head-asplode}(n) = \min\{c \ge 1 \mid \alpha^{*^c} n \le 4\},$$

and so on ad nauseam. Fortunately(?), such functions appear extremely rarely in algorithm analysis. Perhaps the only naturally-occurring example of a super-constant sub-inverse-Ackermann function is a recent result of Seth Pettie[6], who proved that if a splay tree is used as a double-ended queue — insertions and deletions of only smallest or largest elements — then the amortized cost of any operation is $O(\alpha^*(n))$!

Exercises

1. Consider the following solution for the union-find problem, called *union-by-weight*. Each set leader $\overline{x}$ stores the number of elements of its set in the field $\textit{weight}(\overline{x})$. Whenever we UNION two sets, the leader of the *smaller* set becomes a new child of the leader of the *larger* set (breaking ties arbitrarily).

```
MakeSet(x):
  parent(x) ← x
  weight(x) ← 1

Find(x):
  while x ≠ parent(x)
      x ← parent(x)
  return x

Union(x, y)
  x̄ ← Find(x)
  ȳ ← Find(y)
  if weight(x̄) > weight(ȳ)
      parent(ȳ) ← x̄
      weight(x̄) ← weight(x̄) + weight(ȳ)
  else
      parent(x̄) ← ȳ
      weight(x̄) ← weight(x̄) + weight(ȳ)
```

   Prove that if we use union-by-weight, the *worst-case* running time of FIND(x) is $O(\log n)$, where n is the cardinality of the set containing x.

2. Consider a union-find data structure that uses union by depth (or equivalently union by rank) *without* path compression. For all integers m and n such that $m \ge 2n$, prove that there is a sequence of n *MakeSet* operations, followed by m UNION and FIND operations, that require $\Omega(m \log n)$ time to execute.

[6] Splay trees, Davenport-Schinzel sequences, and the deque conjecture. *Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1115–1124, 2008.

3. Suppose you are given a collection of up-trees representing a partition of the set $\{1, 2, \ldots, n\}$ into disjoint subsets. **You have no idea how these trees were constructed.** You are also given an array $node[1..n]$, where $node[i]$ is a pointer to the up-tree node containing element i. Your task is to create a new array $label[1..n]$ using the following algorithm:

```
LabelEverything:
  for i ← 1 to n
      label[i] ← Find(node[i])
```

   (a) What is the worst-case running time of LabelEverything if we implement Find *without* path compression?

   (b) ***Prove*** that if we implement Find using path compression, LabelEverything runs in $O(n)$ time in the worst case.

4. Consider an arbitrary sequence of m MakeSet operations, followed by u Union operations, followed by f Find operations, and let $n = m + u + f$. Prove that if we use union by rank and Find with path compression, all n operations are executed in $O(n)$ time.

5. Suppose we want to maintain an array $X[1..n]$ of bits, which are all initially zero, subject to the following operations.

   - Lookup(i): Given an index i, return $X[i]$.
   - Blacken(i): Given an index $i < n$, set $X[i] \leftarrow 1$.
   - NextWhite(i): Given an index i, return the smallest index $j \geq i$ such that $X[j] = 0$. (Because we never change $X[n]$, such an index always exists.)

   If we use the array $X[1..n]$ itself as the only data structure, it is trivial to implement Lookup and Blacken in $O(1)$ time and NextWhite in $O(n)$ time. But you can do better! Describe data structures that support Lookup in $O(1)$ worst-case time and the other two operations in the following time bounds. (We want a different data structure for each set of time bounds, not one data structure that satisfies all bounds simultaneously!)

   (a) The worst-case time for both Blacken and NextWhite is $O(\log n)$.

   (b) The amortized time for both Blacken and NextWhite is $O(\log n)$. In addition, the *worst-case* time for Blacken is $O(1)$.

   (c) The amortized time for Blacken is $O(\log n)$, and the *worst-case* time for NextWhite is $O(1)$.

   (d) The worst-case time for Blacken is $O(1)$, and the amortized time for NextWhite is $O(\alpha(n))$. *[Hint: There is no Whiten.]*

6. Suppose we want to maintain a collection of strings (sequences of characters) under the following operations:

   - NewString(a) creates a new string of length 1 containing only the character a and returns a pointer to that string.

- CONCAT(S, T) removes the strings S and T (given by pointers) from the data structure, adds the concatenated string ST to the data structure, and returns a pointer to the new string.
- REVERSE(S) removes the string S (given by a pointer) from the data structure, adds the reversal of S to the data structure, and returns a pointer to the new string.
- LOOKUP(S, k) returns the kth character in string S (given by a pointer), or NULL if the length of the S is less than k.

Describe and analyze a *simple* data structure that supports CONCAT in $O(\log n)$ *amortized* time, supports every other operation in $O(1)$ *worst-case* time, and uses $O(n)$ space, where n is the sum of the *current* string lengths. Unlike the similar problem in the previous lecture note, there is no SPLIT operation. *[Hint: Why is this problem here?]*

7. (a) Describe and analyze an algorithm to compute the size of the largest connected component of black pixels in an $n \times n$ bitmap $B[1..n, 1..n]$.

For example, given the bitmap below as input, your algorithm should return the number 9, because the largest conected black component (marked with white dots on the right) contains nine pixels.

(b) Design and analyze an algorithm BLACKEN(i, j) that colors the pixel $B[i, j]$ black and returns the size of the largest black component in the bitmap. For full credit, the *amortized* running time of your algorithm (starting with an all-white bitmap) must be as small as possible.

For example, at each step in the sequence below, we blacken the pixel marked with an X. The largest black component is marked with white dots; the number underneath shows the correct output of the BLACKEN algorithm.

(c) What is the *worst-case* running time of your BLACKEN algorithm?

*8. Consider the following game. I choose a positive integer n and keep it secret; your goal is to discover this integer. We play the game in rounds. In each round, you write a list of *at most* n integers on the blackboard. If you write more than n numbers in a single round, you lose. (Thus, in the first round, you must write only the number 1; do you see why?) If n is one of the numbers you wrote, you win the game; otherwise, I announce which of

the numbers you wrote is smaller or larger than n, and we proceed to the next round. For example:

You	Me
1	It's bigger than 1.
4, 42	It's between 4 and 42.
8, 15, 16, 23, 30	It's between 8 and 15.
9, 10, 11, 12, 13, 14	It's 11; you win!

Describe a strategy that allows you to win in $O(\alpha(n))$ rounds!

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms for the most recent revision.

It was a Game called Yes and No, where Scrooge's nephew had to think of something, and the rest must find out what; he only answering to their questions yes or no, as the case was. The brisk fire of questioning to which he was exposed, elicited from him that he was thinking of an animal, a live animal, rather a disagreeable animal, a savage animal, an animal that growled and grunted sometimes, and talked sometimes, and lived in London, and walked about the streets, and wasn't made a show of, and wasn't led by anybody, and didn't live in a menagerie, and was never killed in a market, and was not a horse, or an ass, or a cow, or a bull, or a tiger, or a dog, or a pig, or a cat, or a bear. At every fresh question that was put to him, this nephew burst into a fresh roar of laughter; and was so inexpressibly tickled, that he was obliged to get up off the sofa and stamp. At last the plump sister, falling into a similar state, cried out :

"I have found it out! I know what it is, Fred! I know what it is!"

"What is it?" cried Fred.

"It's your Uncle Scro-o-o-o-oge!"

Which it certainly was. Admiration was the universal sentiment, though some objected that the reply to "Is it a bear?" ought to have been "Yes;" inasmuch as an answer in the negative was sufficient to have diverted their thoughts from Mr Scrooge, supposing they had ever had any tendency that way.

— Charles Dickens, *A Christmas Carol* (1843)

12 Lower Bounds

12.1 Huh? Whuzzat?

So far in this class we've been developing algorithms and data structures to solve certain problems as quickly as possible. Starting with this lecture, we'll turn the tables, by proving that certain problems *cannot* be solved as quickly as we might like them to be.

Let $T_A(X)$ denote the running time of algorithm A given input X. For most of the semester, we've been concerned with the the worst-case running time of A as a function of the input size:

$$T_A(n) := \max_{|X|=n} T_A(X).$$

The worst-case complexity of a *problem* Π is the worst-case running time of the *fastest* algorithm for solving it:

$$T_\Pi(n) := \min_{A \text{ solves } \Pi} T_A(n) = \min_{A \text{ solves } \Pi} \max_{|X|=n} T_A(X).$$

Any algorithm A that solves Π immediately implies an *upper bound* on the complexity of Π; the inequality $\boldsymbol{T_\Pi(n) \le T_A(n)}$ follows directly from the definition of T_Π. Just as obviously, faster algorithms give us better (smaller) upper bounds. In other words, whenever we give a running time for an algorithm, what we're really doing—and what most computer scientists devote their entire careers doing[1]—is bragging about how *easy* some problem is.

Now, instead of bragging about how easy problems are, we will argue that certain problems are *hard*, by proving *lower bounds* on their complexity. This is considerably harder than proving

[1]This sometimes leads to long sequences of results that sound like an obscure version of "Name that Tune":

Lennes: "I can triangulate that polygon in $O(n^2)$ time."
Shamos: "I can triangulate that polygon in $O(n \log n)$ time."
Tarjan: "I can triangulate that polygon in $O(n \log\log n)$ time."
Seidel: "I can triangulate that polygon in $O(n \log^* n)$ time." [Audience gasps.]
Chazelle: "I can triangulate that polygon in $O(n)$ time." [Audience gasps and applauds.]
"Triangulate that polygon!"

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

an upper bound, because it's no longer enough to examine a single algorithm. To prove an inequality of the form $T_\Pi(n) = \Omega(f(n))$, we must prove that ***every*** algorithm that solves Π has a worst-case running time $\Omega(f(n))$, or equivalently, that ***no*** algorithm runs in $o(f(n))$ time.

12.2 Decision Trees

Unfortunately, there is no formal definition of the phrase 'all algorithms'![2] So when we derive lower bounds, we first have to specify *precisely* what kinds of algorithms we will consider and *precisely* how to measure their running time. This specification is called a ***model of computation***.

One rather powerful model of computation—and the only model we'll talk about in this lecture—is the ***decision tree*** model. A decision tree is, as the name suggests, a tree. Each internal node in the tree is labeled by a *query*, which is just a question about the input. The edges out of a node correspond to the possible answers to that node's query. Each leaf of the tree is labeled with an *output*. To compute with a decision tree, we start at the root and follow a path down to a leaf. At each internal node, the answer to the query tells us which node to visit next. When we reach a leaf, we output its label.

For example, the guessing game where one person thinks of an animal and the other person tries to figure it out with a series of yes/no questions can be modeled as a decision tree. Each internal node is labeled with a question and has two edges labeled 'yes' and 'no'. Each leaf is labeled with an animal.

A decision tree to choose one of six animals.

Here's another simple and familiar example, called the ***dictionary problem***. Let A be a fixed array with n numbers. Suppose we want to determine, given a number x, the position of x in the array A, if any. One solution to the dictionary problem is to sort A (remembering every element's original position) and then use binary search. The (implicit) binary search tree can be used almost directly as a decision tree. Each internal node in the *search* tree stores a key k; the corresponding node in the *decision* tree stores the question 'Is $x < k$?'. Each leaf in the *search* tree stores some value $A[i]$; the corresponding node in the *decision* tree asks 'Is $x = A[i]$?' and has two leaf children, one labeled 'i' and the other 'none'.

We *define* the running time of a decision tree algorithm for a given input to be the number of queries in the path from the root to the leaf. For example, in the 'Guess the animal' tree above,

[2]Complexity-theory ~~snobs~~ purists sometimes argue that 'all algorithms' is just a synonym for 'all Turing machines'. This is utter nonsense; Turing machines are just another model of computation. Turing machines *might* be a reasonable abstraction of *physically realizable* computation—that's the Church-Turing thesis—but it has a few problems. First, computation is an abstract mathematical process, not a physical process. Algorithms that use physically unrealistic components (like exact real numbers, or unbounded memory) are still mathematically well-defined and still provide useful intuition about real-world computation. Moreover, Turing machines don't accurately reflect the complexity of physically realizable algorithms, because (for example) they can't do arithmetic or access arbitrary memory locations in constant time. At best, they estimate algorithmic complexity up to polynomial factors (although even that is unknown).

Left: A binary search tree for the first eight primes.
Right: The corresponding binary decision tree for the dictionary problem (- = 'none').

$T(\text{frog}) = 2$. Thus, the worst-case running time of the algorithm is just the depth of the tree. This definition ignores other kinds of operations that the algorithm might perform that have nothing to do with the queries. (Even the most efficient binary search problem requires more than one machine instruction per comparison!) But the number of decisions is certainly a *lower bound* on the actual running time, which is good enough to prove a lower bound on the complexity of a problem.

Both of the examples describe *binary* decision trees, where every query has only two answers. We may sometimes want to consider decision trees with higher degree. For example, we might use queries like 'Is x greater than, equal to, or less than y?' or 'Are these three points in clockwise order, colinear, or in counterclockwise order?' A *k-ary* decision tree is one where every query has (at most) k different answers. **From now on, I will only consider k-ary decision trees where k is a constant.**

12.3 Information Theory

Most lower bounds for decision trees are based on the following simple observation: ***The answers to the queries must give you enough information to specify any possible output.*** If a problem has N different outputs, then obviously any decision tree must have at least N leaves. (It's possible for several leaves to specify the same output.) Thus, if every query has at most k possible answers, then the depth of the decision tree must be at least $\lceil \log_k N \rceil = \Omega(\log N)$.

Let's apply this to the dictionary problem for a set S of n numbers. Since there are $n+1$ possible outputs, any decision tree must have at least $n+1$ leaves, and thus any decision tree must have depth at least $\lceil \log_k (n+1) \rceil = \Omega(\log n)$. So the complexity of the dictionary problem, in the decision-tree model of computation, is $\Omega(\log n)$. This matches the upper bound $O(\log n)$ that comes from a perfectly-balanced binary search tree. That means that the standard binary search algorithm, which runs in $O(\log n)$ time, is *optimal*—there is no faster algorithm in this model of computation.

12.4 But wait a second...

We can solve the membership problem in $O(1)$ expected time using hashing. Isn't this inconsistent with the $\Omega(\log n)$ lower bound?

No, it isn't. The reason is that hashing involves a query with more than a constant number of outcomes, specifically 'What is the hash value of x?' In fact, if we don't restrict the degree of the decision tree, we can get constant running time even without hashing, by using the obviously unreasonable query 'For which index i (if any) is $A[i] = x$?'. No, I am *not* cheating — remember that the decision tree model allows us to ask *any* question about the input!

This example illustrates a common theme in proving lower bounds: *choosing the right model of computation is absolutely crucial.* If you choose a model that is too powerful, the problem you're studying may have a completely trivial algorithm. On the other hand, if you consider more restrictive models, the problem may not be solvable at all, in which case any lower bound will be meaningless! (In this class, we'll just tell you the right model of computation to use.)

12.5 Sorting

Now let's consider the classical *sorting* problem — Given an array of n numbers, arrange them in increasing order. Unfortunately, decision trees don't have any way of describing moving data around, so we have to rephrase the question slightly:

> Given a sequence $\langle x_1, x_2, \ldots, x_n \rangle$ of n distinct numbers, find the permutation π such that $x_{\pi(1)} < x_{\pi(2)} < \cdots < x_{\pi(n)}$.

Now a k-ary decision-tree lower bound is immediate. Since there are $n!$ possible permutations π, any decision tree for sorting must have at least $n!$ leaves, and so must have depth $\Omega(\log(n!))$. To simplify the lower bound, we apply *Stirling's approximation*

$$n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\left(1 + \Theta\left(\frac{1}{n}\right)\right) > \left(\frac{n}{e}\right)^n.$$

This gives us the lower bound

$$\lceil \log_k(n!) \rceil > \left\lceil \log_k \left(\frac{n}{e}\right)^n \right\rceil = \lceil n \log_k n - n \log_k e \rceil = \Omega(n \log n).$$

This matches the $O(n \log n)$ upper bound that we get from mergesort, heapsort, or quicksort, so those algorithms are optimal. The decision-tree complexity of sorting is $\Theta(n \log n)$.

Well... we're not quite done. In order to say that those algorithms are optimal, we have to demonstrate that they fit into our model of computation. A few minutes of thought will convince you that they can be described as a special type of decision tree called a *comparison* tree, where every query is of the form 'Is x_i bigger or smaller than x_j?' These algorithms treat any two input sequences exactly the same way as long as the same comparisons produce exactly the same results. This is a feature of any comparison tree. In other words, *the actual input values don't matter, only their order*. Comparison trees describe almost all well-known sorting algorithms: bubble sort, selection sort, insertion sort, shell sort, quicksort, heapsort, mergesort, and so forth—but *not* radix sort or bucket sort.

12.6 Finding the Maximum and Adversaries

Finally let's consider the ***maximum problem***: Given an array X of n numbers, find its largest entry. Unfortunately, there's no hope of proving a lower bound in this formulation, since there are an infinite number of possible answers, so let's rephrase it slightly.

> Given a sequence $\langle x_1, x_2, \ldots, x_n \rangle$ of n distinct numbers, find the index m such that x_m is the largest element in the sequence.

We can get an upper bound of $n-1$ comparisons in several different ways. The easiest is probably to start at one end of the sequence and do a linear scan, maintaining a current maximum. Intuitively, this seems like the best we can do, but the information-theoretic bound is

only $\lceil \log_2 n \rceil$. And in fact, this bound is tight! We can locate the maximum element by asking only $\lceil \log_2 n \rceil$ 'unreasonable' questions like "Is the index of the maximum element odd?" No, this is *not* cheating—the decision tree model allows *arbitrary* questions.

To prove a non-trivial lower bound for this problem, we must do two things. First, we need to consider a more reasonable model of computation, by restricting the kinds of questions the algorithm is allowed to ask. We will consider the ***comparison tree model***, where every query must have the form "Is $x_i > x_j$?". Since most algorithms[3] for finding the maximum rely on comparisons to make control-flow decisions, this does not seem like an unreasonable restriction.

Second, we will use something called an ***adversary argument***. The idea is that an all-powerful malicious adversary *pretends* to choose an input for the algorithm. When the algorithm asks a question about the input, the adversary answers in whatever way will make the algorithm do the most work. If the algorithm does not ask enough queries before terminating, then there will be several different inputs, each consistent with the adversary's answers, that should result in different outputs. In this case, whatever the algorithm outputs, the adversary can 'reveal' an input that is consistent with its answers, but contradicts the algorithm's output, and then claim that that was the input that he was using all along.

For the maximum problem, the adversary originally pretends that $x_i = i$ for all i, and answers all comparison queries accordingly. Whenever the adversary reveals that $x_i < x_j$, he *marks* x_i as an item that the algorithm knows (or should know) is not the maximum element. At most one element x_i is marked after each comparison. Note that x_n is never marked. If the algorithm does less than $n-1$ comparisons before it terminates, the adversary must have at least one other unmarked element $x_k \neq x_n$. In this case, the adversary can change the value of x_k from k to $n+1$, making x_k the largest element, without being inconsistent with any of the comparisons that the algorithm has performed. In other words, the algorithm cannot tell that the adversary has cheated. However, x_n is the maximum element in the original input, and x_k is the largest element in the modified input, so the algorithm cannot possibly give the correct answer for both cases. Thus, in order to be correct, any algorithm must perform at least $n-1$ comparisons.

The adversary argument we described has two very important properties. First, no algorithm can distinguish between a malicious adversary and an honest user who actually chooses an input in advance and answers all queries truthfully. But much more importantly, **the adversary makes *absolutely no* assumptions about the order in which the algorithm performs comparisons.** The adversary forces *any* comparison-based algorithm[4] to either perform $n-1$ comparisons, or to give the wrong answer for at least one input sequence.

Exercises

0. Simon bar Kokhba thinks of an integer between 1 and 1,000,000 (or so he claims). You are trying to determine his number by asking as few yes/no questions as possible. How many yes/no questions are required to determine Simon's number in the worst case? Give both an upper bound (supported by an algorithm) and a lower bound.

1. Consider the following *multi-dictionary* problem. Let $A[1..n]$ be a fixed array of distinct integers. Given an array $X[1..k]$, we want to find the position (if any) of each integer

[3] but not all—see Exercise **??**

[4] In fact, the $n-1$ lower bound for finding the maximum holds in a more powerful model called *algebraic* decision trees, which are binary trees where every query is a comparison between two polynomial functions of the input values, such as 'Is $x_1^2 - 3x_2x_3 + x_4^{17}$ bigger or smaller than $5 + x_1x_3^5x_5^2 - 2x_7^{42}$?'

$X[i]$ in the array A. In other words, we want to compute an array $I[1..k]$ where for each i, either $I[i] = 0$ (so zero means 'none') or $A[I[i]] = X[i]$. Determine the *exact* complexity of this problem, as a function of n and k, in the binary decision tree model.

2. We say that an array $A[1..n]$ is *k-sorted* if it can be divided into k blocks, each of size n/k, such that the elements in each block are larger than the elements in earlier blocks, and smaller than elements in later blocks. The elements within each block need not be sorted.

   For example, the following array is 4-sorted:

1	2	4	3	7	6	8	5	10	11	9	12	15	13	16	14

   (a) Describe an algorithm that k-sorts an arbitrary array in $O(n \log k)$ time.

   (b) Prove that any comparison-based k-sorting algorithm requires $\Omega(n \log k)$ comparisons in the worst case.

   (c) Describe an algorithm that completely sorts an already k-sorted array in $O(n \log(n/k))$ time.

   (d) Prove that any comparison-based algorithm to completely sort a k-sorted array requires $\Omega(n \log(n/k))$ comparisons in the worst case.

   In all cases, you can assume that n/k is an integer.

3. Recall the nuts-and-bolts problem from the lecture on randomized algorithms. We are given n bolts and n nuts of different sizes, where each bolt exactly matches one nut. Our goal is to find the matching nut for each bolt. The nuts and bolts are too similar to compare directly; however, we can test whether any nut is too big, too small, or the same size as any bolt.

   (a) Prove that in the worst case, $\Omega(n \log n)$ nut-bolt tests are required to correctly match up the nuts and bolts.

   (b) Now suppose we would be happy to find *most* of the matching pairs. Prove that in the worst case, $\Omega(n \log n)$ nut-bolt tests are required even to find $n/2$ arbitrary matching nut-bolt pairs.

   *(c) Prove that in the worst case, $\Omega(n + k \log n)$ nut-bolt tests are required to find k arbitrary matching pairs. *[Hint: Use an adversary argument for the $\Omega(n)$ term.]*

   *(d) Describe a randomized algorithm that finds k matching nut-bolt pairs in $O(n + k \log n)$ expected time.

*4. Suppose you want to determine the largest number in an n-element set $X = \{x_1, x_2, \ldots, x_n\}$, where each element x_i is an integer between 1 and $2^m - 1$. Describe an algorithm that solves this problem in $O(n + m)$ steps, where at each step, your algorithm compares one of the elements x_i with a *constant*. In particular, your algorithm must never actually compare two elements of X! *[Hint: Construct and maintain a nested set of 'pinning intervals' for the numbers that you have not yet removed from consideration, where each interval but the largest is either the upper half or lower half of the next larger block.]*

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms for the most recent revision.

An adversary means opposition and competition,
but not having an adversary means grief and loneliness.

— Zhuangzi (Chuang-tsu) c. 300 BC

It is possible that the operator could be hit by an asteroid and your $20 could fall off his cardboard box and land on the ground, and while you were picking it up, $5 could blow into your hand. You therefore could win $5 by a simple twist of fate.

— Penn Jillette, explaining how to win at Three-Card Monte (1999)

13 Adversary Arguments

13.1 Three-Card Monte

Until Times Square was turned into a glitzy sanitized tourist trap, you could often find dealers stealing tourists' money using a game called "Three Card Monte" or "Spot the Lady". The dealer show the tourist three cards, say the Queen of Hearts, the two of spades, and three of clubs. The dealer shuffles the cards face down on a table (usually slowly enough that the tourist can follow the Queen), and then asks the tourist to bet on which card is the Queen. In principle, the tourist's odds of winning are at least one in three, more if the tourist was carefully watching the movement of the cards.

In practice, however, the tourist *never* wins, because the dealer cheats. The dealer actually holds at least *four* cards; before he even starts shuffling the cards, the dealer palms the queen or sticks it up his sleeve. No matter what card the tourist bets on, the dealer turns over a black card (which might be the two of clubs, but most tourists won't notice that wasn't one o the original cards). If the tourist gives up, the dealer slides the queen under one of the cards and turns it over, showing the tourist 'where the queen was all along'. If the dealer is really good, the tourist won't see the dealer changing the cards and will think maybe the queen *was* there all along and he just wasn't smart enough to figure that out. As long as the dealer doesn't reveal all the black cards at once, the tourist has no way to prove that the dealer cheated![1]

13.2 n-Card Monte

Now let's consider a similar game, but with an algorithm acting as the tourist and with bits instead of cards. Suppose we have an array of n bits and we want to determine if any of them is a 1. Obviously we can figure this out by just looking at every bit, but can we do better? Is there maybe some complicated tricky algorithm to answer the question "Any ones?" without looking at every bit? Well, of course not, but how do we prove it?

The simplest proof technique is called an *adversary* argument. The idea is that an all-powerful malicious adversary (the dealer) *pretends* to choose an input for the algorithm (the tourist). When the algorithm wants looks at a bit (a card), the adversary sets that bit to whatever value will make the algorithm do the most work. If the algorithm does not look at enough bits before terminating, then there will be several different inputs, each consistent with the bits already seen,

[1]Even if the dealer is a sloppy magician, he'll cheat anyway. The dealer is almost always surrounded by shills; these are the "tourists" who look like they're actually winning, who turn over cards when the dealer "isn't looking", who casually mention how easy the game is to win, and so on. The shills physically protect the dealer from any angry tourists who notice the dealer cheating, and shake down any tourists who refuse to pay after making a bet. Really, you ***cannot*** win this game, ***ever***.

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

the should result in different outputs. Whatever the algorithm outputs, the adversary can 'reveal' an input that is has all the examined bits but contradicts the algorithm's output, and then claim that that was the input that he was using all along. Since the only information the algorithm has is the set of bits it examined, the algorithm cannot distinguish between a malicious adversary and an honest user who actually chooses an input in advance and answers all queries truthfully.

For the n-card monte problem, the adversary originally pretends that the input array is all zeros—whenever the algorithm looks at a bit, it sees a 0. Now suppose the algorithms stops before looking at all three bits. If the algorithm says 'No, there's no 1,' the adversary changes one of the unexamined bits to a 1 and shows the algorithm that it's wrong. If the algorithm says 'Yes, there's a 1,' the adversary reveals the array of zeros and again proves the algorithm wrong. Either way, the algorithm cannot tell that the adversary has cheated.

One absolutely crucial feature of this argument is that *the adversary makes absolutely* ***no*** *assumptions about the algorithm*. The adversary strategy can't depend on some predetermined order of examining bits, and it doesn't care about anything the algorithm might or might not do when it's not looking at bits. *Any* algorithm that doesn't examine every bit falls victim to the adversary.

13.3 Finding Patterns in Bit Strings

Let's make the problem a little more complicated. Suppose we're given an array of n bits and we want to know if it contains the substring 01, a zero followed immediately by a one. Can we answer this question without looking at every bit?

It turns out that if n is odd, we *don't* have to look at all the bits. First we look the bits in every even position: $B[2], B[4], \ldots, B[n-1]$. If we see $B[i] = 0$ and $B[j] = 1$ for any $i < j$, then we know the pattern 01 is in there somewhere—starting at the last 0 before $B[j]$—so we can stop without looking at any more bits. If we see only 1s followed by 0s, we don't have to look at the bit between the last 0 and the first 1. If every even bit is a 0, we don't have to look at $B[1]$, and if every even bit is a 1, we don't have to look at $B[n]$. In the worst case, our algorithm looks at only $n-1$ of the n bits.

But what if n is even? In that case, we can use the following adversary strategy to show that any algorithm *does* have to look at every bit. The adversary will attempt to produce an 'input' string B *without* the substring 01; all such strings have the form $11\ldots100\ldots0$. The adversary maintains two indices ℓ and r and pretends that the prefix $B[1..\ell]$ contains only 1s and the suffix $B[r..n]$ contains only 0s. Initially $\ell = 0$ and $r = n+1$.

$$\mathbf{111111}\square\square\square\square\square\square\mathbf{0000}$$
$$\uparrow \qquad\qquad \uparrow$$
$$\ell \qquad\qquad r$$

What the adversary is thinking; □ represents an unknown bit.

The adversary maintains the invariant that $r - \ell$, the length of the undecided portion of the 'input' string, is even. When the algorithm looks at a bit between ℓ and r, the adversary chooses whichever value preserves the parity of the intermediate chunk of the array, and then moves either ℓ or r. Specifically, here's what the adversary does when the algorithm examines bit $B[i]$. (Note that I'm specifying the adversary strategy as an algorithm!)

```
HIDE01(i):
  if i ≤ ℓ
      B[i] ← 1
  else if i ≥ r
      B[i] ← 0
  else if i − ℓ is even
      B[i] ← 0
      r ← i
  else
      B[i] ← 1
      ℓ ← i
```

It's fairly easy to prove that this strategy forces the algorithm to examine every bit. If the algorithm doesn't look at every bit to the right of r, the adversary could replace some unexamined bit with a 1. Similarly, if the algorithm doesn't look at every bit to the left of ℓ, the adversary could replace some unexamined bit with a zero. Finally, if there are any unexamined bits between ℓ and r, there must be at least two such bits (since $r - \ell$ is always even) and the adversary can put a 01 in the gap.

In general, we say that a bit pattern is *evasive* if we have to look at every bit to decide if a string of n bits contains the pattern. So the pattern 1 is evasive for all n, and the pattern 01 is evasive if and only if n is even. It turns out that the *only* patterns that are evasive for *all* values of n are the one-bit patterns 0 and 1.

13.4 Evasive Graph Properties

Another class of problems for which adversary arguments give good lower bounds is graph problems where the graph is represented by an adjacency matrix, rather than an adjacency list. Recall that the adjacency matrix of an undirected n-vertex graph $G = (V, E)$ is an $n \times n$ matrix A, where $A[i, j] = \big[(i, j) \in E\big]$. We are interested in deciding whether an undirected graph has or does not have a certain *property*. For example, is the input graph connected? Acyclic? Planar? Complete? A tree? We call a graph property *evasive* if we have to look look at all $\binom{n}{2}$ entries in the adjacency matrix to decide whether a graph has that property.

An obvious example of an evasive graph property is *emptiness*: Does the graph have any edges at all? We can show that emptiness is evasive using the following simple adversary strategy. The adversary maintains *two* graphs E and G. E is just the empty graph with n vertices. Initially G is the complete graph on n vertices. Whenever the algorithm asks about an edge, the adversary removes that edge from G (unless it's already gone) and answers 'no'. If the algorithm terminates without examining every edge, then G is not empty. Since both G and E are consistent with all the adversary's answers, the algorithm must give the wrong answer for one of the two graphs.

13.5 Connectedness Is Evasive

Now let me give a more complicated example, *connectedness*. Once again, the adversary maintains two graphs, Y and M ('yes' and 'maybe'). Y contains all the edges that the algorithm knows are definitely in the input graph. M contains all the edges that the algorithm thinks *might* be in the input graph, or in other words, all the edges of Y plus all the unexamined edges. Initially, Y is empty and M is complete.

Here's the strategy that adversary follows when the adversary asks whether the input graph contains the edge e. I'll assume that whenever an algorithm examines an edge, it's in M but not in Y; in other words, algorithms never ask about the same edge more than once.

```
HideConnectedness(e):
  if M \ {e} is connected
      remove (i, j) from M
      return 0
  else
      add e to Y
      return 1
```

Notice that the graphs Y and M are both consistent with the adversary's answers at all times. The adversary strategy maintains a few other simple invariants.

- **Y is a subgraph of M.** This is obvious.
- **M is connected.** This is also obvious.
- **If M has a cycle, none of its edges are in Y.** If M has a cycle, then deleting any edge in that cycle leaves M connected.
- **Y is acyclic.** This follows directly from the previous invariant.
- **If $Y \neq M$, then Y is disconnected.** The only connected acyclic graph is a tree. Suppose Y is a tree and some edge e is in M but not in Y. Then there is a cycle in M that contains e, all of whose other edges are in Y. This violated our third invariant.

We can also think about the adversary strategy in terms of minimum spanning trees. Recall the anti-Kruskal algorithm for computing the *maximum* spanning tree of a graph: Consider the edges one at a time in increasing order of length. If removing an edge would disconnect the graph, declare it part of the spanning tree (by adding it to Y); otherwise, throw it away (by removing it from M). If the algorithm examines all $\binom{n}{2}$ possible edges, then Y and M are both equal to the maximum spanning tree of the complete n-vertex graph, where the weight of an edge is the time when the algorithm asked about it.

Now, if an algorithm terminates before examining all $\binom{n}{2}$ edges, then there is at least one edge in M that is not in Y. Since the algorithm cannot distinguish between M and Y, even though M is connected and Y is not, the algorithm cannot possibly give the correct output for both graphs. Thus, in order to be correct, any algorithm must examine every edge—*Connectedness is evasive!*

13.6 An Evasive Conjecture

A graph property is *nontrivial* is there is at least one graph with the property and at least one graph without the property. (The only trivial properties are 'Yes' and 'No'.) A graph property is *monotone* if it is closed under taking subgraphs — if G has the property, then any subgraph of G has the property. For example, emptiness, planarity, acyclicity, and *non*-connectedness are monotone. The properties of being a tree and of having a vertex of degree 3 are not monotone.

Conjecture 1 (Aanderraa, Karp, and Rosenberg). *Every nontrivial monotone property of n-vertex graphs is evasive.*

The Aanderraa-Karp-Rosenberg conjecture has been proven when $n = p^e$ for some prime p and positive integer exponent e—the proof uses some interesting results from algebraic topology[2]—but it is still open for other values of n.

[2]Let Δ be a contractible simplicial complex whose automorphism group $\mathrm{Aut}(\Delta)$ is vertex-transitive, and let Γ be a vertex-transitive subgroup of $\mathrm{Aut}(\Delta)$. If there are normal subgroups $\Gamma_1 \lhd \Gamma_2 \lhd \Gamma$ such that $|\Gamma_1| = p^\alpha$ for some prime p and integer α, $|\Gamma/\Gamma_2| = q^\beta$ for some prime q and integer β, and Γ_2/Γ_1 is cyclic, then Δ is a simplex.

No, this will not be on the final exam.

There are non-trivial non-evasive graph properties, but all known examples are non-monotone. One such property—'scorpionhood'—is described in an exercise at the end of this lecture note.

13.7 Finding the Minimum and Maximum

Last time, we saw an adversary argument that finding the largest element of an unsorted set of n numbers requires at least $n-1$ comparisons. Let's consider the complexity of finding the largest *and* smallest elements. More formally:

> Given a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$ of n distinct numbers, find indices i and j such that $x_i = \min X$ and $x_j = \max X$.

How many comparisons do we need to solve this problem? An upper bound of $2n-3$ is easy: find the minimum in $n-1$ comparisons, and then find the maximum of everything else in $n-2$ comparisons. Similarly, a lower bound of $n-1$ is easy, since any algorithm that finds the min and the max certainly finds the max.

We can improve both the upper and the lower bound to $\lceil 3n/2 \rceil - 2$. The upper bound is established by the following algorithm. Compare all $\lfloor n/2 \rfloor$ consecutive pairs of elements x_{2i-1} and x_{2i}, and put the smaller element into a set S and the larger element into a set L. if n is odd, put x_n into both L and S. Then find the smallest element of S and the largest element of L. The total number of comparisons is at most

$$\underbrace{\left\lfloor \frac{n}{2} \right\rfloor}_{\text{build } S \text{ and } L} + \underbrace{\left\lceil \frac{n}{2} \right\rceil - 1}_{\text{compute } \min S} + \underbrace{\left\lceil \frac{n}{2} \right\rceil - 1}_{\text{compute } \max L} = \left\lceil \frac{3n}{2} \right\rceil - 2.$$

For the lower bound, we use an adversary argument. The adversary marks each element $+$ if it *might* be the maximum element, and $-$ if it *might* be the minimum element. Initially, the adversary puts both marks $+$ and $-$ on every element. If the algorithm compares two double-marked elements, then the adversary declares one smaller, removes the $+$ mark from the smaller element, and removes the $-$ mark from the larger one. In every other case, the adversary can answer so that at most one mark needs to be removed. For example, if the algorithm compares a double marked element against one labeled $-$, the adversary says the one labeled $-$ is smaller and removes the $-$ mark from the other. If the algorithm compares to $+$'s, the adversary must unmark one of the two.

Initially, there are $2n$ marks. At the end, in order to be correct, exactly one item must be marked $+$ and exactly one other must be marked $-$, since the adversary can make any $+$ the maximum and any $-$ the minimum. Thus, the algorithm must force the adversary to remove $2n-2$ marks. At most $\lfloor n/2 \rfloor$ comparisons remove two marks; every other comparison removes at most one mark. Thus, the adversary strategy forces any algorithm to perform at least $2n - 2 - \lfloor n/2 \rfloor = \lceil 3n/2 \rceil - 2$ comparisons.

13.8 Finding the Median

Finally, let's consider the *median* problem: Given an unsorted array X of n numbers, find its $n/2$th largest entry. (I'll assume that n is even to eliminate pesky floors and ceilings.) More formally:

> Given a sequence $\langle x_1, x_2, \ldots, x_n \rangle$ of n distinct numbers, find the index m such that x_m is the $n/2$th largest element in the sequence.

To prove a lower bound for this problem, we can use a combination of information theory and two adversary arguments. We use one adversary argument to prove the following simple lemma:

Lemma 1. *Any comparison tree that correctly finds the median element also identifies the elements smaller than the median and larger than the median.*

Proof: Suppose we reach a leaf of a decision tree that chooses the median element x_m, and there is still some element x_i that isn't known to be larger or smaller than x_m. In other words, we cannot decide based on the comparisons that we've already performed whether $x_i < x_m$ or $x_i > x_m$. Then in particular no element is known to lie between x_i and x_m. This means that there must be an input that is consistent with the comparisons we've performed, in which x_i and x_m are adjacent in sorted order. But then we can swap x_i and x_m, without changing the result of any comparison, and obtain a different consistent input in which x_i is the median, not x_m. Our decision tree gives the wrong answer for this 'swapped' input. □

This lemma lets us rephrase the median-finding problem yet again.

> Given a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$ of n distinct numbers, find the indices of its $n/2-1$ largest elements L and its $n/2$th largest element x_m.

Now suppose a 'little birdie' tells us the set L of elements larger than the median. This information fixes the outcomes of certain comparisons—any item in L is bigger than any element not in L—so we can 'prune' those comparisons from the comparison tree. The pruned tree finds the largest element of $X \setminus L$ (the median of X), and thus must have depth at least $n/2-1$. In fact, the adversary argument in the last lecture implies that *every* leaf in the pruned tree must have depth at least $n/2-1$, so the pruned tree has at least $2^{n/2-1}$ leaves.

There are $\binom{n}{n/2-1} \approx 2^n/\sqrt{n/2}$ possible choices for the set L. Every leaf in the original comparison tree is also a leaf in exactly one of the $\binom{n}{n/2-1}$ pruned trees, so the original comparison tree must have at least $\binom{n}{n/2-1}2^{n/2-1} \approx 2^{3n/2}/\sqrt{n/2}$ leaves. Thus, any comparison tree that finds the median must have depth at least

$$\left\lceil \frac{n}{2} - 1 + \lg\binom{n}{n/2-1} \right\rceil = \frac{3n}{2} - O(\log n).$$

A more complicated adversary argument (also involving pruning the comparison tree with little birdies) improves this lower bound to $2n - o(n)$.

A similar argument implies that at least $n-k+\left\lceil \lg\binom{n}{k-1} \right\rceil = \Omega((n-k)+k\log(n/k))$ comparisons are required to find the kth largest element in an n-element set. This bound is tight up to constant factors for all $k \le n/2$; there is an algorithm that uses at most $O(n + k\log(n/k))$ comparisons. Moreover, this lower bound is *exactly* tight when $k = 1$ or $k = 2$. In fact, these are the *only* values of $k \le n/2$ for which the exact complexity of the selection problem is known. Even the case $k = 3$ is still open!

Exercises

1. (a) Let X be a set containing an odd number of n-bit strings. Prove that any algorithm that decides whether a given n-bit string is an element of X *must* examine every bit of the input string in the worst case.

   (b) Give a one-line proof that the bit pattern 01 is evasive for all even n.

(c) Prove that the bit pattern 11 is evasive if and only if $n \bmod 3 = 1$.

*(d) Prove that the bit pattern 111 is evasive if and only if $n \bmod 4 = 0$ or 3.

2. Suppose we are given the adjacency matrix of a *directed* graph G with n vertices. Describe an algorithm that determines whether G has a *sink* by probing only $O(n)$ bits in the input matrix. A sink is a vertex that has an incoming edge from every other vertex, but no outgoing edges.

*3. A *scorpion* is an undirected graph with three special vertices: the *sting*, the *tail*, and the *body*. The sting is connected only to the tail; the tail is connected only to the sting and the body; and the body is connected to every vertex except the sting. The rest of the vertices (the head, eyes, legs, antennae, teeth, gills, flippers, wheels, etc.) can be connected arbitrarily. Describe an algorithm that determines whether a given n-vertex graph is a scorpion by probing only $O(n)$ entries in the adjacency matrix.

4. Prove using an adversary argument that acyclicity is an evasive graph property. *[Hint: Kruskal.]*

5. Prove that finding the second largest element in an n-element array requires *exactly* $n - 2 + \lceil \lg n \rceil$ comparisons in the worst case. Prove the upper bound by describing and analyzing an algorithm; prove the lower bound using an adversary argument.

6. Let T be a perfect ternary tree where every leaf has depth ℓ. Suppose each of the 3^ℓ leaves of T is labeled with a bit, either 0 or 1, and each internal node is labeled with a bit that agrees with the *majority* of its children.

   (a) Prove that any deterministic algorithm that determines the label of the root must examine all 3^ℓ leaf bits in the worst case.

   (b) Describe and analyze a *randomized* algorithm that determines the root label, such that the expected number of leaves examined is $o(3^\ell)$. (You may want to review the notes on randomized algorithms.)

*7. UIUC has just finished constructing the new Reingold Building, the tallest dormitory on campus. In order to determine how much insurance to buy, the university administration needs to determine the highest safe floor in the building. A floor is consdered *safe* if ~~a drunk student~~ ***an egg*** can fall from a window on that floor and land without breaking; if the egg breaks, the floor is considered *unsafe*. Any floor that is higher than an unsafe floor is also considered unsafe. The only way to determine whether a floor is safe is to drop an egg from a window on that floor.

   You would like to find the lowest unsafe floor L by performing as few tests as possible; unfortunately, you have only a very limited supply of eggs.

   (a) Prove that if you have only one egg, you can find the lowest unsafe floor with L tests. *[Hint: Yes, this is trivial.]*

(b) Prove that if you have only one egg, you must perform at least L tests in the worst case. In other words, prove that your algorithm from part (a) is optimal. *[Hint: Use an adversary argument.]*

(c) Describe an algorithm to find the lowest unsafe floor using *two* eggs and only $O(\sqrt{L})$ tests. *[Hint: Ideally, each egg should be dropped the same number of times. How many floors can you test with n drops?]*

(d) Prove that if you start with two eggs, you must perform at least $\Omega(\sqrt{L})$ tests in the worst case. In other words, prove that your algorithm from part (c) is optimal.

*(e) Describe an algorithm to find the lowest unsafe floor using k eggs, using as few tests as possible, and prove your algorithm is optimal for all values of k.

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms for the most recent revision.

Jeder Genießende meint, dem Baume habe es an der Frucht gelegen;
aber ihm lag am Samen.
[Everyone who enjoys thinks that the fundamental thing about trees is the fruit,
but in fact it is the seed.]

— Friedrich Wilhelm Nietzsche,
Vermischte Meinungen und Sprüche [Mixed Opinions and Maxims] (1879)

In view of all the deadly computer viruses that have been spreading lately,
Weekend Update would like to remind you:
When you link up to another computer,
you're linking up to every computer that that computer has ever linked up to.

— Dennis Miller, "Saturday Night Live", (c. 1985)

Anything that, in happening, causes itself to happen again, happens again.

— Douglas Adams (2005)

The Curling Stone slides; and, having slid,
Passes me toward thee on this Icy Grid,
If what's reached is passed for'll Crystals amid,
Th'Stone Reaches thee in its Eternal Skid.

— Iraj Kalantari (2007)
writing as "Harak A'Myomy (12th century),
translated by Walt Friz De Gradde (1897)"

Proof by Induction

Induction is a method for proving universally quantified propositions—statements about *all* elements of a (usually infinite) set. Induction is also the single most useful tool for reasoning about, developing, and analyzing algorithms. These notes give several examples of inductive proofs, along with a standard boilerplate and some motivation to justify (and help you remember) why induction works.

1 Prime Divisors: Proof by Smallest Counterexample

A ***divisor*** of a positive integer n is a positive integer p such that the ratio n/p is an integer. The integer 1 is a divisor of every positive integer (because $n/1 = n$), and every integer is a divisor of itself (because $n/n = 1$). A ***proper divisor*** of n is any divisor of n other than n itself. A positive integer is ***prime*** if it has *exactly two* divisors, which must be 1 and itself; equivalently; a number is prime if and only if 1 is its only proper divisor. A positive integer is ***composite*** if it has more than two divisors (or equivalently, more than one proper divisor). The integer 1 is neither prime nor composite, because it has exactly one divisor, namely itself.

Let's prove our first theorem:

Theorem 1. *Every integer greater than* 1 *has a prime divisor.*

The very first thing that you should notice, after reading just one word of the theorem, is that this theorem is *universally quantified*—it's a statement about *all* the elements of a set, namely, the set of positive integers larger than 1. If we were forced at gunpoint to write this sentence

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

using fancy logic notation, the first character would be the universal quantifier $\forall$, pronounced 'for all'. Fortunately, that won't be necessary.

There are only two ways to prove a universally quantified statement: directly or by contradiction. Let's say that again, louder: ***There are only two ways to prove a universally quantified statement: directly or by contradiction.*** Here are the standard templates for these two methods, applied to Theorem 1:

> **Direct proof:** Let n be an arbitrary integer greater than 1.
>
> *. . . blah blah blah . . .*
>
> Thus, n has at least one prime divisor. □

> **Proof by contradiction:** For the sake of argument, assume there is an integer greater than 1 with no prime divisor.
> Let n be an arbitrary integer greater than 1 with no prime divisor.
>
> *. . . blah blah blah . . .*
>
> But that's just silly. Our assumption must be incorrect. □

The shaded boxes *. . . blah blah blah . . .* indicate missing proof details (that you will fill in).

Most people usually find proofs by contradiction easier to discover than direct proofs, so let's try that first.

> **Proof by contradiction:** For the sake of argument, assume there is an integer greater than 1 with no prime divisor.
>
> Let n be an arbitrary integer greater than 1 with no prime divisor.
>
> Since n is a divisor of n, and n has no prime divisors, n cannot be prime.
>
> Thus, n must have at least one divisor d such that $1 < d < n$.
>
> Let d be an arbitrary divisor of n such that $1 < d < n$.
>
> Since n has no prime divisors, d cannot be prime.
>
> Thus, d has at least one divisor d' such that $1 < d' < d$.
>
> Let d' be an arbitrary divisor of d such that $1 < d' < d$.
>
> Because d/d' is an integer, $n/d' = (n/d) \cdot (d/d')$ is also an integer.
>
> Thus, d' is also a divisor of n.
>
> Since n has no prime divisors, d' cannot be prime.
>
> Thus, d' has at least one divisor d_{J} such that $1 < d_{\text{J}} < d'$.
>
> Let d_{J} be an arbitrary divisor of d' such that $1 < d_{\text{J}} < d'$.
>
> Because d'/d_{J} is an integer, $n/d_{\text{J}} = (n/d') \cdot (d'/d_{\text{J}})$ is also an integer.
>
> Thus, d_{J} is also a divisor of n.
>
> Since n has no prime divisors, d_{J} cannot be prime.
>
> *. . . blah HELP! blah I'M STUCK IN AN INFINITE LOOP! blah . . .*
>
> But that's just silly. Our assumption must be incorrect. □

We seem to be stuck in an infinite loop, looking at smaller and smaller divisors $d > d' > d_{\text{J}} > \cdots$, none of which are prime. But this loop can't really be infinite. There are only $n-1$ positive integers smaller than n, so the proof *must* end after *at most* $n-1$ iterations. But how do we turn this observation into a formal proof? We need a single, self-contained proof for all integers n; we're not allowed to write longer proofs for bigger integers. The trick is to jump directly to the ***smallest*** counterexample.

Proof by smallest counterexample: For the sake of argument, assume that there is an integer greater than 1 with no prime divisor.

Let n be ***the smallest*** integer greater than 1 with no prime divisor.

Since n is a divisor of n, and n has no prime divisors, n cannot be prime.

Thus, n has a divisor d such that $1 < d < n$.

Let d be a divisor of n such that $1 < d < n$.

Because n is the smallest counterexample, d has a prime divisor.

Let p be a prime divisor of d.

Because d/p is an integer, $n/p = (n/d) \cdot (d/p)$ is also an integer.

Thus, p is also a divisor of n.

But this contradicts our assumption that n has no prime divisors!

So our assumption must be incorrect. □

Hooray, our first proof! We're done!

Um... well... no, we're definitely *not* done. That's a first draft up there, not a final polished proof. We don't write proofs just to convince ourselves; proofs are primarily a tool to convince other people. (In particular, 'other people' includes the people grading your homeworks and exams.) And while proofs by contradiction are usually easier to *write*, direct proofs are almost always easier to *read*. So as a service to our audience (and our grade), let's transform our minimal-counterexample proof into a direct proof.

Let's first rewrite the indirect proof slightly, to make the structure more apparent. First, we break the assumption that n is the smallest counterexample into three simpler assumptions: (1) n is an integer greater than 1; (2) n has no prime divisors; and (3) there are no smaller counterexamples. Second, instead of dismissing the possibility than n is prime out of hand, we include an explicit case analysis.

Proof by smallest counterexample: Let n be an arbitrary integer greater than 1.

For the sake of argument, suppose n has no prime divisor.

Assume that every integer k such that $1 < k < n$ has a prime divisor.

There are two cases to consider: Either n is prime, or n is composite.

- Suppose n is prime.

  Then n is a prime divisor of n.

- Suppose n is composite.

  Then n has a divisor d such that $1 < d < n$.

  Let d be a divisor of n such that $1 < d < n$.

  Because no counterexample is smaller than n, d has a prime divisor.

  Let p be a prime divisor of d.

  Because d/p is an integer, $n/p = (n/d) \cdot (d/p)$ is also an integer.

  Thus, p is a prime divisor of n.

In each case, we conclude that n has a prime divisor.

But this contradicts our assumption that n has no prime divisors!

So our assumption must be incorrect. □

Now let's look carefully at the structure of this proof. First, we assumed that the statement we want to prove is false. Second, we proved that the statement we want to prove is true. Finally, we concluded from the contradiction that our assumption that the statement we want to prove is false is incorrect, so the statement we want to prove must be true.

But that's just silly. Why do we need the first and third steps? After all, the second step is a proof all by itself! Unfortunately, this redundant style of proof by contradiction is *extremely* common, even in professional papers. Fortunately, it's also very easy to avoid; just remove the first and third steps!

Proof by induction: Let n be an arbitrary integer greater than 1.
Assume that every integer k such that $1 < k < n$ has a prime divisor.
There are two cases to consider: Either n is prime or n is composite.

- First, suppose n is prime.
  Then n is a prime divisor of n.
- Now suppose n is composite.
  Then n has a divisor d such that $1 < d < n$.
  Let d be a divisor of n such that $1 < d < n$.
  Because no counterexample is smaller than n, d has a prime divisor.
  Let p be a prime divisor of d.
  Because d/p is an integer, $n/p = (n/d) \cdot (d/p)$ is also an integer.
  Thus, p is a prime divisor of n.

In both cases, we conclude that n has a prime divisor. □

This style of proof is called ***induction***.[1] The assumption that there are no counterexamples smaller than n is called the ***induction hypothesis***. The two cases of the proof have different names. The first case, which we argue directly, is called the ***base case***. The second case, which actually uses the induction hypothesis, is called the ***inductive case***. You may find it helpful to actually label the induction hypothesis, the base case(s), and the inductive case(s) in your proof.

The following point cannot be emphasized enough: The only difference between a proof by induction and a proof by smallest counterexample is the way we write down the argument. The essential structure of the proofs are exactly the same. The core of our original indirect argument is a proof of the following implication for all n:

$$n \text{ has no prime divisor} \implies \text{some number smaller than } n \text{ has no prime divisor.}$$

The core of our direct proof is the following logically equivalent implication:

$$\text{every number smaller than } n \text{ has a prime divisor} \implies n \text{ has a prime divisor}$$

The left side of this implication is just the induction hypothesis.

The proofs we've been playing with have been very careful and explicit; until you're comfortable writing your own proofs, you should be equally careful. A more mature proof-writer might express the same proof more succinctly as follows:

Proof by induction: Let n be an arbitrary integer greater than 1. Assume that every integer k such that $1 < k < n$ has a prime divisor. If n is prime, then n is a prime divisor of n. On the other hand, if n is composite, then n has a proper divisor; call it d. The induction hypothesis implies that d has a prime divisor p. The integer p is also a divisor of n. □

A proof in this more succinct form is still worth full credit, provided the induction hypothesis is written explicitly and the case analysis is obviously exhaustive.

A professional mathematician would write the proof even more tersely:

Proof: Induction. □

And you can write that tersely, too, when you're a professional mathematician.

[1] Many authors use the high-falutin' name *the principle of mathematical induction*, to distinguish it from *inductive reasoning*, the informal process by which we conclude that pigs can't whistle, horses can't fly, and NP-hard problems cannot be solved in polynomial time. We already know that every proof is mathematical (and arguably, all mathematics is proof), so as a description of a *proof* technique, the adjective 'mathematical' is simply redundant.

2 The Axiom of Induction

Why does this work? Well, let's step back to the original proof by smallest counterexample. How do we know that a smallest counterexample exists? This seems rather obvious, but in fact, it's impossible to prove without using the following seemingly trivial observation, called the ***Well-Ordering Principle***:

> *Every non-empty set of positive integers has a smallest element.*

Every set X of positive integers is the set of counterexamples to some proposition $P(n)$ (specifically, the proposition $n \notin X$). Thus, the Well-Ordering Principle can be rewritten as follows:

> *If the proposition $P(n)$ is false for some positive integer n,*
> *then*
> *the proposition $(P(1) \wedge P(2) \wedge \cdots \wedge P(n-1) \wedge \neg P(n))$ is true for some positive integer n.*

Equivalently, in English:

> *If some statement about positive integers has a counterexample,*
> *then*
> *that statement has a smallest counterexample.*

We can write this implication in contrapositive form as follows:

> *If the proposition $(P(1) \wedge P(2) \wedge \cdots \wedge P(n-1) \wedge \neg P(n))$ is false for every positive integer n,*
> *then*
> *the proposition $P(n)$ is true for every positive integer n.*

or less formally,

> *If some statement about positive integers has no smallest counterexample,*
> *then*
> *that statement is true for all positive integers.*

Finally, let's rewrite the first half of this statement in a logically equivalent form, by replacing $\neg(p \wedge \neg q)$ with $p \to q$.

> *If the implication $(P(1) \wedge P(2) \wedge \cdots \wedge P(n-1)) \to P(n)$ is true for every positive integer n,*
> *then*
> *the proposition $P(n)$ is true for every positive integer n.*

This formulation is usually called the ***Axiom of Induction***. In a proof by induction that $P(n)$ holds for all n, the conjunction $(P(1) \wedge P(2) \wedge \cdots \wedge P(n-1))$ is the inductive hypothesis.

A ***proof by induction*** for the proposition "$P(n)$ for every positive integer n" is nothing but a direct proof of the more complex proposition "$(P(1) \wedge P(2) \wedge \cdots \wedge P(n-1)) \to P(n)$ for every positive integer n". Because it's a direct proof, it *must* start by considering an arbitrary positive integer, which we might as well call n. Then, to prove the implication, we explicitly assume the hypothesis $(P(1) \wedge P(2) \wedge \cdots \wedge P(n-1))$ and then prove the conclusion $P(n)$ *for that particular value of* n. The proof almost always breaks down into two or more cases, each of which may or may not actually use the inductive hypothesis.

Here is the boilerplate for *every* induction proof. Read it. Learn it. Use it.

Theorem: $P(n)$ for every positive integer n.

Proof by induction: Let n be an arbitrary positive integer.
Assume that $P(k)$ is true for every positive integer $k < n$.
There are several cases to consider:

- Suppose n is *. . . blah blah blah . . .*
  Then $P(n)$ is true.
- Suppose n is *. . . blah blah blah . . .*
  The inductive hypothesis implies that *. . . blah blah blah . . .*
  Thus, $P(n)$ is true.

In each case, we conclude that $P(n)$ is true. □

Or more generally:

Bellman's Theorem: Every snark is a boojum.

Proof by induction: Let X be an arbitrary snark.
Assume that for every snark younger than X is a boojum.
There are three cases to consider:

- Suppose X is the youngest snark.
  Then *. . . blah blah blah . . .*
- Suppose X is the second-youngest snark.
  Then *. . . blah blah blah . . .*
- Suppose X is older than the second-youngest snark.
  Then the inductive hypothesis implies *. . . blah blah blah . . .* and therefore *. . . blah blah blah . . .*

An all cases, we conclude that X is a boojum. □

Some textbooks distinguish between several different types of induction: 'regular' induction versus 'strong' induction versus 'complete' induction versus 'structural' induction versus 'transfinite' induction versus 'Noetherian' induction. Distinguishing between these different types of induction is pointless hairsplitting; I won't even define them. Every 'different type' of induction proof is provably equivalent to a proof by smallest counterexample. (Later we will consider inductive proofs of statements about partially ordered sets other than the positive integers, for which 'smallest' has a different meaning, but this difference will prove to be inconsequential.)

3 Stamps and Recursion

Let's move on to a completely different example.

Theorem 2. *Given an unlimited supply of 5-cent stamps and 7-cent stamps, we can make any amount of postage larger than 23 cents.*

We could prove this by contradiction, using a smallest-counterexample argument, but let's aim for a direct proof by induction this time. We start by writing down the induction boilerplate, using the standard induction hypothesis: *There is no counterexample smaller than n.*

> **Proof by induction:** Let n be an arbitrary integer greater than 23.
> Assume that for any integer k such that $23 < k < n$, we can make k cents in postage.
> *. . . blah blah blah . . .*
> Thus, we can make n cents in postage. □

How do we fill in the details? One approach is to think about what you would actually do if you really had to make n cents in postage. For example, you might start with a 5-cent stamp, and then try to make $n-5$ cents in postage. The inductive hypothesis says you can make *any* amount of postage bigger than 23 cents and less than n cents. So if $n-5 > 23$, then *you already know* that you can make $n-5$ cents in postage! (You don't know *how* to make $n-5$ cents in postage, but so what?)

Let's write this observation into our proof as two separate cases: either $n > 28$ (where our approach works) or $n \leq 28$ (where we don't know what to do yet).

> **Proof by induction:** Let n be an arbitrary integer greater than 23.
> Assume that for any integer k such that $23 < k < n$, we can make k cents in postage.
> There are two cases to consider: Either $n > 28$ or $n \leq 28$.
> - Suppose $n > 28$.
> Then $23 < n-5 < n$.
> Thus, ***the induction hypothesis*** implies that we can make $n-5$ cents in postage.
> Adding one more 5-cent stamp gives us n cents in postage.
> - Now suppose $n \leq 28$.
> *. . . blah blah blah . . .*
>
> In both cases, we can make n cents in postage. □

What do we do in the second case? Fortunately, this case considers only five integers: 24, 25, 26, 27, and 28. There might be a clever way to solve all five cases at once, but why bother? They're small enough that we can find a solution by brute force in less than a minute. To make the proof more readable, I'll unfold the nested cases and list them in increasing order.

> **Proof by induction:** Let n be an arbitrary integer greater than 23.
> Assume that for any integer k such that $23 < k < n$, we can make k cents in postage.
> There are six cases to consider: $n = 24$, $n = 25$, $n = 26$, $n = 27$, $n = 28$, and $n > 28$.
> - $24 = 7+7+5+5$
> - $25 = 5+5+5+5+5$
> - $26 = 7+7+7+5$
> - $27 = 7+5+5+5+5$
> - $28 = 7+7+7+7$
> - Suppose $n > 28$.
> Then $23 < n-5 < n$.
> Thus, ***the induction hypothesis*** implies that we can make $n-5$ cents in postage.
> Adding one more 5-cent stamp gives us n cents in postage.
>
> In all cases, we can make n cents in postage. □

Voilà! An induction proof! More importantly, we now have a recipe for *discovering* induction proofs.

1. ***Write down the boilerplate.*** Write down the universal invocation ('Let n be an arbitrary. . . '), the induction hypothesis, and the conclusion, with enough blank space for the remaining details. Don't be clever. Don't even think. Just write. ***This is the easy part.*** To emphasize the common structure, the boilerplate will be indicated in green for the rest of this handout.

2. ***Think big.*** *Don't* think how to solve the problem all the way down to the ground; you'll only make yourself dizzy. *Don't* think about piddly little numbers like 1 or 5 or 10^{100}. Instead, think about how to reduce the proof about some *absfoluckingutely ginormous* value of n to a proof about some other number(s) smaller than n. ***This is the hard part.***

3. ***Look for holes.*** Look for cases where your inductive argument breaks down. Solve those cases directly. Don't be clever here; be stupid but thorough.

4. ***Rewrite everything.*** Your first proof is a rough draft. Rewrite the proof so that your argument is easier for your (unknown?) reader to follow.

The cases in an inductive proof always fall into two categories. Any case that uses the inductive hypothesis is called an *inductive case*. Any case that does not use the inductive hypothesis is called a *base case*. Typically, but *not* always, base cases consider a few small values of n, and the inductive cases consider everything else. Induction proofs are usually clearer if we present the base cases first, but I find it much easier to *discover* the inductive cases first. In other words, I recommend writing induction proofs backwards.

Well-written induction proofs *very* closely resemble well-written recursive programs. We computer scientists use induction primarily to reason about recursion, so maintaining this resemblance is extremely useful—we only have to keep one mental pattern, called 'induction' when we're writing proofs and 'recursion' when we're writing code. Consider the following C and Scheme programs for making n cents in postage:

```
void postage(int n)
{
    assert(n>23);
    switch ($n$)
    {
        case 24:  printf("7+7+5+5");    break;
        case 25:  printf("5+5+5+5+5");  break;
        case 26:  printf("7+7+7+5");    break;
        case 27:  printf("7+5+5+5+5");  break;
        case 28:  printf("7+7+7+7");    break;
        default:
            postage(n-5);
            printf("+5");
    }
}
```

```
(define (postage n)
  (cond ((= n 24) (5 5 7 7))
        ((= n 25) (5 5 5 5 5))
        ((= n 26) (5 7 7 7))
        ((= n 27) (5 5 5 5 7))
        ((= n 28) (7 7 7 7))
        ((> n 28) (cons 5 (postage (- n 5))))))
```

The C program begins by declaring the input parameter ("Let n be an arbitrary integer. . . ") and asserting its range (". . . greater than 23."). (Scheme programs don't have type declarations.) In both languages, the code branches into six cases: five that are solved directly, plus one that is handled ~~by invoking the inductive hypothesis~~ recursively.

4 More on Prime Divisors

Before we move on to different examples, let's prove another fact about prime numbers:

Theorem 3. *Every positive integer is a product of prime numbers.*

First, let's write down the boilerplate. Hey! I saw that! You were *thinking*, weren't you? Stop that this instant! Don't make me turn the car around. ***First*** we write down the boilerplate.

> **Proof by induction:** Let n be an arbitrary positive integer.
>
> Assume that any positive integer $k < n$ is a product of prime numbers.
>
> There are *some* cases to consider:
>
> *. . . blah blah blah . . .*
>
> Thus, n is a product of prime numbers. □

Now let's think about how you would actually factor a positive integer n into primes. There are a couple of different options here. One possibility is to find a prime divisor p of n, as guaranteed by Theorem 1, and recursively factor the integer n/p. This argument works as long as $n \geq 2$, but what about $n = 1$? The answer is simple: 1 *is the product of the* ***empty*** *set of primes*. What else could it be?

> **Proof by induction:** Let n be an arbitrary positive integer.
>
> Assume that any positive integer $k < n$ is a product of prime numbers.
>
> There are two cases to consider: either $n = 1$ or $n \geq 2$.
>
> - If $n = 1$, then n is the product of the elements of the empty set, each of which is prime, green, sparkly, vanilla, and hemophagic.
> - Suppose $n > 1$. Let p be a prime divisor of n, as guaranteed by Theorem 2. The inductive hypothesis implies that the positive integer n/p is a product of primes, and clearly $n = (n/p) \cdot p$.
>
> In both cases, n is a product of prime numbers. □

But an even simpler method is to factor n into any two proper divisors, and recursively handle them both. This method works as long as n is composite, since otherwise there is no way to factor n into smaller integers. Thus, we need to consider prime numbers separately, as well as the special case 1.

Proof by induction: Let n be an arbitrary positive integer.

Assume that any positive integer $k < n$ is a product of prime numbers.

There are three cases to consider: either $n = 1$, n is prime, or n is composite.

- If $n = 1$, then n is the product of the elements of the empty set, each of which is prime, red, broody, chocolate, and lycanthropic.
- If n is prime, then n is the product of one prime number, namely n.
- Suppose n is composite. Let d be any proper divisor of n (guaranteed by the definition of 'composite'), and let $m = n/d$. Since both d and m are positive integers smaller than n, the inductive hypothesis implies that d and m are both products of prime numbers. We clearly have $n = d \cdot m$.

In both cases, n is a product of prime numbers. □

5 Summations

Here's an easy one.

Theorem 4. $\displaystyle\sum_{i=0}^{n} 3^i = \frac{3^{n+1}-1}{2}$ *for every non-negative integer* n.

First let's write down the induction boilerplate, which empty space for details we'll fill in later.

Proof by induction: Let n be an arbitrary non-negative integer.

Assume inductively that $\displaystyle\sum_{i=0}^{k} 3^i = \frac{3^{k+1}-1}{2}$ for every non-negative integer $k < n$.

There are *some number of* cases to consider:

. . . blah blah blah . . .

We conclude that $\displaystyle\sum_{i=0}^{n} 3^i = \frac{3^{n+1}-1}{2}$. □

Now imagine you are part of an infinitely long assembly line of mathematical provers, each assigned to a particular non-negative integer. Your task is to prove this theorem for the integer 8675310. The regulations of the Mathematical Provers Union require you not to think about any other integer but your own. The assembly line starts with the Senior Master Prover, who proves the theorem for the case $n = 0$. Next is the Assistant Senior Master Prover, who proves the theorem for $n = 1$. After him is the Assistant Assistant Senior Master Prover, who proves the theorem for $n = 2$. Then the Assistant Assistant Assistant Senior Master Prover proves the theorem for $n = 3$. As the work proceeds, you start to get more and more bored. You attempt strike up a conversation with Jenny, the prover to your left, but she ignores you, preferring to focus on the proof. Eventually, you fall into a deep, dreamless sleep. An undetermined time later, Jenny wakes you up by shouting, "Hey, doofus! It's your turn!" As you look around, bleary-eyed, you realize that Jenny and everyone to your left has finished their proofs, and that everyone is waiting for you to finish yours. What do you do?

What you do, after wiping the drool off your chin, is stop and think for a moment about what you're trying to prove. What does that $\sum$ notation actually mean? Intuitively, we can expand the

notation as follows:

$$\sum_{i=0}^{8675310} 3^i = 3^0 + 3^1 + \cdots + 3^{8675309} + 3^{8675310}.$$

Notice that this expression also contains the summation that Jenny just finished proving something about:

$$\sum_{i=0}^{8675309} 3^i = 3^0 + 3^1 + \cdots + 3^{8675308} + 3^{8675309}.$$

Putting these two expressions together gives us the following identity:

$$\sum_{i=0}^{8675310} 3^i = \sum_{i=0}^{8675309} 3^i + 3^{8675310}$$

In fact, this recursive identity is the *definition* of $\sum$. Jenny just proved that the summation on the right is equal to $(3^{8675310} - 1)/2$, so we can plug that into the right side of our equation:

$$\sum_{i=0}^{8675310} 3^i = \sum_{i=0}^{8675309} 3^i + 3^{8675310} = \frac{3^{8675310} - 1}{2} + 3^{8675310}.$$

And it's all downhill from here. After a little bit of algebra, you simplify the right side of this equation to $(3^{8675311} - 1)/2$, wake up the prover to your right, and start planning your well-earned vacation.

Let's insert this argument into our boilerplate, only using a generic 'big' integer n instead of the specific integer 8675310:

Proof by induction: Let n be an arbitrary non-negative integer.

Assume inductively that $\displaystyle\sum_{i=0}^{k} 3^i = \frac{3^{k+1} - 1}{2}$ for every non-negative integer $k < n$.

There are two cases to consider: Either n *is big* or n *is small*.

- If n *is big*, then

$$\begin{aligned}
\sum_{i=0}^{n} 3^i &= \sum_{i=0}^{n-1} 3^i + 3^n && \text{[definition of } \textstyle\sum\text{]} \\
&= \frac{3^n - 1}{2} + 3^n && \text{[induction hypothesis, with } k = n-1\text{]} \\
&= \frac{3^{n+1} - 1}{2} && \text{[algebra]}
\end{aligned}$$

- On the other hand, if n *is small*, then *. . . blah blah blah . . .*

In both cases, we conclude that $\displaystyle\sum_{i=0}^{n} 3^i = \frac{3^{n+1} - 1}{2}$. □

Now, how big is 'big', and what do we do when n is 'small'? To answer the first question, let's look at where our existing inductive argument breaks down. In order to apply the induction hypothesis when $k = n - 1$, the integer $n - 1$ must be non-negative; equivalently, n must be *positive*. But that's the only assumption we need: ***The only case we missed is $n = 0$.*** Fortunately, this case is easy to handle directly.

Proof by induction: Let n be an arbitrary non-negative integer.

Assume inductively that $\sum_{i=0}^{k} 3^i = \frac{3^{k+1}-1}{2}$ for every non-negative integer $k < n$.

There are two cases to consider: Either $n = 0$ or $n \geq 1$.

- If $n = 0$, then $\sum_{i=0}^{n} 3^i = 3^0 = 1$, and $\frac{3^{n+1}-1}{2} = \frac{3^1-1}{2} = 1$.
- On the other hand, if $n \geq 1$, then

$$\begin{aligned}
\sum_{i=0}^{n} 3^i &= \sum_{i=0}^{n-1} 3^i + 3^n && \text{[definition of } \textstyle\sum\text{]} \\
&= \frac{3^n-1}{2} + 3^n && \text{[induction hypothesis, with } k = n-1\text{]} \\
&= \frac{3^{n+1}-1}{2} && \text{[algebra]}
\end{aligned}$$

In both cases, we conclude that $\sum_{i=0}^{n} 3^i = \frac{3^{n+1}-1}{2}$. □

Here is the same proof, written more tersely; the non-standard symbol $\overset{IH}{=\!=}$ indicates the use of the induction hypothesis.

Proof by induction: Let n be an arbitrary non-negative integer, and assume inductively that $\sum_{i=0}^{k} 3^i = (3^{k+1}-1)/2$ for every non-negative integer $k < n$. The base case $n = 0$ is trivial, and for any $n \geq 1$, we have

$$\sum_{i=0}^{n} 3^i \;=\; \sum_{i=0}^{n-1} 3^i + 3^n \;\overset{IH}{=\!=}\; \frac{3^n-1}{2} + 3^n \;=\; \frac{3^{n+1}-1}{2}.$$

□

This is not the only way to prove this theorem by induction; here is another:

Proof by induction: Let n be an arbitrary non-negative integer, and assume inductively that $\sum_{i=0}^{k} 3^i = (3^{k+1}-1)/2$ for every non-negative integer $k < n$. The base case $n = 0$ is trivial, and for any $n \geq 1$, we have

$$\sum_{i=0}^{n} 3^i \;=\; 3^0 + \sum_{i=1}^{n} 3^i \;=\; 3^0 + 3 \cdot \sum_{i=0}^{n-1} 3^i \;\overset{IH}{=\!=}\; 3^0 + 3 \cdot \frac{3^n-1}{2} \;=\; \frac{3^{n+1}-1}{2}.$$

□

In the remainder of these notes, I'll give several more examples of induction proofs. In some cases, I give multiple proofs for the same theorem. Unlike the earlier examples, I will not describe the thought process that lead to the proof; in each case, I followed the basic outline on page 7.

6 Tiling with Triominos

The next theorem is about *tiling* a square checkerboard with *triominos*. A triomino is a shape composed of three squares meeting in an L-shape. Our goal is to cover as much of a $2^n \times 2^n$ grid

with triominos as possible, without any two triominos overlapping, and with all triominos inside the square. We can't cover every square in the grid—the number of squares is 4^n, which is not a multiple of 3—but we can cover all but one square. In fact, as the next theorem shows, we can choose *any* square to be the one we don't want to cover.

Almost tiling a 16×16 checkerboard with triominos.

Theorem 5. *For any non-negative integer n, the $2^n \times 2^n$ checkerboard with any square removed can be tiled using L-shaped triominos.*

Here are two inductive proofs for this theorem, one 'top down', the other 'bottom up'.

Proof by top-down induction: Let n be an arbitrary non-negative integer. Assume that for any non-negative integer $k < n$, the $2^k \times 2^k$ grid with any square removed can be tiled using triominos. There are two cases to consider: Either $n = 0$ or $n \geq 1$.

- The $2^0 \times 2^0$ grid has a single square, so removing one square leaves nothing, which we can tile with zero triominos.
- Suppose $n \geq 1$. In this case, the $2^n \times 2^n$ grid can be divided into four smaller $2^{n-1} \times 2^{n-1}$ grids. Without loss of generality, suppose the deleted square is in the upper right quarter. With a single L-shaped triomino at the center of the board, we can cover one square in each of the other three quadrants. The induction hypothesis implies that we can tile each of the quadrants, minus one square.

In both cases, we conclude that the $2^n \times 2^n$ grid with any square removed can be tiled with triominos. □

Top-down inductive proof of Theorem 4.

Proof by bottom-up induction: Let n be an arbitrary non-negative integer. Assume that for any non-negative integer $k < n$, the $2^k \times 2^k$ grid with any square removed can be tiled using triominos. There are two cases to consider: Either $n = 0$ or $n \geq 1$.

- The $2^0 \times 2^0$ grid has a single square, so removing one square leaves nothing, which we can tile with zero triominos.
- Suppose $n \geq 1$. Then by clustering the squares into 2×2 *blocks*, we can transform any $2^n \times 2^n$ grid into a $2^{n-1} \times 2^{n-1}$ grid. Suppose square (i, j) has been removed from the $2^n \times 2^n$ grid. The induction hypothesis implies that the $2^{n-1} \times 2^{n-1}$ grid with block $(\lfloor i/2 \rfloor, \lfloor j/2 \rfloor)$ removed can be tiled with double-size triominos. Each double-size triomono can be tiled with four smaller triominos, and block $(\lfloor i/2 \rfloor, \lfloor j/2 \rfloor)$ with square (i, j) removed is another triomino.

In both cases, we conclude that the $2^n \times 2^n$ grid with any square removed can be tiled with triominos. □

Second proof of Theorem 4.

7 Binary Numbers Exist

Theorem 6. *Every non-negative integer can be written as the sum of distinct powers of* 2.

Intuitively, this theorem states that every number can be represented in binary. (That's not a proof, by the way; it's just a restatement of the theorem.) I'll present *four* distinct inductive proofs for this theorem. The first two are standard, by-the-book induction proofs.

Proof by top-down induction: Let n be an arbitrary non-negative integer. Assume that any non-negative integer less than n can be written as the sum of distinct powers of 2. There are two cases to consider: Either $n = 0$ or $n \geq 1$.

- The base case $n = 0$ is trivial—the elements of the empty set are distinct and sum to zero.
- Suppose $n \geq 1$. Let k be the largest integer such that $2^k \leq n$, and let $m = n - 2^k$. Observe that $m < 2^{k+1} - 2^k = 2^k$. Because $0 \leq m < n$, the inductive hypothesis implies that m can be written as the sum of distinct powers of 2. Moreover, in the summation for m, each power of 2 is at most m, and therefore less than 2^k. Thus, $m + 2^k$ is the sum of distinct powers of 2.

In either case, we conclude that n can be written as the sum of distinct powers of 2. □

Proof by bottom-up induction: Let n be an arbitrary non-negative integer. Assume that any non-negative integer less than n can be written as the sum of distinct powers of 2. There are two cases to consider: Either $n = 0$ or $n \geq 1$.

- The base case $n = 0$ is trivial—the elements of the empty set are distinct and sum to zero.
- Suppose $n \geq 1$, and let $m = \lfloor n/2 \rfloor$. Because $0 \leq m < n$, the inductive hypothesis implies that m can be written as the sum of distinct powers of 2. Thus, $2m$ can also be written as the sum of distinct powers of 2, each of which is greater than 2^0. If n is even, then $n = 2m$ and we are done; otherwise, $n = 2m + 2^0$ is the the sum of distinct powers of 2.

In either case, we conclude that n can be written as the sum of distinct powers of 2. □

The third proof deviates slightly from the induction boilerplate. At the top level, this proof doesn't actually use induction at all! However, a key step requires its own (straightforward) inductive proof.

Proof by algorithm: Let n be an arbitrary non-negative integer. Let S be a multiset containing n copies of 2^0. Modify S by running the following algorithm:

```
while S has more than one copy of any element 2^i
    Remove two copies of 2^i from S
    Insert one copy of 2^(i+1) into S
```

Each iteration of this algorithm reduces the cardinality of S by 1, so the algorithm must eventually halt. When the algorithm halts, the elements of S are distinct. We claim that just after each iteration of the while loop, the elements of S sum to n.

> **Proof by induction:** Consider an arbitrary iteration of the loop. Assume inductively that just after each previous iteration, the elements of S sum to n. Before any iterations of the loop, the elements of S sum to n by definition. The induction hypothesis implies that just before the current iteration begins, the elements of S sum to n. The loop replaces two copies of some number 2^i with their sum 2^{i+1}, leaving the total sum of S unchanged. Thus, when the iteration ends, the elements of S sum to n. □

Thus, when the algorithm halts, the elements of S are distinct powers of 2 that sum to n. We conclude that n can be written as the sum of distinct powers of 2. □

The fourth proof uses so-called 'weak' induction, where the inductive hypothesis can only be applied at $n-1$. Not surprisingly, tying all but one hand behind our backs makes the resulting proof longer, more complicated, and harder to read. It doesn't help that the algorithm used in the proof is overly specific. Nevertheless, this is the first approach that occurs to most students who have not truly accepted the Recursion Fairy into their hearts.

Proof by baby-step induction: Let n be an arbitrary non-negative integer. Assume that any non-negative integer less than n can be written as the sum of distinct powers of 2. There are two cases to consider: Either $n = 0$ or $n \geq 1$.

- The base case $n = 0$ is trivial—the elements of the empty set are distinct and sum to zero.
- Suppose $n \geq 1$. The inductive hypothesis implies that $n - 1$ can be written as the sum of distinct powers of 2. Thus, n can be written as the sum of powers of 2, which are distinct except possibly for two copies of 2^0. Let S be this multiset of powers of 2.

  Now consider the following algorithm:

  ```
  i ← 0
  while S has more than one copy of 2^i
      Remove two copies of 2^i from S
      Insert one copy of 2^(i+1) into S
      i ← i + 1
  ```

  Each iteration of this algorithm reduces the cardinality of S by 1, so the algorithm must eventually halt. We claim that for every non-negative integer i, the following invariants are satisfied after the ith iteration of the while loop (or before the algorithm starts if $i = 0$):

  - The elements of S sum to n.

    Proof by induction: Let i be an arbitrary non-negative integer. Assume that for any non-negative integer $j \leq i$, after the jth iteration of the while loop, the elements of S sum to n. If $i = 0$, the elements of S sum to n by definition of S. Otherwise, the induction hypothesis implies that just *before* the ith iteration, the elements of S sum to n; the ith iteration replaces two copies of 2^i with 2^{i+1}, leaving the sum unchanged. □

  - The elements in S are distinct, except possibly for two copies of 2^i.

    Proof by induction: Let i be an arbitrary non-negative integer. Assume that for any non-negative integer $j \leq i$, after the jth iteration of the while loop, the elements of S are distinct except possibly for two copies of 2^j. If $i = 0$, the invariant holds by definition of S. So suppose $i > 0$. The induction hypothesis implies that just *before* the ith iteration, the elements of S are distinct except possibly for two copies of 2^i. If there are two copies of 2^i, the algorithm replaces them both with 2^{i+1}, and the invariant is established; otherwise, the algorithm halts, and the invariant is again established. □

  The second invariant implies that when the algorithm halts, the elements of S are distinct.

In both cases, we conclude that n can be written as the sum of distinct powers of 2. □

Repeat after me: "Doctor! Doctor! It hurts when I do this!"

8 Irrational Numbers Exist

Theorem 7. *$\sqrt{2}$ is irrational.*

Proof: I will prove that $p^2 \neq 2q^2$ (and thus $p/q \neq \sqrt{2}$) for all positive integers p and q.

Let p and q be arbitrary positive integers. Assume that for any positive integers $i < p$ and $j < q$, we have $i^2 \neq 2j^2$. Let $i = \lfloor p/2 \rfloor$ and $j = \lfloor q/2 \rfloor$. There are three cases to consider:

- Suppose p is odd. Then $p^2 = (2i+1)^2 = 4i^2 + 4i + 1$ is odd, but $2q^2$ is even.
- Suppose p is even and q is odd. Then $p^2 = 4i^2$ is divisible by 4, but $2q^2 = 2(2j+1)^2 = 4(2j^2+2j)+2$ is not divisible by 4.
- Finally, suppose p and q are both even. The induction hypothesis implies that $i^2 \neq 2j^2$. Thus, $p^2 = 4i^2 \neq 8j^2 = 2q^2$.

In every case, we conclude that $p^2 \neq 2q^2$. □

This proof is usually presented as a proof by ***infinite descent***, which is just another form of proof by smallest counterexample. Notice that the induction hypothesis assumed that *both* p and q were as small as possible. Notice also that the 'base cases' included every pair of integers p and q where at least one of the integers is odd.

9 Fibonacci Parity

The ***Fibonacci numbers*** $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$ are recursively defined as follows:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2 \end{cases}$$

Theorem 8. *For all non-negative integers n, F_n is even if and only if n is divisible by 3.*

Proof: Let n be an arbitrary non-negative integer. Assume that for all non-negative integers $k < n$, F_k is even if and only if n is divisible by 3. There are three cases to consider: $n = 0$, $n = 1$, and $n \geq 2$.

- If $n = 0$, then n is divisible by 3, and $F_n = 0$ is even.
- If $n = 1$, then n is not divisible by 3, and $F_n = 1$ is odd.
- If $n \geq 2$, there are two subcases to consider: Either n is divisible by 3, or it isn't.
  - Suppose n is divisible by 3. Then neither $n-1$ nor $n-2$ is divisible by 3. Thus, the inductive hypothesis implies that both F_{n-1} and F_{n-2} are odd. So F_n is the sum of two odd numbers, and is therefore even.
  - Suppose n is not divisible by 3. Then exactly one of the numbers $n-1$ and $n-2$ is divisible by 3. Thus, the inductive hypothesis implies that exactly one of the numbers F_{n-1} and F_{n-2} is even, and the other is odd. So F_n is the sum of an even number and an odd number, and is therefore odd.

In all cases, F_n is even if and only if n is divisible by 3. □

10 Recursive Functions

Theorem 9. *Suppose the function $F : \mathbb{N} \to \mathbb{N}$ is defined recursively by setting $F(0) = 0$ and $F(n) = 1 + F(\lfloor n/2 \rfloor)$ for every positive integer n. Then for every positive integer n, we have $F(n) = 1 + \lfloor \log_2 n \rfloor$.*

Proof: Let n be an arbitrary positive integer. Assume that $F(k) = 1 + \lfloor \log_2 k \rfloor$ for every positive integer $k < n$. There are two cases to consider: Either $n = 1$ or $n \geq 2$.

- Suppose $n = 1$. Then $F(n) = F(1) = 1 + F(\lfloor 1/2 \rfloor) = 1 + F(0) = 1$ and $1 + \lfloor \log_2 n \rfloor = 1 + \lfloor \log_2 1 \rfloor = 1 + \lfloor 0 \rfloor = 1$.
- Suppose $n \geq 2$. Because $1 \leq \lfloor n/2 \rfloor < n$, the induction hypothesis implies that $F(\lfloor n/2 \rfloor) = 1 + \lfloor \log_2 \lfloor n/2 \rfloor \rfloor$. The definition of $F(n)$ now implies that $F(n) = 1 + F(\lfloor n/2 \rfloor) = 2 + \lfloor \log_2 \lfloor n/2 \rfloor \rfloor$.

  Now there are two subcases to consider: n is either even or odd.

  - If n is even, then $\lfloor n/2 \rfloor = n/2$, which implies

$$\begin{aligned} F(n) &= 2 + \lfloor \log_2 \lfloor n/2 \rfloor \rfloor \\ &= 2 + \lfloor \log_2 (n/2) \rfloor \\ &= 2 + \lfloor (\log_2 n) - 1 \rfloor \\ &= 2 + \lfloor \log_2 n \rfloor - 1 \\ &= 1 + \lfloor \log_2 n \rfloor. \end{aligned}$$

  - If n is odd, then $\lfloor n/2 \rfloor = (n-1)/2$, which implies

$$\begin{aligned} F(n) &= 2 + \lfloor \log_2 \lfloor n/2 \rfloor \rfloor \\ &= 2 + \lfloor \log_2 ((n-1)/2) \rfloor \\ &= 1 + \lfloor \log_2 (n-1) \rfloor \\ &= 1 + \lfloor \log_2 n \rfloor \end{aligned}$$

    by the algebra in the even case. Because $n > 1$ and n is odd, n cannot be a power of 2; thus, $\lfloor \log_2 n \rfloor = \lfloor \log_2 (n-1) \rfloor$.

In all cases, we conclude that $F(n) = 1 + \lfloor \log_2 n \rfloor$. □

11 Trees

Recall that a *tree* is a connected undirected graph with no cycles. A *subtree* of a tree T is a connected subgraph of T; a *proper* subtree is any tree except T itself.

Theorem 10. *In every tree, the number of vertices is one more than the number of edges.*

This one is actually pretty easy to prove directly from the definition of 'tree': a connected acyclic graph.

Proof: Let T be an arbitrary tree. Choose an arbitrary vertex v of T to be the root, and direct every edge of T outward from v. Because T is connected, every node except v has *at least* one edge directed into it. Because T is acyclic, every node has *at most* one edge directed into it, and no edge is directed into v. Thus, for every node $x \neq v$, there is *exactly* one edge directed into x. We conclude that the number of edges is one less than the number of nodes. □

But we can prove this theorem by induction as well, in several different ways. Each inductive proof is structured around a different recursive definition of 'tree'. First, a tree is either a single node, or two trees joined by an edge.

Proof: Let T be an arbitrary tree. Assume that in any proper subtree of T, the number of vertices is one more than the number of edges. There are two cases to consider: Either T has one vertex, or T has more than one vertex.

- If T has one vertex, then it has no edges.
- Suppose T has more than one vertex. Because T is connected, every pair of vertices is joined by a path. Thus, T must contain at least one edge. Let e be an arbitrary edge of T, and consider the graph $T \setminus e$ obtained by deleting e from T.

  Because T is acyclic, there is no path in $T \setminus e$ between the endpoints of e. Thus, T has at least two connected components. On the other hand, because T is connected, $T \setminus e$ has at most two connected components. Thus, $T \setminus e$ has exactly two connected components; call them A and B.

  Because T is acyclic, subgraphs A and B are also acyclic. Thus, A and B are subtrees of T, and therefore the induction hypothesis implies that $|E(A)| = |V(A)| - 1$ and $|E(B)| = |V(B)| - 1$.

  Because A and B do not share any vertices or edges, we have $|V(T)| = |V(A)| + |V(B)|$ and $|E(T)| = |E(A)| + |E(B)| + 1$.

  Simple algebra now implies that $|E(T)| = |V(T)| - 1$.

In both cases, we conclude that the number of vertices in T is one more than the number of edges in T. □

Second, a tree is a single node connected by edges to a finite set of trees.

Proof: Let T be an arbitrary tree. Assume that in any proper subtree of T, the number of vertices is one more than the number of edges. There are two cases to consider: Either T has one vertex, or T has more than one vertex.

- If T has one vertex, then it has no edges.
- Suppose T has more than one vertex. Let v be an arbitrary vertex of T, and let d be the degree of v. Delete v and all its incident edges from T to obtain a new graph G. This graph has exactly d connected components; call them $G_1, G_2, \ldots, G_d$. Because T is acyclic, every subgraph of T is acyclic. Thus, every subgraph G_i is a proper subtree of G. So the induction hypothesis implies that $|E(G_i)| = |V(G_i)| - 1$ for each i. We conclude that

$$|E(T)| = d + \sum_{i=1}^{d} |E(G_i)| = d + \sum_{i=1}^{d} (|V(G_i)| - 1) = \sum_{i=1}^{d} |V(G_i)| = |V(T)| - 1.$$

In both cases, we conclude that the number of vertices in T is one more than the number of edges in T. □

But you should ***never*** attempt to argue like this:

<u>Not</u> a Proof: The theorem is clearly true for the 1-node tree. So let T be an arbitrary tree with at least two nodes. Assume inductively that the number of vertices in T is one more than the number of edges in T. Suppose we add one more leaf to T to get a new tree T'. This new tree has one more vertex than T and one more edge than T. Thus, the number of vertices in T' is one more than the number of edges in T'. □

This is not a proof. Every sentence is true, and the connecting logic is correct, but it does not imply the theorem, because it doesn't *explicitly* consider *all possible* trees. Why should the reader believe that their favorite tree can be recursively constructed by adding leaves to a 1-node tree? It's *true*, of course, but that argument doesn't *prove* it. Remember: ***There are only two ways to prove any universally quantified statement:*** Directly ("Let T be an arbitrary tree. . . ") or by contradiction ("Suppose some tree T doesn't. . . ").

Here is a *correct* inductive proof using the same underlying idea. In this proof, I don't have to prove that the proof considers arbitrary trees; it says so right there on the first line! As usual, the proof very strongly resembles a recursive algorithm, including a subroutine to find a leaf.

Proof: Let T be an arbitrary tree. Assume that in any proper subtree of T, the number of vertices is one more than the number of edges. There are two cases to consider: Either T has one vertex, or T has more than one vertex.

- If T has one vertex, then it has no edges.
- Otherwise, T must have at least one vertex of degree 1, otherwise known as a leaf.

  Proof: Consider a walk through the graph T that starts at an arbitrary vertex and continues as long as possible without repeating any edge. The walk can never visit the same vertex more than once, because T is acyclic. Whenever the walk visits a vertex of degree at least 2, it can continue further, because that vertex has at least one unvisited edge. But the walk must eventually end, because T is finite. Thus, the walk must eventually reach a vertex of degree 1. □

  Let ℓ be an arbitrary leaf of T, and let T' be the tree obtained by deleting ℓ from T. Then we have the identity

$$|E(T)| = |E(T')| + 1 = |V(T')| = |V(T)| - 1,$$

  where the first and third equalities follow from the definition of T', and the second equality follows from the inductive hypothesis.

In both cases, we conclude that the number of vertices in T is one more than the number of edges in T. □

Exercises

1. Prove that given an unlimited supply of 6-cent coins, 10-cent coins, and 15-cent coins, one can make any amount of change larger than 29 cents.

2. Prove that $\displaystyle\sum_{i=0}^{n} r^i = \frac{1-r^{n+1}}{1-r}$ for every non-negative integer n and every real number $r \neq 1$.

3. Prove that $\displaystyle\left(\sum_{i=0}^{n} i\right)^2 = \sum_{i=0}^{n} i^3$ for every non-negative integer n.

4. Recall the standard recursive definition of the Fibonacci numbers: $F_0 = 0$, $F_1 = 1$, and

$F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. Prove the following identities for all non-negative integers n and m.

(a) $\sum_{i=0}^{n} F_i = F_{n+2} - 1$

(b) $F_n^2 - F_{n+1}F_{n-1} = (-1)^{n+1}$

*(c) If n is an integer multiple of m, then F_n is an integer multiple of F_m.

5. Prove that every integer (positive, negative, or zero) can be written in the form $\sum_i \pm 3^i$, where the exponents i are distinct non-negative integers. For example:

$$42 = 3^4 - 3^3 - 3^2 - 3^1$$
$$25 = 3^3 - 3^1 + 3^0$$
$$17 = 3^3 - 3^2 - 3^0$$

6. Prove that every integer (positive, negative, or zero) can be written in the form $\sum_i (-2)^i$, where the exponents i are distinct non-negative integers. For example:

$$42 = (-2)^6 + (-2)^5 + (-2)^4 + (-2)^0$$
$$25 = (-2)^6 + (-2)^5 + (-2)^3 + (-2)^0$$
$$17 = (-2)^4 + (-2)^0$$

7. (a) Prove that every non-negative integer can be written as the sum of distinct, non-consecutive Fibonacci numbers. That is, if the Fibonacci number F_i appears in the sum, it appears exactly once, and its neighbors F_{i-1} and F_{i+1} do not appear at all. For example:

$$17 = F_7 + F_4 + F_2$$
$$42 = F_9 + F_6$$
$$54 = F_9 + F_7 + F_5 + F_3$$

(b) Prove that every positive integer can be written as the sum of distinct Fibonacci numbers *with no consecutive gaps*. That is, for any index $i \geq 1$, if the consecutive Fibonacci numbers F_i or F_{i+1} do not appear in the sum, then no larger Fibonacci number F_j with $j > i$ appears in the sum. In particular, the sum *must* include either F_1 or F_2. For example:

$$16 = F_6 + F_5 + F_3 + F_2$$
$$42 = F_8 + F_7 + F_5 + F_3 + F_1$$
$$54 = F_8 + F_7 + F_6 + F_5 + F_4 + F_3 + F_2 + F_1$$

(c) The Fibonacci sequence can be extended backward to negative indices by rearranging the defining recurrence: $F_n = F_{n+2} - F_{n+1}$. Here are the first several negative-index Fibonacci numbers:

n	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
F_n	-55	34	-21	13	-8	5	-3	2	-1	1

Prove that $F_{-n} = (-1)^{n+1} F_n$.

*(d) Prove that *every* integer—positive, negative, or zero—can be written as the sum of distinct, non-consecutive Fibonacci numbers *with negative indices*. For example:

$$\begin{aligned} 17 &= F_{-7} + F_{-5} + F_{-2} \\ -42 &= F_{-10} + F_{-7} \\ 54 &= F_{-9} + F_{-7} + F_{-5} + F_{-3} + F_{-1}. \end{aligned}$$

8. Consider the following game played with a finite number of a identical coins, which are arranged into *stacks*. Each coin belongs to exactly one stack. Let n_i denote the number of coins in stack i. In each turn, you must make one of the following moves:

   - For some i and j such that $n_j \leq n_i - 2$, move one coin from stack i to stack j.
   - Move one coin from any stack into a new stack.
   - Find a stack containing only one coin, and remove that coin from the game.

   The game ends when all coins are gone. For example, the following sequence of turns describes a complete game; each vector lists the number of coins in each non-empty stack:

$$\begin{aligned} \langle 4,2,1\rangle &\Longrightarrow \langle 4,1,1,1\rangle \Longrightarrow \langle 3,2,1,1\rangle \Longrightarrow \langle 2,2,2,1\rangle \Longrightarrow \langle 2,2,1,1,1\rangle \\ &\Longrightarrow \langle 2,1,1,1,1,1\rangle \Longrightarrow \langle 2,1,1,1,1\rangle \Longrightarrow \langle 2,1,1,1\rangle \Longrightarrow \langle 2,1,1\rangle \\ &\Longrightarrow \langle 2,1\rangle \Longrightarrow \langle 2\rangle \Longrightarrow \langle 1,1\rangle \Longrightarrow \langle 1\rangle \Longrightarrow \langle\rangle \end{aligned}$$

   (a) Prove that this game ends after a finite number of turns.

   (b) What are the minimum and maximum number of turns in a game, if we start with a single stack of n coins? Prove your answers are correct.

   (c) Now suppose each time you remove a coin from a stack, you must place *two* coins onto smaller stacks. In each turn, you must make one of the following moves:

   - For some indices i, j, and k such that $n_j \leq n_i - 2$ and $n_k \leq n_i - 2$ and $j \neq k$, remove a coin from stack i, add a coin to stack j, and add a coin to stack k.
   - For some i and j such that $n_j \leq n_i - 2$, remove a coin from stack i, add a coin to stack j, and create a new stack with one coin.
   - Remove one coin from any stack and create two new stacks, each with one coin.
   - Find a stack containing only one coin, and remove that coin from the game.

   For example, the following sequence of turns describes a complete game:

$$\begin{aligned} \langle 4,2,1\rangle &\Longrightarrow \langle 3,3,2\rangle \Longrightarrow \langle 3,2,2,1,1\rangle \Longrightarrow \langle 3,2,2,1\rangle \Longrightarrow \langle 3,2,2\rangle \Longrightarrow \langle 3,2,1,1,1\rangle \\ &\Longrightarrow \langle 2,2,2,2,1\rangle \Longrightarrow \langle 2,2,2,2\rangle \Longrightarrow \langle 2,2,2,1,1,1\rangle \Longrightarrow \langle 2,2,2,1,1\rangle \\ &\Longrightarrow \langle 2,2,2,1\rangle \Longrightarrow \langle 2,2,2\rangle \Longrightarrow \langle 2,2,1,1,1\rangle \Longrightarrow \langle 2,2,1,1\rangle \Longrightarrow \langle 2,2,1\rangle \\ &\Longrightarrow \langle 2,2\rangle \Longrightarrow \langle 2,1,1,1\rangle \Longrightarrow \langle 1,1,1,1,1,1\rangle \Longrightarrow \langle 1,1,1,1,1\rangle \Longrightarrow \langle 1,1,1,1\rangle \\ &\Longrightarrow \langle 1,1,1\rangle \Longrightarrow \langle 1,1\rangle \Longrightarrow \langle 1\rangle \Longrightarrow \langle\rangle. \end{aligned}$$

   Prove that this modified game still ends after a finite number of turns.

   (d) What are the minimum and maximum number of turns in this modified game, starting with a single stack of n coins? Prove your answers are correct.

9. (a) Prove that $|A \times B| = |A| \times |B|$ for all finite sets A and B.

   (b) Prove that for all *non-empty* finite sets A and B, there are exactly $|B|^{|A|}$ functions from A to B.

10. Recall that a binary tree is *full* if every node has either two children (an internal node) or no children (a leaf). ***Give at least four different proofs*** of the following fact: *In any full binary tree, the number of leaves is exactly one more than the number of internal nodes.*

11. A *binomial tree of order* k is defined recursively as follows:

    - A binomial tree of order 0 is a single node.
    - For all $k > 0$, a binomial tree of order k consists of two binomial trees of order $k-1$, with the root of one tree connected as a new child of the root of the other. (See the figure below.)

    Prove the following claims:

    (a) For all non-negative integers k, a binomial tree of order k has exactly 2^k nodes.

    (b) For all positive integers k, attaching a new leaf to every node in a binomial tree of order $k-1$ results in a binomial tree of order k.

    (c) For all non-negative integers k and d, a binomial tree of order k has exactly $\binom{k}{d}$ nodes with depth d. (Hence the name!)

Binomial trees of order 0 through 5.
Top row: The recursive definition. Bottom row: The property claimed in part (b).

12. The nth *Fibonacci binary tree* $\mathcal{F}_n$ is defined recursively as follows:

    - $\mathcal{F}_1$ is a single root node with no children.
    - For all $n \geq 2$, $\mathcal{F}_n$ is obtained from $\mathcal{F}_{n-1}$ by adding a right child to every leaf and adding a left child to every node that has only one child.

    (a) Prove that the number of leaves in $\mathcal{F}_n$ is precisely the nth Fibonacci number: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$.

    (b) How many nodes does $\mathcal{F}_n$ have? Give an exact, closed-form answer in terms of Fibonacci numbers, and prove your answer is correct.

(c) Prove that for all $n \geq 2$, the right subtree of $\mathcal{F}_n$ is a copy of $\mathcal{F}_{n-1}$.

(d) Prove that for all $n \geq 3$, the left subtree of $\mathcal{F}_n$ is a copy of $\mathcal{F}_{n-2}$.

The first six Fibonacci binary trees. In each tree $\mathcal{F}_n$, the subtree of gray nodes is $\mathcal{F}_{n-1}$.

13. The d-dimensional hypercube is the graph defined as follows. There are 2^d vertices, each labeled with a different string of d bits. Two vertices are joined by an edge if and only if their labels differ in exactly one bit.

The 1-dimensional, 2-dimensional, and 3-dimensional hypercubes.

Recall that a Hamiltonian cycle is a closed walk that visits each vertex in a graph exactly once. Prove that for every integer $d \geq 2$, the d-dimensional hypercube has a Hamiltonian cycle.

14. A ***tournament*** is a directed graph with exactly one directed edge between each pair of vertices. That is, for any vertices v and w, a tournament contains either an edge $v \to w$ or an edge $w \to v$, but not both. A ***Hamiltonian path*** in a directed graph G is a directed path that visits every vertex of G exactly once.

(a) Prove that every tournament contains a Hamiltonian path.

(b) Prove that every tournament contains either *exactly one* Hamiltonian path or a directed cycle of length three.

A tournament with two Hamiltonian paths $u \to v \to w \to x \to z \to y$ and $y \to u \to v \to x \to z \to w$ and a directed triangle $w \to x \to z \to w$.

15. Scientists recently discovered a planet, tentatively named "Ygdrasil", that is inhabited by a bizarre species called "nertices" (singular "nertex"). All nertices trace their ancestry back to a particular nertex named Rudy. Rudy is still quite alive, as is every one of his many descendants. Nertices reproduce asexually; every nertex has exactly one parent (except Rudy, who sprang forth fully formed from the planet's core). There are three types of nertices—red, green, and blue. The color of each nertex is correlated exactly with the number and color of its children, as follows:

    - Each red nertex has two children, exactly one of which is green.
    - Each green nertex has exactly one child, which is not green.
    - Blue nertices have no children.

    In each of the following problems, let R, G, and B respectively denote the number of red, green, and blue nertices on Ygdrasil.

    (a) Prove that $B = R + 1$.

    (b) Prove that either $G = R$ or $G = B$.

    (c) Prove that $G = B$ if and only if Rudy is green.

16. ***Well-formed formulas*** (wffs) are defined recursively as follows:

    - T is a wff.
    - F is a wff.
    - Any proposition variable is a wff.
    - If X is a wff, then $(\neg X)$ is also a wff.
    - If X and Y are wffs, then $(X \wedge Y)$ is also a wff.
    - If X and Y are wffs, then $(X \vee Y)$ is also a wff.

    We say that a formula is in ***De Morgan normal form*** if it satisfies the following conditions. ("De Morgan normal form" is not standard terminology; I just made it up.)

    - Every negation in the formula is applied to a variable, not to a more complicated subformula.
    - Either the entire formula is T, or the formula does not contain T.
    - Either the entire formula is F, or the formula does not contain F.

    Prove that for every wff, there is a logically equivalent wff in De Morgan normal form. For example, the well-formed formula

    $$(\neg((p \wedge q) \vee \neg r)) \wedge (\neg(p \vee \neg r) \wedge q)$$

    is logically equivalent to the following wff in De Morgan normal form:

    $$(((\neg p \vee \neg q) \wedge r)) \wedge ((\neg p \wedge r) \wedge q)$$

17. A ***polynomial*** is a function $f : \mathbb{R} \to \mathbb{R}$ of the form $f(x) = \sum_{i=0}^{d} a_i x^i$ for some non-negative integer d (called the degree) and some real numbers $a_0, a_1, \ldots, a_d$ (called the coefficients).

(a) Prove that the sum of two polynomials is a polynomial.

(b) Prove that the product of two polynomials is a polynomial.

(c) Prove that the composition $f(g(x))$ of two polynomials $f(x)$ and $g(x)$ is a polynomial.

(d) Prove that the derivative f' of a polynomial f is a polynomial, using ***only*** the following facts:

- Constant rule: If f is constant, then f' is identically zero.
- Sum rule: $(f+g)' = f' + g'$.
- Product rule: $(f \cdot g)' = f' \cdot g + f \cdot g'$.

*18. An ***arithmetic expression tree*** is a binary tree where every leaf is labeled with a variable, every internal node is labeled with an arithmetic operation, and every internal node has exactly two children. For this problem, assume that the only allowed operations are $+$ and $\times$. Different leaves may or may not represent different variables.

Every arithmetic expression tree represents a function, transforming input values for the leaf variables into an output value for the root, by following two simple rules: (1) The value of any $+$-node is the sum of the values of its children. (2) The value of any $\times$-node is the product of the values of its children.

Two arithmetic expression trees are ***equivalent*** if they represent the same function; that is, the same input values for the leaf variables always leads to the same output value at both roots. An arithmetic expression tree is in ***normal form*** if the parent of every $+$-node (if any) is another $+$-node.

Three equivalent expression trees. Only the third is in normal form.

Prove that for any arithmetic expression tree, there is an equivalent arithmetic expression tree in normal form. *[Hint: This is harder than it looks.]*

*19. A ***Gaussian integer*** is a complex number of the form $x + yi$, where x and y are integers. Prove that any Gaussian integer can be expressed as the sum of distinct powers of the complex number $\alpha = -1 + i$. For example:

$$
\begin{aligned}
4 &= 16 + (-8-8i) + 8i + (-4) &&= \alpha^8 + \alpha^7 + \alpha^6 + \alpha^4 \\
-8 &= (-8-8i) + 8i &&= \alpha^7 + \alpha^6 \\
15i &= (-16+16i) + 16 + (-2i) + (-1+i) + 1 &&= \alpha^9 + \alpha^8 + \alpha^2 + \alpha^1 + \alpha^0 \\
1+6i &= (8i) + (-2i) + 1 &&= \alpha^6 + \alpha^2 + \alpha^0 \\
2-3i &= (4-4i) + (-4) + (2+2i) + (-2i) + (-1+i) + 1 &&= \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 + \alpha^1 + \alpha^0 \\
-4+2i &= (-16+16i) + 16 + (-8-8i) + (4-4i) + (-2i) &&= \alpha^9 + \alpha^8 + \alpha^7 + \alpha^5 + \alpha^2
\end{aligned}
$$

The following list of values may be helpful:

$\alpha^0 = 1$	$\alpha^4 = -4$	$\alpha^8 = 16$	$\alpha^{12} = -64$
$\alpha^1 = -1 + i$	$\alpha^5 = 4 - 4i$	$\alpha^9 = -16 + 16i$	$\alpha^{13} = 64 - 64i$
$\alpha^2 = -2i$	$\alpha^6 = 8i$	$\alpha^{10} = -32i$	$\alpha^{14} = 128i$
$\alpha^3 = 2 + 2i$	$\alpha^7 = -8 - 8i$	$\alpha^{11} = 32 + 32i$	$\alpha^{15} = -128 - 128i$

[Hint: How do you write $-2 - i$?]

*20. *Lazy binary* is a variant of standard binary notation for representing natural numbers where we allow each "bit" to take on one of three values: 0, 1, or 2. Lazy binary notation is defined inductively as follows.

- The lazy binary representation of zero is 0.
- Given the lazy binary representation of any non-negative integer n, we can construct the lazy binary representation of $n + 1$ as follows:
  (a) increment the rightmost digit;
  (b) if any digit is equal to 2, replace the rightmost 2 with 0 and increment the digit immediately to its left.

Here are the first several natural numbers in lazy binary notation:

0, 1, 10, 11, 20, 101, 110, 111, 120, 201, 210, 1011, 1020, 1101, 1110, 1111, 1120, 1201, 1210, 2011, 2020, 2101, 2110, 10111, 10120, 10201, 10210, 11011, 11020, 11101, 11110, 11111, 11120, 11201, 11210, 12011, 12020, 12101, 12110, 20111, 20120, 20201, 20210, 21011, 21020, 21101, 21110, 101111, 101120, 101201, 101210, 102011, 102020, ...

(a) Prove that in any lazy binary number, between any two 2s there is at least one 0, and between two 0s there is at least one 2.

(b) Prove that for any natural number N, the sum of the digits of the lazy binary representation of N is exactly $\lfloor \lg(N + 1) \rfloor$.

★21. Consider the following recursively defined sequence of rational numbers:

$$R_0 = 0$$
$$R_n = \frac{1}{2\lfloor R_{n-1} \rfloor - R_{n-1} + 1} \qquad \text{for all } n \geq 1$$

The first several elements of this sequence are

$$0, 1, \frac{1}{2}, 2, \frac{1}{3}, \frac{3}{2}, \frac{2}{3}, 3, \frac{1}{4}, \frac{4}{3}, \frac{3}{5}, \frac{5}{2}, \frac{2}{5}, \frac{5}{3}, \frac{3}{4}, 4, \frac{1}{5}, \ldots$$

Prove that every non-negative rational number appears in this sequence exactly once.

22. Let $f : \mathbb{R}^+ \to \mathbb{R}^+$ be an ***arbitrary*** (not necessarily continuous) function such that

    - $f(x) > 0$ for all $x > 0$, and
    - $f(x) = \pi f(x/\sqrt{2})$ for all $x > 1$.

    Prove by induction that $f(x) = \Theta(x)$ (as $x \to \infty$). Yes, this is induction over the real numbers.

*23. There is a natural generalization of induction to the real numbers that is familiar to analysts but relatively unknown in computer science. The precise formulation given below is was proposed independently by Hathaway[2] and Clark[3] fairly recently, but the idea dates back to at least to the 1920s. Recall that any real numbers a and z define four different intervals:

    - The ***open*** interval $\mathbf{(a,z)} := \{t \in \mathbb{R} \mid a \le t < z\}$,
    - The ***half-open*** interval $\mathbf{[a,z)} := \{t \in \mathbb{R} \mid a \le t < z\}$,
    - The ***half-open*** interval $\mathbf{(a,z]} := \{t \in \mathbb{R} \mid a < t \le z\}$,
    - The ***closed*** interval $\mathbf{[a,z]} := \{t \in \mathbb{R} \mid a \le t \le z\}$.

    Theorem 11 (Continuous Induction). *Fix a closed interval $[a,z] \subset \mathbb{R}$. Suppose some subset $S \subseteq [a,z]$ has following properties:*

    (a) $a \in S$.

    (b) If $a \le s < z$ and $s \in S$, then $[s,u] \subseteq S$ for some $u > s$.

    (c) If $a \le s \le z$ and $[a,s) \subseteq S$, then $s \in S$.

    Then $S = [a,z]$.

    Proof: For the sake of argument, let S be a proper subset of $[a,b]$. Let $T = [a,z] \setminus S$. Because $\bar{S}$ is bounded but non-empty, it has a greatest lower bound $\ell \in [a,z]$. More explicitly, ℓ be the largest real number such that $\ell \le t$ for all $t \in T$. There are three cases to consider:

    - Suppose $\ell = a$. Condition (a) and (b) imply that $[a,u] \in S$ for some $u > a$. But then we have $\ell = a < u \le t$ for all $t \in T$, contradicting the fact that ℓ is the *greatest* lower bound of T.
    - Suppose $\ell > a$ and $\ell \in S$. If $\ell = z$, then $S = [a,z]$, contradicting our initial assumption. Otherwise, by condition (b), we have $[\ell,u] \subseteq S$ for some $u > \ell$, again contradicting the fact that ℓ is the *greatest* lower bound of T.
    - Finally, suppose $\ell > a$ and $\ell \in \overline{S}$. Because no element of T is smaller than ℓ, we have $[a,\ell) \subseteq S$. But then condition (c) implies that $\ell \in S$, and we have a contradiction.

    In all cases, we have a contradiction. □

    Continuous induction hinges on the ***axiom of completeness***—every non-empty set of positive real numbers has a greatest lower bound—just as standard induction requires the ***well-ordering principle***—every non-empty set of positive integers has a smallest element.

[2] Dan Hathaway. Using continuity induction. *College Math. J.* 42:229–231, 2011.
[3] Pete L. Clark. The instructor's guide to real induction. arXiv:1208.0973.

Thus, continuous induction cannot be used to prove properties of *rational* numbers, because the greatest lower bound of a set of rational numbers need not be rational.

Fix real numbers $a \leq z$. Recall that a function $f : [a, z] \to \mathbb{R}$ is ***continuous*** if it satisfies the following condition: for any $t \in [a, z]$ and any $\varepsilon > 0$, there is some $\delta > 0$ such that for all $u \in [a, z]$ with $|t - u| \leq \delta$, we have $|f(t) - f(u)| \leq \varepsilon$. Prove the following theorems using continuous induction.

(a) ***Connectedness:*** *There is no continuous function from* $[a, z]$ *to the set* $\{0, 1\}$.

(b) ***Intermediate Value Theorem:*** *For any continuous function* $f : [a, z] \to \mathbb{R} \setminus \{0\}$*, if* $f(a) > 0$*, then* $f(t) > 0$ *for all* $a \leq t \leq z$.

(c) ***Extreme Value Theorem:*** *Any continuous function* $f : [a, z] \to \mathbb{R}$ *attains its maximum value; that is, there is some* $t \in [a, z]$ *such that* $f(t) \geq f(u)$ *for all* $u \in [a, z]$.

★(d) ***The Heine-Borel Theorem:*** *The interval* $[a, z]$ *is compact.*

This one requires some expansion.

- A set $X \subseteq \mathbb{R}$ is ***open*** if every point in X lies inside an open interval contained in X.
- An ***open cover*** of $[a, z]$ is a (possibly uncountably infinite) family $\mathcal{U} = \{U_i \mid i \in I\}$ of open sets U_i such that $[a, z] \subseteq \bigcup_{i \in I} U_i$.
- A ***subcover*** of $\mathcal{U}$ is a subset $\mathcal{V} \subseteq \mathcal{U}$ that is also a cover of $[a, z]$.
- A cover $\mathcal{U}$ is ***finite*** if it contains a finite number of open sets.
- Finally, a set $X \subseteq \mathbb{R}$ is ***compact*** if every open cover of X has a finite subcover.

The Heine-Borel theorem is one of the most fundamental results in real analysis, and the proof usually requires several pages. But the continuous-induction proof is shorter than the list of definitions!

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms/ for the most recent revision.

Change is certain. Peace is followed by disturbances; departure of evil men by their return. Such recurrences should not constitute occasions for sadness but realities for awareness, so that one may be happy in the interim.

— *I Ching [The Book of Changes]* (c. 1100 BC)

To endure *the idea of the recurrence one needs: freedom from morality; new means against the fact of pain (pain conceived as a tool, as the father of pleasure; there is no cumulative consciousness of displeasure); the enjoyment of all kinds of uncertainty, experimentalism, as a counterweight to this extreme fatalism; abolition of the concept of necessity; abolition of the "will"; abolition of "knowledge-in-itself."*

— Friedrich Nietzsche *The Will to Power* (1884)
[translated by Walter Kaufmann]

Wil Wheaton: *Embrace the dark side!*
Sheldon: *That's not even from your franchise!*

— "The Wheaton Recurrence", *Bing Bang Theory*, April 12, 2010

Solving Recurrences

1 Introduction

A ***recurrence*** is a recursive description of a function, or in other words, a description of a function in terms of itself. Like all recursive structures, a recurrence consists of one or more *base cases* and one or more *recursive cases*. Each of these cases is an equation or inequality, with some function value $f(n)$ on the left side. The base cases give explicit values for a (typically finite, typically small) subset of the possible values of n. The recursive cases relate the function value $f(n)$ to function value $f(k)$ for one or more integers $k < n$; typically, each recursive case applies to an infinite number of possible values of n.

For example, the following recurrence (written in two different but standard ways) describes the identity function $f(n) = n$:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ f(n-1)+1 & \text{otherwise} \end{cases} \qquad \begin{aligned} &f(0) = 0 \\ &f(n) = f(n-1)+1 \text{ for all } n > 0 \end{aligned}$$

In both presentations, the first line is the only base case, and the second line is the only recursive case. The same function can satisfy *many* different recurrences; for example, both of the following recurrences also describe the identity function:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) & \text{otherwise} \end{cases} \qquad f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2 \cdot f(n/2) & \text{if } n \text{ is even and } n > 0 \\ f(n-1)+1 & \text{if } n \text{ is odd} \end{cases}$$

We say that a particular function ***satisfies*** a recurrence, or is the ***solution*** to a recurrence, if each of the statements in the recurrence is true. Most recurrences—at least, those that we will encounter in this class—have a solution; moreover, if every case of the recurrence is an equation, that solution is unique. Specifically, if we transform the recursive formula into a recursive *algorithm*, the solution to the recurrence is the function computed by that algorithm!

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edi/teaching/algorithms/ for the most recent revision.

Recurrences arise naturally in the analysis of algorithms, especially recursive algorithms. In many cases, we can express the running time of an algorithm as a recurrence, where the recursive cases of the recurrence correspond exactly to the recursive cases of the algorithm. Recurrences are also useful tools for solving counting problems—How many objects of a particular kind exist?

By itself, a recurrence is not a satisfying description of the running time of an algorithm or a bound on the number of widgets. Instead, we need a ***closed-form*** solution to the recurrence; this is a *non-recursive* description of a function that satisfies the recurrence. For recurrence *equations*, we sometimes prefer an *exact* closed-form solution, but such a solution may not exist, or may be too complex to be useful. Thus, for most recurrences, especially those arising in algorithm analysis, we are satisfied with an *asymptotic* solution of the form $\Theta(g(n))$, for some explicit (non-recursive) function $g(n)$.

For recursive *inequalities*, we prefer a ***tight*** solution; this is a function that would still satisfy the recurrence if all the inequalities were replaced with the corresponding equations. Again, exactly tight solutions may not exist, or may be too complex to be useful, in which case we seek either a looser bound or an asymptotic solution of the form $O(g(n))$ or $\Omega(g(n))$.

2 The Ultimate Method: Guess and Confirm

Ultimately, there is only one fail-safe method to solve *any* recurrence:

Guess the answer, and then prove it correct by induction.

Later sections of these notes describe techniques to generate guesses that are guaranteed to be correct, provided you use them correctly. But if you're faced with a recurrence that doesn't seem to fit any of these methods, or if you've forgotten how those techniques work, don't despair! If you guess a closed-form solution and then try to verify your guess inductively, usually either the proof will succeed, in which case you're done, or the proof will fail, in which case *your failure will help you refine your guess*. Where you get your initial guess is utterly irrelevant[1]—from a classmate, from a textbook, on the web, from the answer to a different problem, scrawled on a bathroom wall in Siebel, included in a care package from your mom, dictated by the machine elves, whatever. If you can prove that the answer is correct, then it's correct!

2.1 Tower of Hanoi

The classical Tower of Hanoi problem gives us the recurrence $\boldsymbol{T(n) = 2T(n-1) + 1}$ with base case $\boldsymbol{T(0) = 0}$. Just looking at the recurrence we can guess that $T(n)$ is something like 2^n. If we write out the first few values of $T(n)$, we discover that they are each one less than a power of two.

$$T(0) = 0, \quad T(1) = 1, \quad T(2) = 3, \quad T(3) = 7, \quad T(4) = 15, \quad T(5) = 31, \quad T(6) = 63, \quad \ldots,$$

It looks like $\boldsymbol{T(n) = 2^n - 1}$ might be the right answer. Let's check.

$$T(0) = 0 = 2^0 - 1 \quad \checkmark$$

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2(2^{n-1} - 1) + 1 && \text{[induction hypothesis]} \\ &= 2^n - 1 \quad \checkmark && \text{[algebra]} \end{aligned}$$

[1]. . . except of course during exams, where you aren't supposed to use *any* outside sources

We were right! Hooray, we're done!

Another way we can guess the solution is by ***unrolling*** the recurrence, by substituting it into itself:

$$\begin{aligned} T(n) &= 2T(n-1)+1 \\ &= 2(2T(n-2)+1)+1 \\ &= 4T(n-2)+3 \\ &= 4(2T(n-3)+1)+3 \\ &= 8T(n-3)+7 \\ &= \cdots \end{aligned}$$

It looks like unrolling the initial Hanoi recurrence k times, for any non-negative integer k, will give us the new recurrence $T(n) = 2^k T(n-k) + (2^k - 1)$. Let's prove this by induction:

$$T(n) = 2T(n-1)+1 \quad \checkmark \qquad [k=0, \text{ by definition}]$$

$$\begin{aligned} T(n) &= 2^{k-1}T(n-(k-1)) + (2^{k-1}-1) && [\text{inductive hypothesis}] \\ &= 2^{k-1}\big(2T(n-k)+1\big) + (2^{k-1}-1) && [\text{initial recurrence for } T(n-(k-1))] \\ &= 2^k T(n-k) + (2^k - 1) \quad \checkmark && [\text{algebra}] \end{aligned}$$

Our guess was correct! In particular, unrolling the recurrence n times give us the recurrence $T(n) = 2^n T(0) + (2^n - 1)$. Plugging in the base case $T(0) = 0$ give us the closed-form solution $T(n) = 2^n - 1$.

2.2 Fibonacci numbers

Let's try a less trivial example: the Fibonacci numbers $\boldsymbol{F_n = F_{n-1} + F_{n-2}}$ with base cases $\boldsymbol{F_0 = 0}$ and $\boldsymbol{F_1 = 1}$. There is no obvious pattern in the first several values (aside from the recurrence itself), but we can reasonably guess that F_n is exponential in n. Let's try to prove inductively that $F_n \le \alpha \cdot c^n$ for some constants $a > 0$ and $c > 1$ and see how far we get.

$$\begin{aligned} F_n &= F_{n-1} + F_{n-2} \\ &\le \alpha \cdot c^{n-1} + \alpha \cdot c^{n-2} && [\text{“induction hypothesis”}] \\ &\le \alpha \cdot c^n \ \textbf{???} \end{aligned}$$

The last inequality is satisfied if $c^n \ge c^{n-1} + c^{n-2}$, or more simply, if $c^2 - c - 1 \ge 0$. The smallest value of c that works is $\phi = (1+\sqrt{5})/2 \approx 1.618034$; the other root of the quadratic equation has smaller absolute value, so we can ignore it.

So we have *most* of an inductive proof that $\boldsymbol{F_n \le \alpha \cdot \phi^n}$ for *some* constant α. All that we're missing are the base cases, which (we can easily guess) must determine the value of the coefficient a. We quickly compute

$$\frac{F_0}{\phi^0} = \frac{0}{1} = 0 \quad \text{and} \quad \frac{F_1}{\phi^1} = \frac{1}{\phi} \approx 0.618034 > 0,$$

so the base cases of our induction proof are correct as long as $\alpha \ge 1/\phi$. It follows that $\boldsymbol{F_n \le \phi^{n-1}}$ for all $n \ge 0$.

What about a matching lower bound? Essentially the same inductive proof implies that $F_n \ge \beta \cdot \phi^n$ for some constant β, but the only value of β that works for *all* n is the trivial $\beta = 0$!

We could try to find some lower-order term that makes the base case non-trivial, but an easier approach is to recall that asymptotic $\Omega(\,)$ bounds only have to work for *sufficiently large* n. So let's ignore the trivial base case $F_0 = 0$ and assume that $F_2 = 1$ is a base case instead. Some more easy calculation gives us

$$\frac{F_2}{\phi^2} = \frac{1}{\phi^2} \approx 0.381966 < \frac{1}{\phi}.$$

Thus, the new base cases of our induction proof are correct as long as $\beta \le 1/\phi^2$, which implies that $\boldsymbol{F_n \ge \phi^{n-2}}$ for all $n \ge 1$.

Putting the upper and lower bounds together, we obtain the tight asymptotic bound $\boldsymbol{F_n = \Theta(\phi^n)}$. It *is* possible to get a more exact solution by speculatively refining and conforming our current bounds, but it's not easy. Fortunately, if we really need it, we can get an exact solution using the *annihilator* method, which we'll see later in these notes.

2.3 Mergesort

Mergesort is a classical recursive divide-and-conquer algorithm for sorting an array. The algorithm splits the array in half, recursively sorts the two halves, and then merges the two sorted subarrays into the final sorted array.

```
MergeSort(A[1..n]):
  if (n > 1)
      m ← ⌊n/2⌋
      MergeSort(A[1..m])
      MergeSort(A[m+1..n])
      Merge(A[1..n], m)
```

```
Merge(A[1..n], m):
  i ← 1;  j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i+1
      else if i > m
          B[k] ← A[j];  j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i+1
      else
          B[k] ← A[j];  j ← j+1
  for k ← 1 to n
      A[k] ← B[k]
```

Let $T(n)$ denote the worst-case running time of MergeSort when the input array has size n. The Merge subroutine clearly runs in $\Theta(n)$ time, so the function $T(n)$ satisfies the following recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T\big(\lceil n/2 \rceil\big) + T\big(\lfloor n/2 \rfloor\big) + \Theta(n) & \text{otherwise.} \end{cases}$$

For now, let's consider the special case where n is a power of 2; this assumption allows us to take the floors and ceilings out of the recurrence. (We'll see how to deal with the floors and ceilings later; the short version is that they don't matter.)

Because the recurrence itself is given only asymptotically—in terms of $\Theta(\,)$ expressions—we can't hope for anything but an asymptotic solution. So we can safely simplify the recurrence further by removing the Θ's; any asymptotic solution to the simplified recurrence will also satisfy the original recurrence. (This simplification is actually important for another reason; if we kept the asymptotic expressions, we might be tempted to simplify them inappropriately.)

Our simplified recurrence now looks like this:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/2) + n & \text{otherwise.} \end{cases}$$

To guess at a solution, let's try unrolling the recurrence.

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2\big(2T(n/4) + n/2\big) + n \\ &= 4T(n/4) + 2n \\ &= 8T(n/8) + 3n = \cdots \end{aligned}$$

It looks like $T(n)$ satisfies the recurrence $T(n) = 2^k T(n/2^k) + kn$ for *any* positive integer k. Let's verify this by induction.

$$T(n) = 2T(n/2) + n = 2^1 T(n/2^1) + 1 \cdot n \quad \checkmark \qquad [k = 1, \text{ given recurrence}]$$

$$\begin{aligned} T(n) &= 2^{k-1} T(n/2^{k-1}) + (k-1)n && [\text{inductive hypothesis}] \\ &= 2^{k-1}\big(2T(n/2^k) + n/2^{k-1}\big) + (k-1)n && [\text{substitution}] \\ &= 2^k T(n/2^k) + kn \quad \checkmark && [\text{algebra}] \end{aligned}$$

Our guess was right! The recurrence becomes trivial when $n/2^k = 1$, or equivalently, when $k = \log_2 n$:

$$T(n) = nT(1) + n\log_2 n = n\log_2 n + n.$$

Finally, we have to put back the Θ's we stripped off; our final closed-form solution is $\boldsymbol{T(n) = \Theta(n \log n)}$.

2.4 An uglier divide-and-conquer example

Consider the divide-and-conquer recurrence $\boldsymbol{T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n}$. This doesn't fit into the form required by the Master Theorem (which we'll see below), but it still sort of resembles the Mergesort recurrence—the total size of the subproblems at the first level of recursion is n—so let's *guess* that $T(n) = O(n \log n)$, and then try to prove that our guess is correct. (We could also attack this recurrence by unrolling, but let's see how far just guessing will take us.)

Let's start by trying to prove an upper bound $T(n) \le a\, n \lg n$ for all sufficiently large n and some constant a to be determined later:

$$\begin{aligned} T(n) &= \sqrt{n} \cdot T(\sqrt{n}) + n \\ &\le \sqrt{n} \cdot a\sqrt{n} \lg \sqrt{n} + n && [\text{induction hypothesis}] \\ &= (a/2)n \lg n + n && [\text{algebra}] \\ &\le an \lg n \checkmark && [\text{algebra}] \end{aligned}$$

The last inequality assumes only that $1 \le (a/2)\log n$,or equivalently, that $n \ge 2^{2/a}$. In other words, the induction proof is correct if n is sufficiently large. So we were right!

But before you break out the champagne, what about the multiplicative constant a? The proof worked for *any* constant a, no matter how small. This strongly suggests that our upper bound $T(n) = O(n \log n)$ is not tight. Indeed, if we try to prove a matching lower bound $T(n) \ge b\, n \log n$ for sufficiently large n, we run into trouble.

$$\begin{aligned} T(n) &= \sqrt{n} \cdot T(\sqrt{n}) + n \\ &\ge \sqrt{n} \cdot b\sqrt{n} \log \sqrt{n} + n && [\text{induction hypothesis}] \\ &= (b/2)n \log n + n \\ &\not\ge bn \log n \end{aligned}$$

The last inequality would be correct only if $1 > (b/2)\log n$, but that inequality is false for large values of n, no matter which constant b we choose.

Okay, so $\Theta(n \log n)$ is too big. How about $\Theta(n)$? The lower bound is easy to prove directly:

$$T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n \geq n \checkmark$$

But an inductive proof of the upper bound fails.

$$\begin{aligned}
T(n) &= \sqrt{n} \cdot T(\sqrt{n}) + n \\
&\leq \sqrt{n} \cdot a\sqrt{n} + n && \text{[induction hypothesis]} \\
&= (a+1)n && \text{[algebra]} \\
&\not\leq an
\end{aligned}$$

Hmmm. So what's bigger than n and smaller than $n \lg n$? How about $n\sqrt{\lg n}$?

$$\begin{aligned}
T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n &\leq \sqrt{n} \cdot a\sqrt{n}\sqrt{\lg \sqrt{n}} + n && \text{[induction hypothesis]} \\
&= (a/\sqrt{2})\, n\sqrt{\lg n} + n && \text{[algebra]} \\
&\leq a\, n\sqrt{\lg n} \quad \text{for large enough } n \checkmark
\end{aligned}$$

Okay, the upper bound checks out; how about the lower bound?

$$\begin{aligned}
T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n &\geq \sqrt{n} \cdot b\sqrt{n}\sqrt{\lg \sqrt{n}} + n && \text{[induction hypothesis]} \\
&= (b/\sqrt{2})\, n\sqrt{\lg n} + n && \text{[algebra]} \\
&\not\geq b\, n\sqrt{\lg n}
\end{aligned}$$

No, the last step doesn't work. So $\Theta(n\sqrt{\lg n})$ doesn't work.

Okay... what else is between n and $n \lg n$? How about $n \lg\lg n$?

$$\begin{aligned}
T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n &\leq \sqrt{n} \cdot a\sqrt{n} \lg\lg \sqrt{n} + n && \text{[induction hypothesis]} \\
&= a\, n \lg\lg n - a\, n + n && \text{[algebra]} \\
&\leq a\, n \lg\lg n \qquad \textbf{if } \boldsymbol{a \geq 1} \checkmark
\end{aligned}$$

Hey look at that! For once, our upper bound proof requires a constraint on the hidden constant a. This is an good indication that we've found the right answer. Let's try the lower bound:

$$\begin{aligned}
T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n &\geq \sqrt{n} \cdot b\sqrt{n} \lg\lg \sqrt{n} + n && \text{[induction hypothesis]} \\
&= b\, n \lg\lg n - b\, n + n && \text{[algebra]} \\
&\geq b\, n \lg\lg n \qquad \textbf{if } \boldsymbol{b \leq 1} \checkmark
\end{aligned}$$

Hey, it worked! We have most of an inductive proof that $T(n) \leq an \lg\lg n$ for any $a \geq 1$ and most of an inductive proof that $T(n) \geq bn \lg\lg n$ for any $b \leq 1$. Technically, we're still missing the base cases in both proofs, but we can be fairly confident at this point that $\boldsymbol{T(n) = \Theta(n \log\log n)}$.

3 Divide and Conquer Recurrences (Recursion Trees)

Many divide and conquer algorithms give us running-time recurrences of the form

$$\boxed{T(n) = a\, T(n/b) + f(n)} \tag{1}$$

where a and b are constants and $f(n)$ is some other function. There is a simple and general technique for solving many recurrences in this and similar forms, using a ***recursion tree***. The root of the recursion tree is a box containing the value $f(n)$; the root has a children, each of which is the root of a (recursively defined) recursion tree for the function $T(n/b)$.

Equivalently, a recursion tree is a complete a-ary tree where each node at depth i contains the value $f(n/b^i)$. The recursion stops when we get to the base case(s) of the recurrence. Because we're only looking for asymptotic bounds, the exact base case doesn't matter; we can safely assume that $T(1) = \Theta(1)$, or even that $T(n) = \Theta(1)$ for all $n \leq 10^{100}$. I'll also assume for simplicity that n is an integral power of b; we'll see how to avoid this assumption later (but to summarize: it doesn't matter).

Now $T(n)$ is just the sum of all values stored in the recursion tree. For each i, the ith level of the tree contains a^i nodes, each with value $f(n/b^i)$. Thus,

$$\boxed{T(n) = \sum_{i=0}^{L} a^i f(n/b^i)} \tag{Σ}$$

where L is the depth of the recursion tree. We easily see that $\boldsymbol{L = \log_b n}$, because $n/b^L = 1$. The base case $f(1) = \Theta(1)$ implies that the last non-zero term in the summation is $\Theta(a^L) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$.

For *most* divide-and-conquer recurrences, the level-by-level sum (Σ) is a *geometric* series—each term is a constant factor larger or smaller than the previous term. In this case, only the largest term in the geometric series matters; all of the other terms are swallowed up by the $\Theta(\cdot)$ notation.

A recursion tree for the recurrence $T(n) = a\,T(n/b) + f(n)$

Here are several examples of the recursion-tree technique in action:

- **Mergesort (simplified): $\boldsymbol{T(n) = 2T(n/2) + n}$**

  There are 2^i nodes at level i, each with value $n/2^i$, so every term in the level-by-level sum (Σ) is the same:

  $$T(n) = \sum_{i=0}^{L} n.$$

  The recursion tree has $L = \log_2 n$ levels, so $\boldsymbol{T(n) = \Theta(n \log n)}$.

- **Randomized selection: $T(n) = T(3n/4) + n$**

  The recursion tree is a single path. The node at depth i has value $(3/4)^i n$, so the level-by-level sum (Σ) is a decreasing geometric series:

$$T(n) = \sum_{i=0}^{L} (3/4)^i n.$$

  This geometric series is dominated by its initial term n, so $\boldsymbol{T(n) = \Theta(n)}$. The recursion tree has $L = \log_{4/3} n$ levels, but so what?

- **Karatsuba's multiplication algorithm: $T(n) = 3T(n/2) + n$**

  There are 3^i nodes at depth i, each with value $n/2^i$, so the level-by-level sum (Σ) is an increasing geometric series:

$$T(n) = \sum_{i=0}^{L} (3/2)^i n.$$

  This geometric series is dominated by its final term $(3/2)^L n$. Each leaf contributes 1 to this term; thus, the final term is equal to the number of leaves in the tree! The recursion tree has $L = \log_2 n$ levels, and therefore $3^{\log_2 n} = n^{\log_2 3}$ leaves, so $\boldsymbol{T(n) = \Theta(n^{\log_2 3})}$.

- **$T(n) = 2T(n/2) + n/\lg n$**

  The sum of all the nodes in the ith level is $n/(\lg n - i)$. This implies that the depth of the tree is at most $\lg n - 1$. The level sums are neither constant nor a geometric series, so we just have to evaluate the overall sum directly.

  Recall (or if you're seeing this for the first time: Behold!) that the nth *harmonic number* H_n is the sum of the reciprocals of the first n positive integers:

$$H_n := \sum_{i=1}^{n} \frac{1}{i}$$

  It's not hard to show that $H_n = \Theta(\log n)$; in fact, we have the stronger inequalities $\ln(n+1) \le H_n \le \ln n + 1$.

$$T(n) = \sum_{i=0}^{\lg n - 1} \frac{n}{\lg n - i} = \sum_{j=1}^{\lg n} \frac{n}{j} = nH_{\lg n} = \boldsymbol{\Theta(n \lg\lg n)}$$

- **$T(n) = 4T(n/2) + n \lg n$**

  There are 4^i nodes at each level i, each with value $(n/2^i)\lg(n/2^i) = (n/2^i)(\lg n - i)$; again, the depth of the tree is at most $\lg n - 1$. We have the following summation:

$$T(n) = \sum_{i=0}^{\lg n - 1} n2^i(\lg n - i)$$

  We can simplify this sum by substituting $j = \lg n - i$:

$$T(n) = \sum_{j=i}^{\lg n} n2^{\lg n - j} j = \sum_{j=i}^{\lg n} \frac{n^2 j}{2^j} = n^2 \sum_{j=i}^{\lg n} \frac{j}{2^j} = \boldsymbol{\Theta(n^2)}$$

The last step uses the fact that $\sum_{i=1}^{\infty} j/2^j = 2$. Although this is not quite a geometric series, it is still dominated by its largest term.

- **Ugly divide and conquer: $T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n$**

  We solved this recurrence earlier by guessing the right answer and verifying, but we can use recursion trees to get the correct answer directly. The *degree* of the nodes in the recursion tree is no longer constant, so we have to be a bit more careful, but the same basic technique still applies. It's not hard to see that the nodes in any level sum to n. The depth L satisfies the identity $n^{2^{-L}} = 2$ (we can't get all the way down to 1 by taking square roots), so $L = \lg\lg n$ and $\boldsymbol{T(n) = \Theta(n \lg\lg n)}$.

- **Randomized quicksort: $T(n) = T(3n/4) + T(n/4) + n$**

  This recurrence isn't in the standard form described earlier, but we can still solve it using recursion trees. Now modes in the same level of the recursion tree have different values, and different leaves are at different levels. However, the nodes in any *complete* level (that is, above any of the leaves) sum to n. Moreover, every leaf in the recursion tree has depth between $\log_4 n$ and $\log_{4/3} n$. To derive an upper bound, we overestimate $T(n)$ by ignoring the base cases and extending the tree downward to the level of the *deepest* leaf. Similarly, to derive a lower bound, we overestimate $T(n)$ by counting only nodes in the tree up to the level of the *shallowest* leaf. These observations give us the upper and lower bounds $n \log_4 n \le T(n) \le n \log_{4/3} n$. Since these bounds differ by only a constant factor, we have $\boldsymbol{T(n) = \Theta(n \log n)}$.

- **Deterministic selection: $T(n) = T(n/5) + T(7n/10) + n$**

  Again, we have a lopsided recursion tree. If we look only at complete levels of the tree, we find that the level sums form a descending geometric series $T(n) = n + 9n/10 + 81n/100 + \cdots$. We can get an upper bound by ignoring the base cases entirely and growing the tree out to infinity, and we can get a lower bound by only counting nodes in complete levels. Either way, the geometric series is dominated by its largest term, so $\boldsymbol{T(n) = \Theta(n)}$.

- **Randomized search trees: $T(n) = \frac{1}{4}T(n/4) + \frac{3}{4}T(3n/4) + 1$**

  This looks like a divide-and-conquer recurrence, but what does it mean to have a quarter of a child? The right approach is to imagine that each node in the recursion tree has a *weight* in addition to its value. Alternately, we get a standard recursion tree again if we add a second real parameter to the recurrence, defining $T(n) = T(n, 1)$, where

  $$T(n, \alpha) = T(n/4, \alpha/4) + T(3n/4, 3\alpha/4) + \alpha.$$

  In each complete level of the tree, the (weighted) node values sum to exactly 1. The leaves of the recursion tree are at different levels, but all between $\log_4 n$ and $\log_{4/3} n$. So we have upper and lower bounds $\log_4 n \le T(n) \le \log_{4/3} n$, which differ by only a constant factor, so $\boldsymbol{T(n) = \Theta(\log n)}$.

- **Ham-sandwich trees: $T(n) = T(n/2) + T(n/4) + 1$**

  Again, we have a lopsided recursion tree. If we only look at complete levels, we find that the level sums form an *ascending* geometric series $T(n) = 1 + 2 + 4 + \cdots$, so the solution

is dominated by the number of leaves. The recursion tree has $\log_4 n$ complete levels, so there are more than $2^{\log_4 n} = n^{\log_4 2} = \sqrt{n}$; on the other hand, every leaf has depth at most $\log_2 n$, so the total number of leaves is at most $2^{\log_2 n} = n$. Unfortunately, the crude bounds $\boldsymbol{\sqrt{n} \ll T(n) \ll n}$ are the best we can derive using the techniques we know so far!

The following theorem completely describes the solution for any divide-and-conquer recurrence in the 'standard form' $T(n) = aT(n/b) + f(n)$, where a and b are constants and $f(n)$ is a polynomial. This theorem allows us to bypass recursion trees for "standard" divide-and-conquer recurrences, but many people (including Jeff) find it harder to even remember the statement of the theorem than to use the more powerful and general recursion-tree technique. Your mileage may vary.

The Master Theorem. *The recurrence $T(n) = aT(n/b) + f(n)$ can be solved as follows.*

- *If $a\,f(n/b) = \kappa\, f(n)$ for some constant $\kappa < 1$, then $T(n) = \Theta(f(n))$.*
- *If $a\,f(n/b) = K\, f(n)$ for some constant $K > 1$, then $T(n) = \Theta(n^{\log_b a})$.*
- *If $a\,f(n/b) = f(n)$, then $T(n) = \Theta(f(n)\log_b n)$.*
- *If none of these three cases apply, you're on your own.*

Proof: If $f(n)$ is a *constant factor larger* than $a\,f(b/n)$, then by induction, the level sums define a descending geometric series. The sum of any geometric series is a constant times its largest term. In this case, the largest term is the first term $f(n)$.

If $f(n)$ is a *constant factor smaller* than $a\,f(b/n)$, then by induction, the level sums define an ascending geometric series. The sum of any geometric series is a constant times its largest term. In this case, this is the last term, which by our earlier argument is $\Theta(n^{\log_b a})$.

Finally, if $a\,f(b/n) = f(n)$, then by induction, each of the $L+1$ terms in the sum is equal to $f(n)$, and the recursion tree has depth $L = \Theta(\log_b n)$. □

*4 The Nuclear Bomb

Finally, let me describe *without proof* a powerful generalization of the recursion tree method, first published by Lebanese researchers Mohamad Akra and Louay Bazzi in 1998. Consider a general divide-and-conquer recurrence of the form

$$T(n) = \sum_{i=1}^{k} a_i T(n/b_i) + f(n),$$

where k is a constant, $a_i > 0$ and $b_i > 1$ are constants for all i, and $f(n) = \Omega(n^c)$ and $f(n) = O(n^d)$ for some constants $0 < c \le d$. (As usual, we assume the standard base case $T(\Theta(1)) = \Theta(1)$.) Akra and Bazzi prove that this recurrence has the closed-form asymptotic solution

$$T(n) = \Theta\left(n^{\rho}\left(1 + \int_1^n \frac{f(u)}{u^{\rho+1}}\,du\right)\right),$$

where ρ is the unique real solution to the equation

$$\sum_{i=1}^{k} a_i / b_i^{\rho} = 1.$$

In particular, the Akra-Bazzi theorem immediately implies the following form of the Master Theorem:

$$T(n) = aT(n/b) + n^c \implies T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } c < \log_b a - \varepsilon \\ \Theta(n^c \log n) & \text{if } c = \log_b a \\ \Theta(n^c) & \text{if } c > \log_b a + \varepsilon \end{cases}$$

The Akra-Bazzi theorem does not require that the parameters a_i and b_i are integers, or even rationals; on the other hand, even when all parameters are integers, the characteristic equation $\sum_i a_i/b_i^\rho = 1$ may have no analytical solution.

Here are a few examples of recurrences that are difficult (or impossible) for recursion trees, but have easy solutions using the Akra-Bazzi theorem.

- **Randomized quicksort: $T(n) = T(3n/4) + T(n/4) + n$**

  The equation $(3/4)^\rho + (1/4)^\rho = 1$ has the unique solution $\rho = 1$, and therefore

  $$T(n) = \Theta\left(n\left(1 + \int_1^n \frac{1}{u}\,du\right)\right) = O(n \log n).$$

- **Deterministic selection: $T(n) = T(n/5) + T(7n/10) + n$**

  The equation $(1/5)^\rho + (7/10)^\rho = 1$ has no analytical solution. However, we easily observe that $(1/5)^x + (7/10)^x$ is a decreasing function of x, and therefore $0 < \rho < 1$. Thus, we have

  $$\int_1^n \frac{f(u)}{u^{\rho+1}}\,du = \int_1^n u^{-\rho}\,du = \left.\frac{u^{1-\rho}}{1-\rho}\right|_{u=1}^n = \frac{n^{1-\rho}-1}{1-\rho} = \Theta(n^{1-\rho}),$$

  and therefore

  $$T(n) = \Theta(n^\rho \cdot (1 + \Theta(n^{1-\rho})) = \Theta(n).$$

  (A bit of numerical computation gives the approximate value $\rho \approx 0.83978$, but why bother?)

- **Randomized search trees: $T(n) = \frac{1}{4}T(n/4) + \frac{3}{4}T(3n/4) + 1$**

  The equation $\frac{1}{4}(\frac{1}{4})^\rho + \frac{3}{4}(\frac{3}{4})^\rho = 1$ has the unique solution $\rho = 0$, and therefore

  $$T(n) = \Theta\left(1 + \int_1^n \frac{1}{u}\,du\right) = \Theta(\log n).$$

- **Ham-sandwich trees: $T(n) = T(n/2) + T(n/4) + 1$.** Recall that we could only prove the very weak bounds $\sqrt{n} \ll T(n) \ll n$ using recursion trees. The equation $(1/2)^\rho + (1/4)^\rho = 1$ has the unique solution $\rho = \log_2((1+\sqrt{5})/2) \approx 0.69424$, which can be obtained by setting $x = 2^\rho$ and solving for x. Thus, we have

  $$\int_1^n \frac{1}{u^{\rho+1}}\,du = \left.\frac{u^{-\rho}}{-\rho}\right|_{u=1}^n = \frac{1-n^{-\rho}}{\rho} = \Theta(1)$$

  and therefore

  $$T(n) = \Theta\left(n^\rho(1 + \Theta(1))\right) = \Theta(n^{\lg \phi}).$$

The Akra-Bazzi method is that it can solve *almost* any divide-and-conquer recurrence with just a few lines of calculation. (There are a few nasty exceptions like $T(n) = \sqrt{n}\,T(\sqrt{n}) + n$ where we have to fall back on recursion trees.) On the other hand, the steps appear to be magic, which makes the method hard to remember, and for most divide-and-conquer recurrences that arise in practice, there are much simpler solution techniques.

*5 Linear Recurrences (Annihilators)

Another common class of recurrences, called ***linear*** recurrences, arises in the context of recursive backtracking algorithms and counting problems. These recurrences express each function value $f(n)$ as a *linear* combination of a small number of nearby values $f(n-1), f(n-2), f(n-3), \ldots$. The Fibonacci recurrence is a typical example:

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

It turns out that the solution to *any* linear recurrence is a simple combination of polynomial and exponential functions in n. For example, we can verify by induction that the linear recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{if } n = 1 \text{ or } n = 2 \\ 3T(n-1) - 8T(n-2) + 4T(n-3) & \text{otherwise} \end{cases}$$

has the closed-form solution $\boldsymbol{T(n) = (n-3)2^n + 4}$. First we check the base cases:

$$\begin{aligned} T(0) &= (0-3)2^0 + 4 = 1 \quad \checkmark \\ T(1) &= (1-3)2^1 + 4 = 0 \quad \checkmark \\ T(2) &= (2-3)2^2 + 4 = 0 \quad \checkmark \end{aligned}$$

And now the recursive case:

$$\begin{aligned} T(n) &= 3T(n-1) - 8T(n-2) + 4T(n-3) \\ &= 3((n-4)2^{n-1} + 4) - 8((n-5)2^{n-2} + 4) + 4((n-6)2^{n-3} + 4) \\ &= \left(\frac{3}{2} - \frac{8}{4} + \frac{4}{8}\right) n \cdot 2^n - \left(\frac{12}{2} - \frac{40}{4} + \frac{24}{8}\right) 2^n + (2 - 8 + 4) \cdot 4 \\ &= (n-3) \cdot 2^n + 4 \quad \checkmark \end{aligned}$$

But how could we have possibly come up with that solution? In this section, I'll describe a general method for solving linear recurrences that's arguably easier than the induction proof!

5.1 Operators

Our technique for solving linear recurrences relies on the theory of ***operators***. Operators are higher-order functions, which take one or more functions as input and produce different functions as output. For example, your first two semesters of calculus focus almost exclusively on the *differential* and *integral* operators $\frac{d}{dx}$ and $\int dx$. All the operators we will need are combinations of three elementary building blocks:

- **Sum:** $(f+g)(n) := f(n)+g(n)$
- **Scale:** $(\alpha \cdot f)(n) := \alpha \cdot (f(n))$
- **Shift:** $(\boldsymbol{E}f)(n) := f(n+1)$

The shift and scale operators are ***linear***, which means they can be distributed over sums; for example, for any functions f, g, and h, we have $\boldsymbol{E}(f-3(g-h)) = \boldsymbol{E}f + (-3)\boldsymbol{E}g + 3\boldsymbol{E}h$.

We can combine these building blocks to obtain more complex *compound* operators. For example, the compound operator $\boldsymbol{E}-2$ is defined by setting $(\boldsymbol{E}-2)f := \boldsymbol{E}f + (-2)f$ for any function f. We can also apply the shift operator twice: $(\boldsymbol{E}(\boldsymbol{E}f))(n) = f(n+2)$; we write usually $\boldsymbol{E}^2 f$ as a synonym for $\boldsymbol{E}(\boldsymbol{E}f)$. More generally, for any positive integer k, the operator $\boldsymbol{E}^k$ shifts its argument k times: $\boldsymbol{E}^k f(n) = f(n+k)$. Similarly, $(\boldsymbol{E}-2)^2$ is shorthand for the operator $(\boldsymbol{E}-2)(\boldsymbol{E}-2)$, which applies $(\boldsymbol{E}-2)$ twice.

For example, here are the results of applying different operators to the exponential function $f(n) = 2^n$:

$$\begin{aligned}
2f(n) &= 2 \cdot 2^n = 2^{n+1} \\
3f(n) &= 3 \cdot 2^n \\
\boldsymbol{E}f(n) &= 2^{n+1} \\
\boldsymbol{E}^2 f(n) &= 2^{n+2} \\
(\boldsymbol{E}-2)f(n) &= \boldsymbol{E}f(n) - 2f(n) = 2^{n+1} - 2^{n+1} = 0 \\
(\boldsymbol{E}^2-1)f(n) &= \boldsymbol{E}^2 f(n) - f(n) = 2^{n+2} - 2^n = 3 \cdot 2^n
\end{aligned}$$

These compound operators can be manipulated exactly as though they were polynomials over the "variable" $\boldsymbol{E}$. In particular, we can factor compound operators into "products" of simpler operators, which can be applied in any order. For example, the compound operators $\boldsymbol{E}^2 - 3\boldsymbol{E} + 2$ and $(\boldsymbol{E}-1)(\boldsymbol{E}-2)$ are equivalent:

$$\text{Let} \quad g(n) := (\boldsymbol{E}-2)f(n) = f(n+1) - 2f(n).$$

$$\begin{aligned}
\text{Then} \quad (\boldsymbol{E}-1)(\boldsymbol{E}-2)f(n) &= (\boldsymbol{E}-1)g(n) \\
&= g(n+1) - g(n) \\
&= (f(n+2) - 2f(n-1)) - (f(n+1) - 2f(n)) \\
&= f(n+2) - 3f(n+1) + 2f(n) \\
&= (\boldsymbol{E}^2 - 3\boldsymbol{E} + 2)f(n). \quad \checkmark
\end{aligned}$$

It is an easy exercise to confirm that $\boldsymbol{E}^2 - 3\boldsymbol{E} + 2$ is also equivalent to the operator $(\boldsymbol{E}-2)(\boldsymbol{E}-1)$.

The following table summarizes everything we need to remember about operators.

Operator	**Definition**
addition	$(f+g)(n) := f(n)+g(n)$
subtraction	$(f-g)(n) := f(n)-g(n)$
multiplication	$(\alpha \cdot f)(n) := \alpha \cdot (f(n))$
shift	$\boldsymbol{E}f(n) := f(n+1)$
k-fold shift	$\boldsymbol{E}^k f(n) := f(n+k)$
composition	$(\boldsymbol{X}+\boldsymbol{Y})f := \boldsymbol{X}f + \boldsymbol{Y}f$
	$(\boldsymbol{X}-\boldsymbol{Y})f := \boldsymbol{X}f - \boldsymbol{Y}f$
	$\boldsymbol{X}\boldsymbol{Y}f := \boldsymbol{X}(\boldsymbol{Y}f) = \boldsymbol{Y}(\boldsymbol{X}f)$
distribution	$\boldsymbol{X}(f+g) = \boldsymbol{X}f + \boldsymbol{X}g$

5.2 Annihilators

An ***annihilator*** of a function f is any nontrivial operator that transforms f into the zero function. (We can trivially annihilate any function by multiplying it by zero, so as a technical matter, we do not consider the zero operator to be an annihilator.) Every compound operator we consider annihilates a specific class of functions; conversely, every function composed of polynomial and exponential functions has a unique (minimal) annihilator.

We have already seen that the operator $(\boldsymbol{E}-2)$ annihilates the function 2^n. It's not hard to see that the operator $(\boldsymbol{E}-c)$ annihilates the function $\alpha \cdot c^n$, for any constants c and α. More generally, the operator $(\boldsymbol{E}-c)$ annihilates the function a^n if and only if $c = a$:

$$(\boldsymbol{E}-c)a^n = \boldsymbol{E}a^n - c \cdot a^n = a^{n+1} - c \cdot a^n = (a-c)a^n.$$

Thus, $(\boldsymbol{E}-2)$ is essentially the *only* annihilator of the function 2^n.

What about the function $2^n + 3^n$? The operator $(\boldsymbol{E}-2)$ annihilates the function 2^n, but leaves the function 3^n unchanged. Similarly, $(\boldsymbol{E}-3)$ annihilates 3^n while *negating* the function 2^n. But if we apply *both* operators, we annihilate both terms:

$$\begin{aligned}(\boldsymbol{E}-2)(2^n+3^n) &= \boldsymbol{E}(2^n+3^n) - 2(2^n+3^n)\\ &= (2^{n+1}+3^{n+1}) - (2^{n+1}+2\cdot 3^n) = 3^n\end{aligned}$$

$$\implies (\boldsymbol{E}-3)(\boldsymbol{E}-2)(2^n+3^n) = (\boldsymbol{E}-3)3^n = 0$$

In general, for any integers $a \neq b$, the operator $(\boldsymbol{E}-a)(\boldsymbol{E}-b) = (\boldsymbol{E}-b)(\boldsymbol{E}-a) = (\boldsymbol{E}^2-(a+b)\boldsymbol{E}+ab)$ annihilates any function of the form $\alpha a^n + \beta b^n$, but nothing else.

What about the operator $(\boldsymbol{E}-a)(\boldsymbol{E}-a) = (\boldsymbol{E}-a)^2$? It turns out that this operator annihilates all functions of the form $(\alpha n + \beta)a^n$:

$$\begin{aligned}(\boldsymbol{E}-a)((\alpha n+\beta)a^n) &= (\alpha(n+1)+\beta)a^{n+1} - a(\alpha n+\beta)a^n\\ &= \alpha a^{n+1}\end{aligned}$$

$$\implies (\boldsymbol{E}-a)^2((\alpha n+\beta)a^n) = (\boldsymbol{E}-a)(\alpha a^{n+1}) = 0$$

More generally, the operator $(\boldsymbol{E}-a)^d$ annihilates all functions of the form $p(n)\cdot a^n$, where $p(n)$ is a polynomial of degree at most $d-1$. For example, $(\boldsymbol{E}-1)^3$ annihilates any polynomial of degree at most 2.

The following table summarizes everything we need to remember about annihilators.

Operator	Functions annihilated	
$\boldsymbol{E}-1$	α	
$\boldsymbol{E}-a$	αa^n	
$(\boldsymbol{E}-a)(\boldsymbol{E}-b)$	$\alpha a^n + \beta b^n$	[if $a \neq b$]
$(\boldsymbol{E}-a_0)(\boldsymbol{E}-a_1)\cdots(\boldsymbol{E}-a_k)$	$\sum_{i=0}^{k} \alpha_i a_i^n$	[if a_i distinct]
$(\boldsymbol{E}-1)^2$	$\alpha n + \beta$	
$(\boldsymbol{E}-a)^2$	$(\alpha n+\beta)a^n$	
$(\boldsymbol{E}-a)^2(\boldsymbol{E}-b)$	$(\alpha n+\beta)a^b + \gamma b^n$	[if $a \neq b$]
$(\boldsymbol{E}-a)^d$	$\left(\sum_{i=0}^{d-1} \alpha_i n^i\right)a^n$	

If $\boldsymbol{X}$ annihilates f, then $\boldsymbol{X}$ also annihilates $\boldsymbol{E}f$.
If $\boldsymbol{X}$ annihilates both f and g, then $\boldsymbol{X}$ also annihilates $f \pm g$.
If $\boldsymbol{X}$ annihilates f, then $\boldsymbol{X}$ also annihilates αf, for any constant α.

If $\boldsymbol{X}$ annihilates f and $\boldsymbol{Y}$ annihilates g, then $\boldsymbol{XY}$ annihilates $f \pm g$.

5.3 Annihilating Recurrences

Given a linear recurrence for a function, it's easy to extract an annihilator for that function. For many recurrences, we only need to rewrite the recurrence in operator notation. Once we have an annihilator, we can factor it into operators of the form $(\boldsymbol{E}-c)$; the table on the previous page then gives us a generic solution with some unknown coefficients. If we are given explicit base cases, we can determine the coefficients by examining a few small cases; in general, this involves solving a small system of linear equations. If the base cases are not specified, the generic solution almost always gives us an asymptotic solution. Here is the technique step by step:

1. Write the recurrence in operator form
2. Extract an annihilator for the recurrence
3. Factor the annihilator (if necessary)
4. Extract the *generic solution* from the annihilator
5. Solve for coefficients using base cases (if known)

Here are several examples of the technique in action:

- $\boldsymbol{r(n) = 5r(n-1)}$***, where*** $\boldsymbol{r(0) = 3}$**.**

  1. We can write the recurrence in operator form as follows:

  $$r(n) = 5r(n-1) \implies r(n+1) - 5r(n) = 0 \implies (\boldsymbol{E}-5)r(n) = 0.$$

  2. We immediately see that $(\boldsymbol{E}-5)$ annihilates the function $r(n)$.
  3. The annihilator $(\boldsymbol{E}-5)$ is already factored.
  4. Consulting the annihilator table on the previous page, we find the generic solution $r(n) = \alpha 5^n$ for some constant α.
  5. The base case $r(0) = 3$ implies that $\alpha = 3$.

  We conclude that $\boldsymbol{r(n) = 3 \cdot 5^n}$. We can easily verify this closed-form solution by induction:

  $$\begin{aligned} r(0) &= 3 \cdot 5^0 = 3 \quad \checkmark && \text{[definition]} \\ r(n) &= 5r(n-1) && \text{[definition]} \\ &= 5 \cdot (3 \cdot 5^{n-1}) && \text{[induction hypothesis]} \\ &= 5^n \cdot 3 \quad \checkmark && \text{[algebra]} \end{aligned}$$

- **Fibonacci numbers:** $\boldsymbol{F(n) = F(n-1) + F(n-2)}$**, where** $\boldsymbol{F(0) = 0}$ **and** $\boldsymbol{F(1) = 1}$**.**

  1. We can rewrite the recurrence as $(\boldsymbol{E}^2 - \boldsymbol{E} - 1)F(n) = 0$.
  2. The operator $\boldsymbol{E}^2 - \boldsymbol{E} - 1$ clearly annihilates $F(n)$.
  3. The quadratic formula implies that the annihilator $\boldsymbol{E}^2 - \boldsymbol{E} - 1$ factors into $(\boldsymbol{E} - \phi)(\boldsymbol{E} - \hat{\phi})$, where $\phi = (1+\sqrt{5})/2 \approx 1.618034$ is the golden ratio and $\hat{\phi} = (1-\sqrt{5})/2 = 1 - \phi = -1/\phi \approx -0.618034$.
  4. The annihilator implies that $\boldsymbol{F(n) = \alpha\phi^n + \hat{\alpha}\hat{\phi}^n}$ for some unknown constants α and $\hat{\alpha}$.

5. The base cases give us two equations in two unknowns:

$$F(0) = 0 = \alpha + \hat{\alpha}$$
$$F(1) = 1 = \alpha\phi + \hat{\alpha}\hat{\phi}$$

Solving this system of equations gives us $\alpha = 1/(2\phi - 1) = 1/\sqrt{5}$ and $\hat{\alpha} = -1/\sqrt{5}$.

We conclude with the following exact closed form for the nth Fibonacci number:

$$\boldsymbol{F(n) = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}} = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n}$$

With all the square roots in this formula, it's quite amazing that Fibonacci numbers are integers. However, if we do all the math correctly, all the square roots cancel out when i is an integer. (In fact, this is pretty easy to prove using the binomial theorem.)

- **Towers of Hanoi: $T(n) = 2T(n-1) + 1$, where $T(0) = 0$.** This is our first example of a *non-homogeneous* recurrence, which means the recurrence has one or more non-recursive terms.

  1. We can rewrite the recurrence as $(E - 2)T(n) = 1$.
  2. The operator $(E - 2)$ doesn't quite annihilate the function; it leaves a *residue* of 1. But we can annihilate the residue by applying the operator $(E - 1)$. Thus, the compound operator $(E - 1)(E - 2)$ annihilates the function.
  3. The annihilator is already factored.
  4. The annihilator table gives us the generic solution $T(n) = \alpha 2^n + \beta$ for some unknown constants α and β.
  5. The base cases give us $T(0) = 0 = \alpha 2^0 + \beta$ and $T(1) = 1 = \alpha 2^1 + \beta$. Solving this system of equations, we find that $\alpha = 1$ and $\beta = -1$.

  We conclude that $\boldsymbol{T(n) = 2^n - 1}$.

For the remaining examples, I won't explicitly enumerate the steps in the solution.

- **Height-balanced trees: $H(n) = H(n-1) + H(n-2) + 1$, where $H(-1) = 0$ and $H(0) = 1$.** (Yes, we're starting at -1 instead of 0. So what?)

  We can rewrite the recurrence as $(E^2 - E - 1)H = 1$. The residue 1 is annihilated by $(E - 1)$, so the compound operator $(E - 1)(E^2 - E - 1)$ annihilates the recurrence. This operator factors into $(E - 1)(E - \phi)(E - \hat{\phi})$, where $\phi = (1 + \sqrt{5})/2$ and $\hat{\phi} = (1 - \sqrt{5})/2$. Thus, we get the generic solution $H(n) = \alpha \cdot \phi^n + \beta + \gamma \cdot \hat{\phi}^n$, for some unknown constants α, β, γ that satisfy the following system of equations:

$$\begin{aligned} H(-1) = 0 = \alpha\phi^{-1} + \beta + \gamma\hat{\phi}^{-1} &= \alpha/\phi + \beta - \gamma/\hat{\phi} \\ H(0) = 1 = \alpha\phi^0 + \beta + \gamma\hat{\phi}^0 &= \alpha + \beta + \gamma \\ H(1) = 2 = \alpha\phi^1 + \beta + \gamma\hat{\phi}^1 &= \alpha\phi + \beta + \gamma\hat{\phi} \end{aligned}$$

  Solving this system (using Cramer's rule or Gaussian elimination), we find that $\alpha = (\sqrt{5} + 2)/\sqrt{5}$, $\beta = -1$, and $\gamma = (\sqrt{5} - 2)/\sqrt{5}$. We conclude that

$$\boldsymbol{H(n) = \frac{\sqrt{5}+2}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - 1 + \frac{\sqrt{5}-2}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n.}$$

- $\boldsymbol{T(n) = 3T(n-1) - 8T(n-2) + 4T(n-3)}$**, where** $\boldsymbol{T(0) = 1}$**,** $\boldsymbol{T(1) = 0}$**, and** $\boldsymbol{T(2) = 0}$**.** This was our original example of a linear recurrence.

  We can rewrite the recurrence as $(E^3 - 3E^2 + 8E - 4)T = 0$, so we immediately have an annihilator $E^3 - 3E^2 + 8E - 4$. Using high-school algebra, we can factor the annihilator into $(\boldsymbol{E} - 2)^2(\boldsymbol{E} - 1)$, which implies the generic solution $T(n) = \alpha n 2^n + \beta 2^n + \gamma$. The constants α, β, and γ are determined by the base cases:

$$\begin{aligned} T(0) &= 1 = \alpha \cdot 0 \cdot 2^0 + \beta 2^0 + \gamma = \beta + \gamma \\ T(1) &= 0 = \alpha \cdot 1 \cdot 2^1 + \beta 2^1 + \gamma = 2\alpha + 2\beta + \gamma \\ T(2) &= 0 = \alpha \cdot 2 \cdot 2^2 + \beta 2^2 + \gamma = 8\alpha + 4\beta + \gamma \end{aligned}$$

  Solving this system of equations, we find that $\alpha = 1$, $\beta = -3$, and $\gamma = 4$, so $\boldsymbol{T(n) = (n-3)2^n + 4}$.

- $\boldsymbol{T(n) = T(n-1) + 2T(n-2) + 2^n - n^2}$

  We can rewrite the recurrence as $(\boldsymbol{E}^2 - \boldsymbol{E} - 2)T(n) = \boldsymbol{E}^2(2^n - n^2)$. Notice that we had to shift up the non-recursive parts of the recurrence when we expressed it in this form. The operator $(\boldsymbol{E} - 2)(\boldsymbol{E} - 1)^3$ annihilates the residue $2^n - n^2$, and therefore also annihilates the shifted residue $\boldsymbol{E}^2(2^n + n^2)$. Thus, the operator $(\boldsymbol{E} - 2)(\boldsymbol{E} - 1)^3(\boldsymbol{E}^2 - \boldsymbol{E} - 2)$ annihilates the entire recurrence. We can factor the quadratic factor into $(\boldsymbol{E} - 2)(E + 1)$, so the annihilator factors into $(\boldsymbol{E} - 2)^2(\boldsymbol{E} - 1)^3(\boldsymbol{E} + 1)$. So the generic solution is $\boldsymbol{T(n) = \alpha n 2^n + \beta 2^n + \gamma n^2 + \delta n + \varepsilon + \eta(-1)^n}$. The coefficients α, β, γ, δ, ε, η satisfy a system of six equations determined by the first six function values $T(0)$ through $T(5)$. For almost[2] every set of base cases, we have $\alpha \neq 0$, which implies that $\boldsymbol{T(n) = \Theta(n2^n)}$.

For a more detailed explanation of the annihilator method, see George Lueker, Some techniques for solving recurrences, *ACM Computing Surveys* 12(4):419-436, 1980.

6 Transformations

Sometimes we encounter recurrences that don't fit the structures required for recursion trees or annihilators. In many of those cases, we can transform the recurrence into a more familiar form, by defining a new function in terms of the one we want to solve. There are many different kinds of transformations, but these three are probably the most useful:

- **Domain transformation:** Define a new function $S(n) = T(f(n))$ with a simpler recurrence, for some simple function f.
- **Range transformation:** Define a new function $S(n) = f(T(n))$ with a simpler recurrence, for some simple function f.
- **Difference transformation:** Simplify the recurrence for $T(n)$ by considering the difference function $\Delta T(n) = T(n) - T(n-1)$.

Here are some examples of these transformations in action.

[2] In fact, the only possible solutions with $\alpha = 0$ have the form $-2^{n-1} - n^2/2 - 5n/2 + \eta(-1)^n$ for some constant η.

- **Unsimplified Mergesort: $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$**

  When n is a power of 2, we can simplify the mergesort recurrence to $T(n) = 2T(n/2) + \Theta(n)$, which has the solution $T(n) = \Theta(n \log n)$. Unfortunately, for other values values of n, this simplified recurrence is incorrect. When n is odd, then the recurrence calls for us to sort a fractional number of elements! Worse yet, if n is not a power of 2, we will *never* reach the base case $T(1) = 1$.

  So we really need to solve the original recurrence. We have no hope of getting an *exact* solution, even if we ignore the $\Theta(\,)$ in the recurrence; the floors and ceilings will eventually kill us. But we can derive a tight asymptotic solution using a domain transformation—we can rewrite the function $T(n)$ as a nested function $S(f(n))$, where $f(n)$ is a simple function and the function $S(\,)$ has an simpler recurrence.

  First let's overestimate the time bound, once by pretending that the two subproblem sizes are equal, and again to eliminate the ceiling:

  $$T(n) \le 2T\big(\lceil n/2 \rceil\big) + n \le 2T(n/2 + 1) + n.$$

  Now we define a new function $S(n) = T(n + \alpha)$, where α is a unknown constant, chosen so that $S(n)$ satisfies the Master-Theorem-ready recurrence $S(n) \le 2S(n/2) + O(n)$. To figure out the correct value of α, we compare two versions of the recurrence for the function $T(n + \alpha)$:

  $$\begin{aligned} S(n) \le 2S(n/2) + O(n) \quad &\Longrightarrow \quad T(n+\alpha) \le 2T(n/2 + \alpha) + O(n) \\ T(n) \le 2T(n/2 + 1) + n \quad &\Longrightarrow \quad T(n+\alpha) \le 2T((n+\alpha)/2 + 1) + n + \alpha \end{aligned}$$

  For these two recurrences to be equal, we need $n/2 + \alpha = (n + \alpha)/2 + 1$, which implies that $\alpha = 2$. The Master Theorem now tells us that $S(n) = O(n \log n)$, so

  $$T(n) = S(n-2) = O((n-2)\log(n-2)) = O(n \log n).$$

  A similar argument implies the matching lower bound $T(n) = \Omega(n \log n)$. So $\boldsymbol{T(n) = \Theta(n \log n)}$ after all, just as though we had ignored the floors and ceilings from the beginning!

  Domain transformations are useful for removing floors, ceilings, and lower order terms from the arguments of any recurrence that otherwise looks like it ought to fit either the Master Theorem or the recursion tree method. But now that we know this, we don't need to bother grinding through the actual gory details!

- **Ham-Sandwich Trees: $T(n) = T(n/2) + T(n/4) + 1$**

  As we saw earlier, the recursion tree method only gives us the uselessly loose bounds $\sqrt{n} \ll T(n) \ll n$ for this recurrence, and the recurrence is in the wrong form for annihilators. The authors who discovered ham-sandwich trees (Yes, this is a real data structure!) solved this recurrence by guessing the solution and giving a complicated induction proof. We got a tight solution using the Akra-Bazzi method, but who can remember that?

  In fact, a simple domain transformation allows us to solve the recurrence in just a few lines. We define a new function $t(k) = T(2^k)$, which satisfies the simpler linear recurrence $t(k) = t(k-1) + t(k-2) + 1$. This recurrence should immediately remind you of Fibonacci

numbers. Sure enough, the annihilator method implies the solution $t(k) = \Theta(\phi^k)$, where $\phi = (1+\sqrt{5})/2$ is the golden ratio. We conclude that

$$T(n) = t(\lg n) = \Theta(\phi^{\lg n}) = \boldsymbol{\Theta(n^{\lg \phi})} \approx \Theta(n^{0.69424}).$$

This is the same solution we obtained earlier using the Akra-Bazzi theorem.

Many other divide-and-conquer recurrences can be similarly transformed into linear recurrences and then solved with annihilators. Consider once more the simplified mergesort recurrence $T(n) = 2T(n/2) + n$. The function $t(k) = T(2^k)$ satisfies the recurrence $t(k) = 2t(k-1) + 2^k$. The annihilator method gives us the generic solution $t(k) = \Theta(k \cdot 2^k)$, which implies that $T(n) = t(\lg n) = \Theta(n \log n)$, just as we expected.

On the other hand, for some recurrences like $T(n) = T(n/3) + T(2n/3) + n$, the recursion tree method gives an easy solution, but there's no way to transform the recurrence into a form where we can apply the annihilator method directly.[3]

- **Random Binary Search Trees: $T(n) = \frac{1}{4}T(n/4) + \frac{3}{4}T(3n/4) + 1$**

  This looks like a divide-and-conquer recurrence, so we might be tempted to apply recursion trees, but what does it mean to have a quarter of a child? If we're not comfortable with weighted recursion trees (or the Akra-Bazzi theorem), we can instead apply the following range transformation. The function $U(n) = n \cdot T(n)$ satisfies the more palatable recurrence $U(n) = U(n/4) + U(3n/4) + n$. As we've already seen, recursion trees imply that $U(n) = \Theta(n \log n)$, which immediately implies that $\boldsymbol{T(n) = \Theta(\log n)}$.

- **Randomized Quicksort: $T(n) = \dfrac{2}{n}\displaystyle\sum_{k=0}^{n-1} T(k) + n$**

  This is our first example of a ***full history*** recurrence; each function value $T(n)$ is defined in terms of *all* previous function values $T(k)$ with $k < n$. Before we can apply any of our existing techniques, we need to convert this recurrence into an equivalent *limited history* form by shifting and subtracting away common terms. To make this step slightly easier, we first multiply both sides of the recurrence by n to get rid of the fractions.

$$n \cdot T(n) = 2\sum_{k=0}^{n-1} T(j) + n^2 \qquad \text{[multiply both sides by } n\text{]}$$

$$(n-1) \cdot T(n-1) = 2\sum_{k=0}^{n-2} T(j) + (n-1)^2 \qquad \text{[shift]}$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n - 1 \qquad \text{[subtract]}$$

$$T(n) = \frac{n+1}{n}T(n-1) + 2 - \frac{1}{n} \qquad \text{[simplify]}$$

[3] However, we can still get a solution via functional transformations as follows. The function $t(k) = T((3/2)^k)$ satisfies the recurrence $t(n) = t(n-1) + t(n-\lambda) + (3/2)^k$, where $\lambda = \log_{3/2} 3 = 2.709511\ldots$. The ***characteristic function*** for this recurrence is $(r^\lambda - r^{\lambda-1} - 1)(r - 3/2)$, which has a double root at $r = 3/2$ and nowhere else. Thus, $t(k) = \Theta(k(3/2)^k)$, which implies that $T(n) = t(\log_{3/2} n) = \Theta(n \log n)$. This line of reasoning is the core of the Akra-Bazzi method.

We can solve this limited-history recurrence using another functional transformation. We define a new function $t(n) = T(n)/(n+1)$, which satisfies the simpler recurrence

$$t(n) = t(n-1) + \frac{2}{n+1} - \frac{1}{n(n+1)},$$

which we can easily unroll into a summation. If we only want an asymptotic solution, we can simplify the final recurrence to $t(n) = t(n-1) + \Theta(1/n)$, which unrolls into a very familiar summation:

$$t(n) = \sum_{i=1}^{n} \Theta(1/i) = \Theta(H_n) = \Theta(\log n).$$

Finally, substituting $T(n) = (n+1)t(n)$ gives us a solution to the original recurrence: $\boldsymbol{T(n) = \Theta(n \log n)}$.

Exercises

1. For each of the following recurrences, first ***guess*** an exact closed-form solution, and then prove your guess is correct. You are free to use any method you want to make your guess—unrolling the recurrence, writing out the first several values, induction proof template, recursion trees, annihilators, transformations, 'It looks like that other one', whatever—but please describe your method. All functions are from the non-negative integers to the reals. If it simplifies your solutions, express them in terms of Fibonacci numbers F_n, harmonic numbers H_n, binomial coefficients $\binom{n}{k}$, factorials $n!$, and/or the floor and ceiling functions $\lfloor x \rfloor$ and $\lceil x \rceil$.

   (a) $A(n) = A(n-1) + 1$, where $A(0) = 0$.

   (b) $B(n) = \begin{cases} 0 & \text{if } n < 5 \\ B(n-5) + 2 & \text{otherwise} \end{cases}$

   (c) $C(n) = C(n-1) + 2n - 1$, where $C(0) = 0$.

   (d) $D(n) = D(n-1) + \binom{n}{2}$, where $D(0) = 0$.

   (e) $E(n) = E(n-1) + 2^n$, where $E(0) = 0$.

   (f) $F(n) = 3 \cdot F(n-1)$, where $F(0) = 1$.

   (g) $G(n) = \frac{G(n-1)}{G(n-2)}$, where $G(0) = 1$ and $G(1) = 2$. *[Hint: This is easier than it looks.]*

   (h) $H(n) = H(n-1) + 1/n$, where $H(0) = 0$.

   (i) $I(n) = I(n-2) + 3/n$, where $I(0) = I(1) = 0$. *[Hint: Consider even and odd n separately.]*

   (j) $J(n) = J(n-1)^2$, where $J(0) = 2$.

   (k) $K(n) = K(\lfloor n/2 \rfloor) + 1$, where $K(0) = 0$.

   (l) $L(n) = L(n-1) + L(n-2)$, where $L(0) = 2$ and $L(1) = 1$.
   [Hint: Write the solution in terms of Fibonacci numbers.]

   (m) $M(n) = M(n-1) \cdot M(n-2)$, where $M(0) = 2$ and $M(1) = 1$.
   [Hint: Write the solution in terms of Fibonacci numbers.]

(n) $N(n) = 1 + \sum_{k=1}^{n} (N(k-1) + N(n-k))$, where $N(0) = 1$.

(p) $P(n) = \sum_{k=0}^{n-1} (k \cdot P(k-1))$, where $P(0) = 1$.

(q) $Q(n) = \frac{1}{2-Q(n-1)}$, where $Q(0) = 0$.

(r) $R(n) = \max_{1 \le k \le n} \{R(k-1) + R(n-k) + n\}$

(s) $S(n) = \max_{1 \le k \le n} \{S(k-1) + S(n-k) + 1\}$

(t) $T(n) = \min_{1 \le k \le n} \{T(k-1) + T(n-k) + n\}$

(u) $U(n) = \min_{1 \le k \le n} \{U(k-1) + U(n-k) + 1\}$

(v) $V(n) = \max_{n/3 \le k \le 2n/3} \{V(k-1) + V(n-k) + n\}$

2. Use recursion trees or the Akra-Bazzi theorem to solve each of the following recurrences.

(a) $A(n) = 2A(n/4) + \sqrt{n}$

(b) $B(n) = 2B(n/4) + n$

(c) $C(n) = 2C(n/4) + n^2$

(d) $D(n) = 3D(n/3) + \sqrt{n}$

(e) $E(n) = 3E(n/3) + n$

(f) $F(n) = 3F(n/3) + n^2$

(g) $G(n) = 4G(n/2) + \sqrt{n}$

(h) $H(n) = 4H(n/2) + n$

(i) $I(n) = 4I(n/2) + n^2$

(j) $J(n) = J(n/2) + J(n/3) + J(n/6) + n$

(k) $K(n) = K(n/2) + K(n/3) + K(n/6) + n^2$

(l) $L(n) = L(n/15) + L(n/10) + 2L(n/6) + \sqrt{n}$

*(m) $M(n) = 2M(n/3) + 2M(2n/3) + n$

(n) $N(n) = \sqrt{2n}\, N(\sqrt{2n}) + \sqrt{n}$

(p) $P(n) = \sqrt{2n}\, P(\sqrt{2n}) + n$

(q) $Q(n) = \sqrt{2n}\, Q(\sqrt{2n}) + n^2$

(r) $R(n) = R(n-3) + 8^n$ — Don't use annihilators!

(s) $S(n) = 2S(n-2) + 4^n$ — Don't use annihilators!

(t) $T(n) = 4T(n-1) + 2^n$ — Don't use annihilators!

3. Make up a bunch of linear recurrences and then solve them using annihilators.

4. Solve the following recurrences, using any tricks at your disposal.

   (a) $T(n) = \sum_{i=1}^{\lg n} T(n/2^i) + n$ *[Hint: Assume n is a power of 2.]*

   (b) More to come...

© Copyright 2018 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edi/teaching/algorithms/ for the most recent revision.