N and n Relations

, N = 2n – 1, Largest num with n bits can be rep. By

Addition and Multiplication Costs:

loop w/ add O(N log(N)) or O(n), Mult: O( ) or O()

Adding or Multiplying Number Sizes

Add two n-bit nums = n+1 bit num max Mult two n-bit nums = 2n bit num max

Big – Oh, Big – Theta, and Big - Omega

f = , g = 🡪 f = Ω(g) f =, g = 3n 🡪 f = Θ(g)

f = 3, g = 🡪 f = O(g) f = , g = 🡪 f = Θ(g)

Some Helpful Equations

1 + 2 + … + n = r ≠ 1 🡪

r > 1 🡪 f(i) = 🡪 f(i) = Θ() by = x 🡪 logbx = y

0 < r < 1 🡪 f(i) = 🡪 f(i) = Θ(1) FibN ~ 20.694N ~ logbN bit to rep N

Everything Modular

x ≡ z mod N implies N | (x − z) 13 mod 7 = 6 100 mod 9 = 1 100 mod 20 = 0 -21 mod 10 = 9 -3 mod 12 = 9 0 mod 6 = 6, 12, …

(a/b)(modN) exists if and only if GCD(b, N) = 1

20/15 (mod 50) 🡪 undefined 109/19 (mod 115) 🡪 defined

Fermat: ≡ 1 (mod p). mod 53 = 1

(25 + 7 · 51) mod 53 =

(25mod 53)·( 25mod 53) + (7 mod 53)·( mod 53) · ( mod 53) = 8

Modular Inverse

6/5 mod 49 🡪 GCD(5, 49) = 1 🡪 extended-euclid(49, 5)

d = 49x + 5y

49 = 5(9) + 4 1 = 5 + (49 + 5(-9))(-1) 1 = 49(-1) + 5(10)

5 = 4(1) + 1 1 = 5 + 4(-1)

4 = 1(4) + 0

so…d = 1, x = -1, y = 10 so…10 is the modular inverse of 5 mod 49

6/5 mod 49 = 6 · 10 mod 49 = 11

Random Primes p of rand prime = 1.44/n, trying to find x primes in n nums, expect tries

RSA

RSA. p, q are prime. N = p\*q. GCD(e, (p-1)\*(q-1)) = 1. d = inverse of e mod (p-1)\*(q-1).

N, e are public. Anyone can encrypt a message x with . Person with d decrypts with

P = 5, q = 11, N = 55. (p-1)\*(q-1) = 4\*10 = 40. e = 3 (it’s usually 3).

Mult inverse of 3 mod 40 = d = 27. Can confirm with 27\*3 = 1 mod 40

Send x = 9. . Decrypt:

Define T(n)

T(n) = # rec calls \* T(size of new input) + O(local work)

Master Theorem

if: a > 0, b > 1, d ≥ 0

O() if d > O( if d = O() if d <

Master Theorem Examples (note that the d and flip sides here)

T(n) = T(n/2) + O(1) a = 1, b = 2, d = 0. so O() = O(log(n))

T(n) = 3\*T(n/2) + O(n2) a = 3, b = 2, d = 2. so O() = O()

T(n) = 3\*T(n/2) + O(n) a = 3, b = 2, d = 1. so O(= O()

Solving Exact Recurrences

T(n) = 2T(n/3) +1;  T(1) = 1

number of nodes at Level i is

the value in the nodes at Level i:

bottom nodes have the value 1 (since the base case is n = 1)

Solving for i, we get that the bottom level is Level

In levels above bottom, local contribution at each node is 1 (from the +1 term in the recurrence)

local contribution at the nodes in the bottom level is also 1, because T(1) = 1.

total contribution at level i is × 1 =

Summing the total contribution at each level is

So T(n) = = = =

T(n) = 2T(n/3);  T(1) = 2

Same as above, but now no local work in nodes above the bottom level. And the work at the bottom level is 2

So T(N) = = = =

Solving for Run-Times without Master Theorem

T(n) = T(n − 2) + 5n; T(1) = T(2) = 1

Number of nodes at level i = n - 2(i)

Solving for i, we get that the bottom level is level: 1 = n - 2(i) 🡪 i = n/2

= 5n + T(n-2)

= 5n + (5n -2) + T(n-2-2) = 5n + (5n - 2) + T(2(2))

= 5n -2(0) + 5n - 2(1) + 5n - 2(2) + T(2(3)

= 5n -2(0) + 5n - 2(1) + 5n - 2(2) + … + 5n - 2(n/2)

So you’re adding 5n n/2-times, so the runtime is Θ(n2)

T(n) = T(n/4) +

Number of nodes at level i = n/4i

Solving for i, we get that the bottom level is level: 1 = n/4i = log4n

= n1/3 + T(n/4)

=

so the total runtime is Θ(n1/3)

Fast Integer Multiplication (runtime = O(nlog 2 3)

x\*y. x = 87 (= 10101112 in binary) and y = 67 (= 10000112).

xL = (0101)2 xR = (0111)2 xL = 5 and xR = 7

yL = (0100)2 yR = (0011)2 yL = 4 and yR = 3

P1 = multiply(xL, yL) = multiply(5, 4) = multiply(0101, 0100)

P2 = multiply(xR, yR) = multiply(7, 3) = multiply(0111, 0011)

P3 = multiply(xL + xR, yL + yR) = multiply(12, 7) = multiply(1100, 0111)

P1 = 5 × 4 = 20 P2 = 7 × 3 = 21 P3 = 12 × 7 = 84.

x\*y = P1 × + (P3 − P1 − P2) × + P2

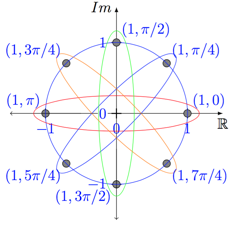
20 · 256 + 43 · 16 + 21 = 5120 + 688 + 21 = 5829 or 87 · 67 = 5829,

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Imaginary Numbers and Everything FFT

*i*0 = 1 *i*1 = *i* *i*2 = -1 *i*3 = *i*\* *i*2 = -*i* *i*4 = (*i*2)2 = 1

Plot (a+b*i*) as the point (a,b) on the complex plane. (x-axis is real, y-axis is imag.)

(a+b*i*) can be written as r(cosθ +*i*sinθ) where , 0≤θ≤2π, cosθ = a/r or sinθ = b/r  
  
or in polar coordinates: (r1,θ1)\*(r2,θ2) = (r1r2,(θ1 + θ2) mod 2π)  
(r,θ)k = (rk, kθ mod 2π)  
-1 in polar coordinates = (1,π) so –(r,θ) = (r,θ)\*(1,π) = (r,(θ+π) mod 2 π)  
Visual of the 8th roots of unity. Squaring them gives 4 roots of unity: {(1,0), (1,π/2), (1,π), (1,3π/2)}  
(1,10π/7) is a 7th root of unity b/c (1,10π/7)7 = (1,(7\*10π/7)mod 2π) = (1,0) = 1  
and a 14th root for the same reason. But it is not primitive 14th root of unity b/c there is k < 14 where (1,10π/7)k = 1  
For ω = e*i*2π/8 = cos π/4 + *i*sin π/4 = 1/√2 + (1/√2)*i*: ω0 = 1, ω1 = 1/√2 + (1/√2)*i*, ω2 = *i*,   
ω3 = -1/√2 + (1/√2)*i*, ω4 = -1, ω5 = -1/√2 - (1/√2)*i*, ω6 = -*i*, ω7 = 1/√2 - (1/√2)*i*  
1+3x+5x2+7x3+8x4+6x5+3x6+2x7 with ω = e*i*2π/8. Ae={1,5,8,3},ω2 = *i*, Ao={3,7,6,2},ω2 = *i* Ae(x)=1+5x+8x2+3x3 for x = ω0, ω2, ω4, ω6 or 1, *i,* -1*, -i*Ae(ω0)= Ae(1) = 1+5\*1+8\*12+3\*13 = 17. Ae(ω2)= Ae(*i*) = 1+5\**i*+8\* *i* 2+3\* *i* 3 = -7+2*i*.   
Ae(ω4)= Ae(-1) = 1+5\*-1+8\*-1 2+3\*-1 3 = 1. Ae(ω6)= Ae(-*i*) = 1+5\*-*i*+8\**-i* 2+3\**-i* 3 = -7-2*i*.   
Ao(ω0) = 3+7\*1+6\*12+2\*13 =18. Ao(ω2) = -3+5*i.* Ao(ω4) = 0. Ao(ω6) = -3-5*i*.  
A(ω3) = Ae(ω6) + ω3 Ao(ω6) = -7-2*i* + (-1/√2 + 1/√2*i*) (-3-5*i*)  
A(ω7) = Ae(ω14 = ω6) + ω7 Ao(ω14 = ω6) = -7-2*i* + (1/√2 - 1/√2*i*) (-3-5*i*)

Randomized Selection

1. Pick random number in set. 2. Put all small numbers on one side, equal to in the middle, larger on another. 3. Repeat as long as need. Runtime: O(n) (Divide and conquer)

Closest Pair of Points (  
1. Dividing line down the middle of all the points. 2. Recuse left and right to find smallest of the left and right. 3. Find all points within left and right from the min dist. from the divide line. 4. Sort those points by y coordinate. 5. Find smallest dist of all points in the list. 6. Return closest pair. Runtime: T(n) = 2T(n/2) + O(n log n) = O(n log2n)

DFS  
u = [], v = {} [ { } ] Tree or forward. { [ ] } Back Edge. {} [] cross edge

Runtime: O(|V| + |E|) (so is BFS) note that for BFS only show tree edges

Directed Acyclic Graph (DAG)

A directed graph is acyclic if it can be topologically sorted. Meaning run DFS and list vertices in descending order of post times. Show out all the edges. If there are no back edges its acyclic.

Strongly Connected Components

u and v are SCC if u goes to v and v goes to u

Steps: 1. Reverse the graph. 2. Find the DFS forest of the reversed graph. 3. Order vertices by highest to lowest post times. 4. Run DFS on first graph based off the of new order. O(|V| + |E|)

Connected Components

a-b has 2 nodes but 1 connected component. a-b-c d-e has 5 nodes but 2 connected components \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Dijkstra’s:

Binary heap 🡪 deleteMin, insert/decrease tree: O(log |V|)

Bellman –Ford (Negative Edges):

Steps: 1. Set everything to infinity except for where you’re starting. 2. Go to all the adjacent nodes from the start node and update their dist. 3. Update all the known nodes dist every time. 4. Do this |V| - 1 times. Runtime: O(|V| \* |E|).

DAG Shortest Path:

Steps: 1. Do Topological sort. 2. Once everything is ordered, go to all the nodes from the first node updating their dist. 3. Repeat going from node 1 until the end. (works because no back edges because it’s a DAG) Runtime: O(|V| + |E|).

Cut Property:

If you have a minimum spanning tree, and you want to finish making it to some group of nodes that are not in the tree, then the edge that is smallest that connects those two groups will be part of the tree.

Disjoint Sets Data Structure:

Has make, find and union functions, and parent and rank. Used to find if there would be a cycle in Kruskal’s. Make makes a new set with one element. Find finds the top node of the set. Union joins two sets together done by union by rank, where smaller trees get attached to larger trees, and then no ranks need to change. Rank only increases if two trees union with equal ranks.

Kruskal’s:

(Greedy). Always pick the smallest edge that doesn’t make a cycle. Uses disjoint sets data structure. Runtime: O(|E| log |V|)

Prim’s:

Essentially the same as Kruskal’s except you’re starting with a node and continuing the tree. For Kruskal’s you are making a forest eventually converge. Runtime: O(|E| log |V|)

Set Cover:

(Greedy). Not perfect but close approximation.

Edit Distance:

(Dynamic). Recurrence: Min(E(i-1, j), E(i,j-1), E(i-1,j-1)+diff(i,j)). Makes a 2D array. Runtime: O(m\*n)

Knapsack:

(Dynamic). Recurrence: Max(K(w-wj, j-1) + vj, K(w, j-1)). Makes a 2D array where we either use the weight of j, or we don’t. Runtime: O(nW). This is good as long as W is small.

All Pairs Shortest Path (Floyd-Warshall)

Could run Dijkstra’s from every point in the graph, but that’s O(|V|2\*E). Runtime: O(|V|3)

Recurrence: Min(dist(i, k ,k-1) + dist(k, j, k-1), dist(i, j, k-1). Start by filling up a table of going to only adjacent nodes and then increase it in size by once each time. Return the last table.

Linear Programming:

Steps: 1. You are given a set of inequalities. The one that you are trying to maximize or minimize is called the *Objective Function.* The other inequalities are called the *Linear Constraints*. 2. Graph the constraints. 3. This graph will make an area of possible results called the *Feasible Region*. 4. The range of possible results is essentially just the points of inflection on this shaded region, called the *Vertices of Feasible Region*

Optimization Problems:

These are problems where you are trying to maximize or minimize some value. An example would be most of the dynamic programming questions.

Decision Problems:

These are questions that are yes/no answers. So an example would be is there a solution to this system of equations. Like the SAT problem.

Satisfiability (SAT):

One of the first proven NP-Complete problems. It is asking if you have an equation of Booleans, is there some way to set the variables so that the equation evaluates to true.

NP vs. P

P means that there is an algorithm to find the solution in polynomial time. NP means that the solution is verifiable in polynomial time, but the solution itself is or is not.

NP Completeness:

These problems are all of the hardest problems in the class of NP problems. Namely that all of the NP-complete problems can be mapped to the NP problems. A fast solution to any NP-complete problem gives a fast solution to all NP problems. If P does not = NP, then the NP-complete problems are harder to solve than they are to verify. Proving P = NP will win you 1 Million dollars. Solving one NP-complete problem solves them all.

Reduction:

A reduction is transforming one algorithm into another. A reduction from problem A to problem B shows that problem B is at least as difficult as A. A problem x is in NP-complete if every other problem in NP can be reduced into x. This means that x, or that all the NP-complete problems are the hardest ones.

Integer Linear Programming:

This is linear programming, but now the answer has to be an integer. With this new constraint, suddenly the problem is NP-Complete.