Algorithms Lecture Notes

Brent Yorgey

October 19, 2019

These are my lecture notes for CSCI 280 / CSCI 382, Algorithms, at Hendrix College. Much of the basis for the course (including some of the lecture notes themselves) came from a similar course taught by Brent Heeringa at Williams College.

There are two "tracks": a "traditional" lecture-based version that I taught in S'16 and S'17, and a revamped version which is a hybrid between lecture and POGIL activities (`http://pogil.org`). I have left them both here for posterity and also because I may move material back and forth between them in the future. Sections marked with (L*) were used only in the lecture-based version of the course; sections marked with (P*) only in the POGIL version; sections marked with (L/P) are used in both, but presented via a POGIL activity rather than lecture; sections marked (L) are presented via lecture in both. Contact me if you would like to receive the latest versions of the POGIL activities I have developed.

1 (P*) Introduction to POGIL and Algorithms

2 (P*) GCD analysis

3 (P*) Proving the Euclidean Algorithm correct

Here are the two algorithms we considered last time. Throughout the following we assume that $a, b \geq 0$.

```
GCDI($a$,$b$) =
  while ($b \neq 0$) and ($a \neq 0$)
    if $a \leq b$
      then $b \leftarrow b \bmod a$
      else $a \leftarrow a \bmod b$
  if $a = 0$ then return $b$ else return $a$
```

```
GCDR($a$,$b$) =
  if $b = 0$
    then $a$
    else GCDR($b$, $a \bmod b$)
```

How can we formally prove that these algorithms correctly compute the GCD of two nonnegative integers?

In proving either of these algorithms (or any algorithm) correct we really have two things to prove:

1. The algorithm always terminates (stops in a finite amount of time) for any inputs.
2. If the algorithm does terminate, it returns the correct answer.

Sometimes the fact that the algorithm terminates is obvious; but in this case it does require some proof. It often makes things simpler to break up a proof of correctness in this way.

We'll start with the recursive version; it's typically easier to prove things about recursive algorithms than iterative ones.

Theorem 3.1. *GCDR(a, b) always terminates.*

Proof. By (strong) induction on b.

- If $b = 0$, GCDR$(a, b) = a$.
- Otherwise, if $b > 0$, GCDR makes a recursive call to GCDR$(b, a \bmod b)$. This is well-defined since $b \neq 0$. By definition $0 \leq a \bmod b < b$, so the second argument to the recursive call is strictly smaller than b. Hence, by the induction hypothesis, that recursive call must terminate.

  Put more informally, the second argument gets strictly smaller with every recursive call. It cannot keep getting smaller forever; it must eventually reach the base case of 0 and stop. SDG

Now, in order to prove that GCDR is correct, we first need a small lemma about the mathematical gcd function. Note carefully the difference between GCDR, GCDI, and gcd: the former two refer to the particular algorithms defined by the code above, whereas gcd refers to the abstract mathematical function defined in terms of divisors and so on. gcd is a *specification* whereas GCDR and GCDI are *implementations*; our job is to prove that they actually produce identical results, that is, that the implementations match the specification.

Lemma 3.2. *If $b \neq 0$ then* $\gcd(a, b) = \gcd(a \bmod b, b)$.

Proof. The proof uses some basic number theory. I typically skip it since we're more interested in the algorithmic aspects of the proof. However, the proof is included here in the notes for completeness.

It suffices to prove that for any $d > 0$, d evenly divides both a and b if and only if it evenly divides $(a \bmod b)$ and b. Then (a, b) and $(a \bmod b, b)$ have the same set of common divisors, and hence the same greatest common divisor.

Note that $a \bmod b = a - kb$ for some constant k. If d divides a and b then it also divides kb and therefore $a - kb$. Conversely, suppose some e divides both b and $a-kb$. Then e divides kb, and hence it divides the sum $(a-kb)+kb = a$. $\boxed{\text{SDG}}$

Theorem 3.3. $\mathsf{GCDR}(a, b) = \gcd(a, b)$.

Proof. By induction on b.

- In the base case, when $b = 0$, $\mathsf{GCDR}(a, 0) = a$, which is indeed the definition of $\gcd(a, 0)$.
- Now suppose $b > 0$ and that for all a, $\mathsf{GCDR}(a, b') = \gcd(a, b')$ whenever $b' < b$. In this case $\mathsf{GCDR}(a, b) = \mathsf{GCDR}(b, a \bmod b)$. We know $a \bmod b < b$, so the induction hypothesis applies (note in particular that the IH holds for *any* a, and in particular we can pick b as our "a"). We concude that $\mathsf{GCDR}(a, b) = \mathsf{GCDR}(b, a \bmod b) = \gcd(b, a \bmod b) = \gcd(a \bmod b, b) = \gcd(a, b)$, where the last two steps follow from the fact that gcd is symmetric (easily deduced from its definition) and the previous lemma. $\boxed{\text{SDG}}$

Now we'll prove the iterative algorithm correct, beginning with termination.

Theorem 3.4. $\mathsf{GCDI}(a, b)$ *always terminates.*

Proof. Each time through the loop, either a gets smaller and b stays the same, or b gets smaller and a stays the same: if the loop starts out with $a \leq b$, then b gets smaller, since $b \bmod a < a \leq b$. Otherwise a gets smaller, since $a \bmod b < b < a$. Since neither ever gets bigger and one gets strictly smaller every loop iteration, one of them must eventually hit 0, which will stop the loop.[1] $\boxed{\text{SDG}}$

Theorem 3.5. $\mathsf{GCDI}(a, b) = \gcd(a, b)$.

Proof. Suppose we initially call $\mathsf{GCDI}(m, n)$. Then we claim the following *loop invariant*: each time right before doing the loop check again, $\gcd(a, b) = \gcd(m, n)$. That is, the gcd of the *current* values of a and b is always equal to the gcd of the *original* arguments to GCDI.

We prove this by induction on the number of loop executions.

- In the base case, before the loop has executed at all, a and b are equal to the original arguments to GCDI, hence the invariant holds trivially.

[1] Technically this is a proof by (well-founded) induction on pairs $\mathbb{N}^2$ under the well-founded relation $(a, b) < (x, y)$ if $a < x$ and $b = y$ or $a = x$ and $b < y$.

- Otherwise, suppose the invariant held after n loop iterations; we must show it will still hold after one more loop iteration. But this follows from Lemma 3.2.

Finally, after the loop ends, we will have either $a = 0$ or $b = 0$, and we can see that we will return wichever one is nonzero. This correctly returns the gcd of a and b since $\gcd(a, 0) = a$. But because of the loop invariant we know that $\gcd(a, b) = \gcd(m, n)$, hence $\mathsf{GCDI}(m, n) = \gcd(m, n)$. $\boxed{\text{SDG}}$

4 (L*) Stable matchings and Gale-Shapley

Administrivia

- Show course webpage.
- Weekly HW, due Fri 4pm. (See short HW due this Friday!!) Solutions handed out Wednesday.
- Collaboration but separate write-up. I take academic integrity very seriously.
- 2 midterms and a final—given problems ahead of time, come in and write solutions.
- Explain office hours, youcanbook.me, email habits.

Course goals

What is this course all about? Two main things:

1. *Solving* problems: learning techniques, skills, methods, common problems and solutions, etc.
2. *Proving* that our solutions are good (correct, fast, low memory, etc.)

Connecting theory and practice!

Stable matching problem

Input:

- n doctors $A = \{a_1, \dots, a_n\}$
- n hospitals with a slot for a resident $B = \{b_1, \dots, b_n\}$
- Rank lists: each doctor has a list of all hospitals in preferred order, and each hospital has a list of doctors.

Example:

- Doctors x, y, z
- Hospitals MGH, Mayo, ACH
- Rank lists:

x	MGH	Mayo	ACH
y	ACH	MGH	Mayo
z	Mayo	MGH	ACH

MGH	z	x	y
Mayo	y	x	z
ACH	y	z	x

Goal: match up doctors and hospitals so no one wants to swap. That is, for any given doctor d and hospital h, either:

- d and h are matched, or
- d prefers their current hospital over h, or
- h prefers their current doctor over d.

So d and h don't want to both abandon their current match and switch to each other. This kind of matching where no one wants to swap is called *stable*.

- Example of stable matching: x—MGH, y—ACH, z—Mayo.
- Example of unstable matching: x—Mayo, y—MGH, z—ACH. Note that x prefers MGH over Mayo, and MGH prefers x over y.

How long would a *brute-force* solution take?

- List every possible matching ($n!$)
- Check each matching to see if it is stable
    - Check every pair of doctor & hospital (n^2)

So something like $O(n^2 n!)$, yikes.

The Gale-Shapley algorithm

Historically this has been called the *stable marriage* problem, phrased in terms of men & women pairing off. Studied by Gale and Shapley (1962), who gave the following algorithm. This algorithm (a variant of it) is actually used by the National Resident Matching Program to match residents and hospitals.

Algorithm 1 Propose-Reject - finds a stable matching

```
Initialize each proposer $p$ and accepter $a$ as Free
while there is a free proposer who hasn't proposed to every accepter do
    Choose a free proposer $p$
    $a \leftarrow$ first accepter on $p$'s list to whom $p$ has not yet proposed
    if $a$ is Free then
        $p$ and $a$ are Matched (for now)
    else if $a$ prefers $p$ to their current match $p'$ then
        $p$ and $a$ are Matched and $p'$ is Free
    else
        $a$ rejects $p$ and $p$ remains Free
```

Pick two volunteers to be algorithm masters—job is to make sure algorithm is being correctly followed. Then split remaining students into 2 equal groups (add myself if an odd number), doctors and hospitals.

- Hospitals will have a letter. Have each doctor make up a ranking of hospital letters.
- Have each hospital make up a ranking of numbers.
- Now randomly assign letters and numbers. Write assignments up on the board so everyone can write down names next to their ranking.
- On your piece of paper:
  - Your ranking
  - Your identity
  - Who you are currently matched with
  - Doctors: remember to cross off hospitals you have already proposed to

5 (L*) Proof of Gale-Shapley correctness

A lot of what we will do in this course revolves around creating formal mathematical models of problems and giving careful, formal mathematical proofs. Today we will describe a formal model of the stable matching problem and give a formal proof of the correctness of the Gale-Shapley algorithm.

We have a set of *proposers* $P = \{p_1, \ldots, p_n\}$ and a set of *accepters* $A = \{a_1, \ldots, a_n\}$. Each proposer p_i has a *ranking* of accepters, which is a list of all the A in some particular order. We say p *prefers* a_i over a_j when a_i occurs earlier in p's list than a_j. Similarly, each a_i has a ranking of proposers.

Definition 5.1. A *matching* is a subset $M \subseteq P \times A$ (a *relation*) such that each $p \in P$ appears in at most one element of M, and similarly for each $a \in A$.

Definition 5.2. A *perfect matching* is a matching M in which each element of P occurs exactly once, and similarly for each element of A.

Definition 5.3. A *stable matching* is a perfect matching M such that for each $a \in A$ and $p \in P$, at least one of the following holds:

- $(a, p) \in M$
- $(a, p') \in M$ and a prefers p' over p
- $(a', p) \in M$ and p prefers a' over a.

Let's prove that the Gale-Shapley algorithm always produces a stable matching. We actually have to prove several things: first, that the algorithm terminates, and second, that when it terminates it will result in a matching that is perfect and stable. NOTE: there are a lot of details missing from the pseudocode! Pseudocode allows us to talk about the *correctness* of an algorithm but not its efficiency. We'll get to that next class.

We start by making a few simple observations.

Observation 1. Once an accepter becomes Matched, they never become Free again.

Proof: lines 6, 8, 10 never make accepters Free.

Observation 2. An accepter ends up MATCHED to their most preferred proposer who proposed to them.

Proof: accepters only ever trade up.

Observation 3. No one is ever MATCHED to more than one other at a time.

Now we prove that the algorithm terminates. In fact:

Claim 5.4. *The Gale-Shapley algorithm terminates after at most n^2 iterations.*

Proof. Generally, we need a measure of *progress* that changes monotonically every iteration (*i.e.* always goes up or always goes down), along with a limit that it can't go above/below. In this case the number of, say, free proposers doesn't work, since it might not go up. Number of free accepters doesn't work either. Instead, consider the number of pairs (p, a) where p has proposed to a: in each round, some p proposes to some a they have never proposed to, so this set always increases by 1 each round. The total number of such pairs is n^2, so that is the maximum number of iterations of the loop. SDG

Claim 5.5. *The Gale-Shapley algorithm returns a perfect matching.*

Proof. By contradiction. By Observation 3, it definitely returns a *matching*, so suppose the matching is not perfect. Then there must be some $a \in A$ and $p \in P$ which are both FREE. We must derive a contradiction.

By Observation 1, a must have been FREE for the entire time. The only way the algorithm could terminate with p FREE is if p proposed to all A; hence p must have proposed to a at some point. But a would have been FREE then and hence would have been matched; this is a contradiction. SDG

Claim 5.6. *The perfect matching returned is stable.*

Proof. By contradiction. Suppose it returns a matching which is perfect but unstable. Then there exist some matched pairs (p_1, a_1) and (p_2, a_2) where p_1 and a_2 prefer each other over their current matches. Since proposers always propose in order of preference, p_1 must have proposed to a_2 at some point *before* proposing to a_1. But by Observation 2, accepters always end up with the most preferred out of those who proposed to them. This is a contradiction since we assumed a_2 ended up with p_2 even though p_1, whom they prefer, also proposed to them. SDG

6 (L*) Data representation for Gale-Shapley

[Should have plenty of time, get them to figure out some of this on their own. Project G-S algorithm on the screen again.]

Note that pseudocode lets us reason about *correctness*, but not about *time complexity*! Does the G-S algorithm run faster than brute force ($O(n!n^2)$)? We proved the number of loop iterations has n^2 as an upper bound. So, is G-S $O(n^2)$? Not necessarily! It depends on how long each loop iteration takes. We have to talk concretely about the actual data structures used to implement the algorithm.

First, how to represent the input? Give P, A ID numbers from 1 through n. (*Practical note*: in practice we can actually use dictionaries/maps instead of arrays, and directly index by names or something like that.) Use an $n \times n$ matrix $A[i, j]$ to represent A preferences: $A[i, j] = p$ means p is a_i's jth choice. Similarly use an $n \times n$ matrix P. (*Draw example preference matrices on the board.*)

Now, let's go through each operation and figure out how to make it as fast as we can.

- Identify the next FREE p? (Have them discuss in small groups.) We *don't* want to iterate over all P and find one that is free—that would be $O(n)$. (Ask them for input.) Instead, we can use a linked list/queue/stack of free proposers—any container with $O(1)$ add and remove will do. (What's the difference? Different choices of data structure will result in different orders of proposers getting to propose, which you might think could result in different matchings. Actually, it turns out the algorithm will always return the same matching no matter what order for proposers is chosen! Extra credit challenge: prove this.) Pull off the next one in $O(1)$. If they remain FREE (or if another proposer becomes FREE) add them in $O(1)$.

- For a given p, how do we get next preferred a not yet proposed to? (Again have them discuss in small groups.) We *don't* want to iterate down their preference list and check whether each one has already been proposed to ($O(n)$ again). Keep a length-n array *next*. $\mathit{next}[i] = j$ means p_i should next propose to their jth choice $P[i, j]$. Update $\mathit{next}[i]$ by incrementing after each proposal.

- How do we keep track of who is currently matched? Array $\mathit{matched}[i] = j$ means a_i is matched to p_j. $\mathit{matched}[i] = 0$ means a_i is FREE.

- How do we check the preferences of an a that is proposed to? We *don't* want to scan through a's whole preference list looking for their current match and new proposer. Are we stuck? Actually we can do something clever here: keep an *inverted index* of the matrix A which is another $n \times n$ matrix *rank*, defined so that $\mathit{rank}[i, j]$ is the rank of p_j in the preference list of a_i. For example, if a_2's top choice is p_3 then $\mathit{rank}[2, 3] = 1$. Across each row, we have switched values and indices. Now, to see whether a_i

prefers their current match p_j (which we can find by looking in $matched[i]$) or the new proposer p_k, we can just compare the values of $rank[i, j]$ and $rank[i, k]$ in $O(1)$ time.

But how do we compute *rank* in the first place? We can build it up in $O(n^2)$ time by iterating over A. But we only have to do this once, at the very beginning.

Therefore we spend $O(n^2)$ time precomputing *rank*, and then execute a loop at most $O(n^2)$ times doing a constant amount of work each iteration, so the whole algorithm runs in $O(n^2) + O(n^2) = O(n^2)$ time.

7 (L/P) Asymptotic analysis I

Motivation

Why should we examine problems analytically?

1. The analysis is independent of the algorithm implementation, the language in which the program is implemented, and the architecture in which the program is run. We insulate ourselves to all these variables.
2. Theoretically efficient almost always implies practical efficiency.

Why perform worst-case analysis?

1. Worst-case analysis captures efficiency reasonably well in practice. There are exceptions (like Quicksort and the Simplex method for linear programming).
2. Worst-case is a *real* guarantee.
3. Average-case analysis is hard to nail down: You often don't known anything about the distribution of inputs (although randomized algorithms can help).

What does efficient mean?

1. Theoretically, we take efficient to mean *runs in time polynomial in the size of the input* but practical efficiency is usually bounded above somewhere between $O(n \log n)$ to $O(n^3)$ depending on the application.

Why should we use asymptotic analysis?

1. Precise bounds are difficult.
2. Precise bounds on runtime are meaningless since they always depend on the choice of language, architecture, library, etc.
3. Equivalency up to a constant factor is often the *right* level of detail when making algorithmic comparisons.

Definitions

From now on, $n \geq 0$ and $T(n) \geq 0$.

Definition 7.1 (Big-O)**.** *$T(n)$ is $O(g(n))$ iff $\exists c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, $T(n) \leq c \cdot g(n)$.*

Draw a couple pictures. One with $T(n)$ being $\Theta(g(n))$, one with it being much less. Draw $g(n)$, then a multiple of $g(n)$, then have $T(n)$ bounce around a bit before falling in below the multiple; draw in n_0.

Example.

$$\begin{aligned} T(n) &= 3n^2 + 17n + 8 \\ &\leq 3n^2 + 17n^2 + 8n^2 \quad (n \geq 1) \\ &= 28n^2 \end{aligned}$$

So choose $c = 28, n_0 = 1$, then $T(n)$ is $O(n^2)$. (In fact we could also pick $c = 4$ along with a bigger n_0.)

Note $T(n)$ is also $O(n^3)$, and so on. Big-O is like *less than or equal to.*

Definition 7.2 (Big-Omega)**.** *$T(n)$ is $\Omega(g(n))$ iff $\exists c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, $T(n) \geq c \cdot g(n)$. (greater-than-or-equal)*

Definition 7.3 (Big-Theta)**.** *$T(n)$ is $\Theta(g(n))$ iff $T(n)$ is $O(g(n))$ and $\Omega(g(n))$. (equal)*

A lot of the time people use big-O when what they really mean is Θ. In this class we will be very careful to use them properly. It's useful to have all of them at our disposal. For some problems we know exactly how fast they can be solved, so we use Θ. For some problems we know some O and some Ω but they are not the same. For example, matrix multiplication: we know it must take $\Omega(n^2)$ time, and there are algorithms that show it is $O(n^{2.37\cdots})$, but no one knows what the theoretical limit is (we will look at matrix multiplication later in the semester).

Now for three proofs that $1 + 2 + \cdots + n$ is $\Theta(n^2)$.

Proof. $1 + 2 + \cdots + n < n + n + \cdots + n = n^2$, so it is $O(n^2)$ $(c = n_0 = 1)$.

Also, $1 + 2 + \cdots + n > n/2 + (n/2 + 1) + \cdots + n > n/2 + n/2 + \cdots + n/2 = (n/2)^2 = n^2/4$, so it is $\Omega(n^2)$ $(c = 1/4,\ n_0 = 1)$. SDG

Proof. By geometry. Draw a triangle, it's inside a square, also there is a square 1/4 the size inside it. SDG

It is usually really annoying to use these definitions directly. Instead we can often use this theorem:

Theorem 7.4. *If $0 \leq \lim_{n\to\infty} T(n)/g(n) < \infty$ then $T(n)$ is $O(g(n))$.*

Proof. If $\lim_{n\to\infty} T(n)/g(n) = k \geq 0$ then there must exist some $c > k$ and n_0 (draw a picture!) such that $T(n)/g(n) \leq c$ for all $n \geq n_0$, *i.e.* $T(n) \leq cg(n)$. SDG

Remark. Note that the converse is not true since $T(n)$ might be $O(g(n))$ even if the limit does not exist. For example, let $T(n) = 0$ when n is even, 1 when n is odd. Then $T(n)$ is $O(1)$ but $\lim T(n)/1$ does not exist. But typically this won't be an issue with the functions we will see.

We also have:

Theorem 7.5. *If* $0 < \lim_{n\to\infty} T(n)/g(n) \leq \infty$ *then* $T(n)$ *is* $\Omega(g(n))$.

Proof. Exercise. SDG

Corollary 7.6. *If* $0 < \lim_{n\to\infty} T(n)/g(n) < \infty$ *then* $T(n)$ *is* $\Theta(g(n))$.

And now for the third proof that $1 + 2 + \cdots + n$ is $\Theta(n^2)$:

Proof. $1+2+\cdots+n = n(n+1)/2 = n^2/2+n/2$, and $\lim_{n\to\infty}(n^2/2+n/2)/n^2 = 1/2$. SDG

Arithmetic with big-O (same for Omega and Theta):

- $k\Theta(f) = \Theta(f)$ when k is a constant
- $\Theta(f)\Theta(g) = \Theta(fg)$ (e.g. *nested loops*)
- $\Theta(f) + \Theta(g) = \Theta(\max(f, g))$ (e.g. *adjacent loops*)

For example, $O(3n^2+17n+8) = O(3n^2)+O(17n)+O(8) = O(n^2)+O(n)+O(1) = O(n^2)$.

8 (L/P) Asymptotic analysis II

Definition 8.1 (Little-o)**.** $T(n)$ is $o(g(n))$ if $\lim_{n\to\infty} T(n)/g(n) = 0$. Stronger than big-O: T is really *less than* g. If the limit is a positive constant then they grow at the same rate; if the limit is zero then g outstrips T.

Present "complexity zoo", with examples of things having each common order, and some relevant facts interspersed. (Look in the textbook for more examples.) Our zoo will be ordered from smallest to biggest; each thing will be little-o of the next thing.

Constant time: $\Theta(1)$

Does not depend on size of the input. Example: "Return the first element of a list."

Logarithmic time: $\Theta(\lg n)$

Examples: binary search, height of tree with n nodes; in general, *repeatedly halving.*

Note: we write lg for $\log_2$. Turns out $\Theta(\log n)$ vs $\Theta(\lg n)$ vs $\Theta(\ln n)$ etc doesn't matter: $\log_a n = \log_b n / \log_b a$, just a constant factor.

Theorem 8.2. *$\log n$ is $o(n^x)$ for all $x > 0$.*

Proof. Since the base of the log doesn't matter, it suffices to prove this for $\ln n$. Consider $\lim_{n\to\infty} \ln n/n^x$. Using l'Hôpital's rule, this is

$$\lim_{n\to\infty} (1/n)/(xn^{x-1}) = \lim_{n\to\infty} 1/(xn^x) = 0.$$

$\boxed{\text{SDG}}$

This is somewhat surprising! $\Theta(\log n)$ is actually smaller than n to *any* positive power: even $\Theta(\sqrt[15]{n})$ or whatever. $\Theta(\log n)$ is not "halfway between" $\Theta(1)$ and $\Theta(n)$; in some sense it is much closer to $\Theta(1)$.

Linear time: $\Theta(n)$

Examples: linear search; maximum/minimum/sum of a list; merge sorted lists. In general, do something to every item of input.

$\Theta(n \lg n)$

Examples: mergesort or quicksort; in general, divide & conquer with $\Theta(n)$ work to merge. We'll study this in more detail later in the semester.

[Note: don't prove that mergesort is $\Theta(n \lg n)$, we'll do that when we get to divide & conquer.]

[Insert here proof of lower bounds for comparison-based sorting?? IF TIME, come back and stick it in. Probably won't be enough time.]

Quadratic time: $\Theta(n^2)$

Examples: nested loops (often), sum $1 \dots n$. (But have to be careful with loops where the number of iterations is not simply n!) All pairs.

Polynomial time: $\Theta(n^k)$

k nested loops. Number of subsets of size k, i.e. $\binom{n}{k}$ is $\Theta(n^k)$. Each n^j is $o(n^k)$ for $j < k$: $\lim_{n\to\infty} n^j/n^k = 0$.

Exponential time: $\Theta(2^n)$

Examples: all subsets. All bitstrings of length n. Number of nodes (also number of leaves) in a tree of height n. $1+2+4+8+\cdots+2^n = 2^{n+1}-1$, so it is $\Theta(2^n)$. In general: doubling every time.

Theorem 8.3. *n^k is $o(r^n)$ for all integers $k \geq 0$ and real numbers $r > 1$.*

Proof. $\lim_{n\to\infty} n^k/r^n = \lim_{n\to\infty} kn^{k-1}/r^n \ln r = \lim_{n\to\infty} k!/r^n(\ln r)^k = 0$. $\boxed{\text{SDG}}$

There is an insurmountable gulf between polynomial and exponential time. For example, even n^{295} is $o(1.001^n)$! We usually take this to be the dividing line between "efficient/feasible" and "inefficient/infeasible". (Of course n^{295} is probably not actually feasible but in practice we don't see that.)

Note j^n is $o(k^n)$ for $j < k$: $\lim_{n\to\infty} j^n/k^n = \lim_{n\to\infty}(j/k)^n = 0$.

Factorial time: $O(n!)$

$O(n!)$: all orderings of input. This is really, really bad. k^n is $o(n!)$ for all k.

9 (L*) Largest Sum Subinterval (LSS) problem

Input: array $A[n]$ of integers (1-indexed). Output: Largest sum of any subinterval. Empty interval sum = 0.

Examples:

- $[10, 20, -50, 40]$
- $[-2, 3, -2, 4, -1, 8, -20]$

(Note the problem is boring when all the entries are positive!)

How to solve this?

Brute-force

- Look at all subintervals: there are $\binom{n+1}{2}$ (choose two endpoints, but they can be the same) which is $\Theta(n^2)$.
- Sum each subinterval: $O(n)$, though it is often smaller.

3 nested loops, so $O(n^3)$.

How to prove it is also $\Omega(n^3)$? Note this is nontrivial! The total number of items we look at is $n \times 1 + (n-1) \times 2 + (n-2) \times 3 + \cdots + 1 \times n$. Several proof methods:

1. **Algebraic**. Throw some algebra at it. Can show this is greater than a multiple of n^3, but it takes some work.
2. **Geometric**. 1 long row, then 2 slightly shorter rows, then 3 even shorter rows, etc. yields a tetrahedron which obviously has a *volume* which is proportional to n^3.
3. **Combinatorial**. Number of subintervals is about $\binom{n}{2}$—actually it is exactly $\binom{n+1}{2}$. And the number of elements in subintervals that we examine is in fact $\binom{n+2}{3}$, which we know is $\Theta(n^3)$.

   (Combinatorial proof: add two new slots to either side of the array; consider all possible choices of 3 slots including the two new ones. Take the two outermost choices to identify the subinterval strictly contained by them; the inner choice is the state of the inner loop counter. So the total number of loops is exactly $\binom{n+2}{3}$ which is $\Theta(n^3)$.)

Avoid repeated work/precomputation

- Make a table of partial sums $S[n]$, where $S[i]$ = sum of $A[1] \ldots A[i]$. This takes time $\Theta(n)$. We can now compute the sum of any interval in $\Theta(1)$: $A[i]+\cdots+A[j] = S[j]-S[i-1]$. So now the total is $\Theta(n)+\Theta(n^2) = \Theta(n^2)$.

- Or use some sort of “sliding window” technique to go from the sum of each subinterval to the next in $\Theta(1)$ time.

Be clever

Actually we can do even better! Notation: let $[j, k]$ denote $\sum_{i=j}^{k} A[i]$.

Some observations:

1. If $[1, j] \geq 0$ for all j, then LSS = whichever $[1, k]$ is biggest.

   Proof. $[i, j] \leq [1, j]$ (since $[1, (i-1)] \geq 0$) and $[1, j] \leq [1, k]$ by definition. $\boxed{\text{SDG}}$

2. Let j be the smallest index such that $[1, j] < 0$. Then the LSS does not include index j. (Intuitively, when $[1, j]$ first falls below 0, the problem “resets”. Can consider the rest of the array as a new smaller array.)

   Proof. Suppose j is the smallest such that $[1, j] < 0$, and let $u \leq j \leq v$. Note $[u, j] < 0$ since $[u, j] = [1, j] - [1, u-1]$. ($[1, j]$ is assumed < 0 and $[1, u-1]$ is positive by assumption.) Hence $[u, v] = [u, j] + [j+1, v] < [j+1, v]$. $\boxed{\text{SDG}}$

This means that we can scan looking for the biggest sum so far from the beginning of the array to the current point, but as soon as the sum becomes negative we “reset the start of the array” and keep looking.

Algorithm 2 LargestSum(A)

```
$sum, largest \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    $sum \leftarrow \max(sum + A[i], 0)$
    $largest \leftarrow \max(sum, largest)$
return $largest$
```

Running time is clearly $\Theta(n)$. Formal proof of correctness uses observations from before: *sum* always contains the running sum from the most recent place where the previous running sum went negative.

10 (L/P) Graphs (KT 3.1)

Use slides (`slides.04.graph-definitions.odp`) to present basic definitions (undirected, m, n, matrix vs adj list representations, paths, connectivity, cycles, trees).

Theorem 10.1. *Let $G = (V, E)$ be a graph with $|V| = n \geq 1$. Any two of the following imply the third:*

1. *G is connected.*
2. *G is acyclic.*
3. *G has $n - 1$ edges.*

Do these proofs carefully—as a model for them!

Lemma 10.2. $(1, 2) \implies (3)$. *If G is acyclic and has $n - 1$ edges, then G is connected.*

Proof. Suppose we have a graph G which is connected and acyclic, with n vertices. We prove that it has exactly $n - 1$ edges by induction on n. Proof idea: induction; show we can always find a leaf to delete.

- When $n = 1$, there are indeed $n - 1 = 1 - 1 = 0$ edges.
- In the inductive case, we assume the lemma holds for all connected, acyclic graphs with $n - 1$ vertices.

  Pick a vertex $v_1 \in V$ and take a walk $v_1, v_2, \ldots$, never repeating an edge. Claim: we must eventually reach a leaf (a vertex with degree 1) in at most $n - 1$ steps. All the vertices must be distinct since we can never backtrack along an edge and G has no cycles. So we can take at most $n - 1$ steps before we have visited all the vertices. Also, we must get stuck at a leaf v_j: if we get stuck at a vertex with more than one edge that we've visited, it would create a cycle; it's impossible to get stuck at a vertex with degree 0 (we could have picked v_1 to be such) because G is connected.

  Now consider $G - v_j$, that is, $(V - \{v_j\}, E - \{v_j, v_{j-1}\})$. It has $n - 1$ nodes and no cycles (removing an edge can't create any cycles), and it is also connected (since we removed a leaf). So by assumption it has $n - 2$ edges; hence G has $n - 1$.

SDG

Lemma 10.3. $(2, 3) \implies (1)$. *If G is acyclic and has $n - 1$ edges, then G is connected.*

Proof. Proof idea: counting argument.

Suppose G is acyclic and has $n-1$ edges. Let l be the number of connected components of G; we wish to show that $l=1$.

Number the components of G arbitrarily, and let component j of G have k_j vertices. Then $\sum_{j=1}^{l} k_j = n$ (the sum of all the components gives the total number of vertices). Each connected component is connected and acyclic, so by the previous lemma it has $k_j - 1$ edges. Since every edge of G lies in some component, we can add these up to get the total number of edges in G:

$$|E| = \sum_{j=1}^{l}(k_j - 1) = \left(\sum_{j=1}^{l} k_j\right) - l = n - l$$

But we assumed $|E| = n-1$, so in fact $l=1$. $\boxed{\text{SDG}}$

Lemma 10.4 (Handshake lemma)**.** *In an undirected graph $G=(V,E)$,*

$$\sum_{v \in V} \deg(v) = 2|E|.$$

Proof. Each edge gets counted twice, once in the degree for each of its endpoints. $\boxed{\text{SDG}}$

Lemma 10.5. $(1,3) \implies (2)$*: If G is connected with $n-1$ edges, then it is acyclic.*

This one will be on the HW.

11 (L) BFS (KT 3.2, 3.3)

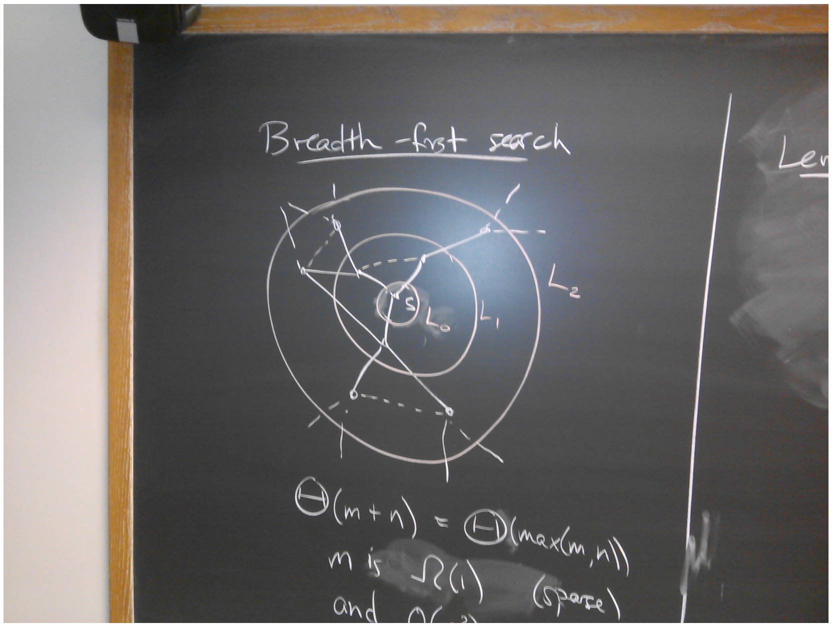
As a quick review, recall how we represent graphs as a data structure in a program. Typically we want to use a so-called *adjacency list* structure, where we store a dictionary mapping each vertex to information about its adjacent edges. It is called an "adjacency list" since traditionally each vertex might be associated with a *list* of adjacent edges. However, it makes much more sense to use a *set* rather than a list: `UGraph = Map<Vertex, Set<Vertex>>`. For a *weighted* graph we can use `WGraph = Map<Vertex, Map<Vertex, Weight>>`. If we assume $\Theta(1)$ lookup for maps or sets (*e.g.* using a hash table implementation), this lets us look up a particular edge in $\Theta(1)$ and iterate over all neighbors of a given vertex in $\Theta(\deg(v))$.

Today we'll start exploring variants on the *s-t connectivity* problem, a fundamental question we can ask about a graph:

1. Given vertices s, t in an undirected graph G, is there a path from s to t?
2. What is the length of the *shortest* path from s to t?

These problems come up in lots of applications, *e.g.* the number of introductions needed to connect to someone in a social network; path planning *e.g.* for a robot in a factory; *etc.*

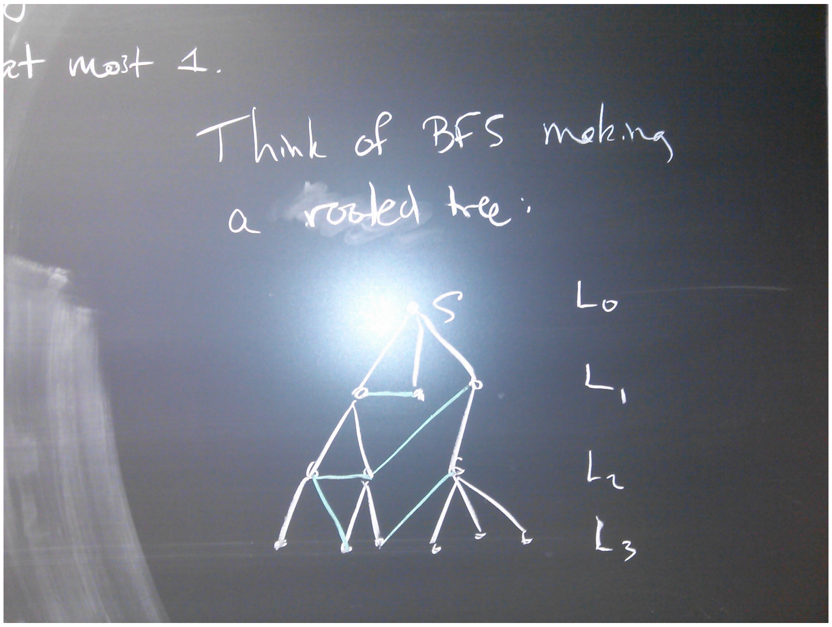
This problem is solved by the *breadth-first search* algorithm. Idea: start at vertex s and explore outward in all directions, adding vertices one "layer" at a time. That is, $L_0 = \{s\}$; L_1 = all neighbors of L_0; $\ldots L_i$ = all neighbors of L_{i-1} not in any previous layer.



Properties of BFS:

1. The shortest path from s to v has length i if and only if $v \in L_i$. (The proof is by induction on i: in the base case $i = 0$ and $L_0 = \{s\}$; then, if we assume it is true for L_k, we can show it is also true for L_{k+1}.)
2. There exists a path from s to t if and only if t is in some layer of the BFS from s.
3. For each $(u, v) \in E$, the layer of u and v differ by at most 1. (As soon as one shows up the other will show up in the next layer. Note they could be in the same layer.)

Thinka of the BFS as making a tree:



We can store the tree simply using a dictionary π (which stands for πarent) mapping each vertex to its parent in the BFS tree. This is good enough since we never need to traverse back *down* the tree. To find a shortest path from s to some ending vertex we just look up the ending vertex in the tree to get its parent, then look up its grandparent, and so on, until we reach s; the final path is just the reverse of the vertices we visited on our way up the tree.

BFS, formally:

Algorithm 3 BFS(G,s)

```
Require: Undirected graph $G = (V, E)$, vertex $s \in V$.
 1: $Q \leftarrow$ empty queue
 2: $\pi \leftarrow$ empty dictionary (parent map)
 3: $level \leftarrow$ empty dictionary (level map)
 4: Mark all vertices Unvisited
 5: Mark $s$ Visited, add $s$ to $Q$, $level[s] \leftarrow 0$
 6: while $Q$ is not empty do
 7:     remove $u$ from $Q$
 8:     for each edge $(u, v)$ adjacent to $u$ do
 9:         if $v$ is Unvisited then
10:             Mark $v$ Visited
11:             $\pi[v] \leftarrow u$
12:             $level[v] \leftarrow level[u] + 1$
13:             Add $v$ to $Q$
    return $\pi$, $level$
```

Notes:

- We could also make a version that takes a target vertex and stops early, returning just a path from s to t.
- Keeping track of the *level* array is optional.
- Note we can actually tell whether a node has been Visitedby whether or not it exists as a key in the *level* map.

Running time? Two nested loops—$\Theta(V^2)$? Not necessarily!

Theorem 11.1. *This implementation of BFS runs in $\Theta(V + E)$ time (if the graph uses an adjacency list representation).*

Proof. Line 4 takes $\Theta(V)$. Use an array/dictionary of booleans. Or if we use e.g. a hash table-based set, we can initialize an empty set of visited vertices in $\Theta(1)$ time.

Note on line 8, assuming G is using an adjacency list, it takes only $\Theta(\deg(u))$ to iterate over edges adjacent to u. (If adjacency matrix this is not as efficient.)

Loop on line 8 executes a TOTAL of $2|E|$ times, twice for each edge, because each vertex ends up being visited exactly once. Note we don't really have to consider the loop on line 6, since we can directly quantify the TOTAL number of times the innermost loop executes.

Now, how about the contents of the innermost loop?

- Checking whether Visited or Unvisited takes $\Theta(1)$ given array/dictionary.
- Adding edge to π is $\Theta(1)$.
- Adding v to L_{i+1} is $\Theta(1)$, just add to end of list or something.

Hence total time is $\Theta(V) + \Theta(E) = \Theta(E + V)$, or just $\Theta(E)$ if we use a set to represent visited vertices. SDG

Of course $\Theta(E+V) = \Theta(\max(E, V))$, but we don't *a priori* know which one is bigger. $\Theta(E)$ can be as small as $\Theta(1)$ (if there are very few edges, *i.e.* the graph is *sparse*) and as big as $\Theta(V^2)$ (if there are a lot of edges, *i.e.* the graph is *dense*).

12 (L/P) Bipartite and directed graphs (KT 3.4, 3.5)

Definition 12.1. An undirected graph $G = (V, E)$ is *bipartite* if V can be partitioned into two sets L, R such that every edge has one endpoint in L and one in R. (Draw a picture.)

These show up a lot—they are an important special class of graphs. They can be used to model relationships between two sets (*e.g.* matchings). Many problems which are difficult for graphs in general become tractable for bipartite graphs.

Another way to talk about this:

Definition 12.2. A graph is *k-colorable* if each node can be assigned one of k colors such that no two vertices connected by an edge have the same color.

Note that 2-colorable is the same thing as bipartite. We will also talk about red/blue instead of L/R. (Aside: the notion of k-colorability for $k \geq 3$ turns out to be algorithmically *much* more difficult to deal with! We will return to this much later in the semester.)

Do some examples: draw some graphs and ask whether they are bipartite.

Theorem 12.3. *G is bipartite iff it has no odd-length cycles.*

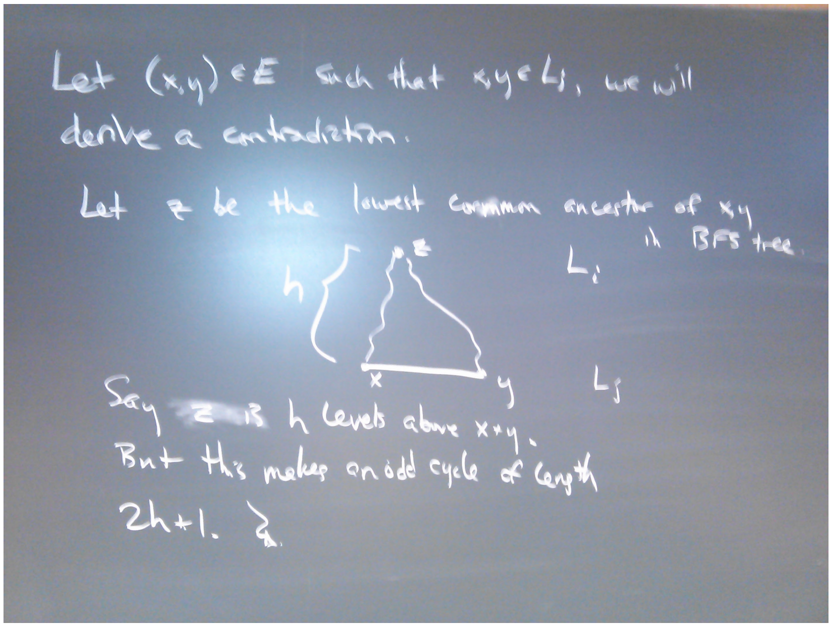
Proof. ($\Longrightarrow$) All paths must alternate between L and R. Hence every cycle is even.

($\Longleftarrow$) Pick an arbitrary vertex s and run a BFS, generating layers L_0, L_1, L_2, ... Then pick

$$L = L_0 \cup L_2 \cup L_4 \cup \dots$$
$$R = L_1 \cup L_3 \cup L_5 \cup \dots$$

Claim: every edge $(x, y) \in E$ has one endpoint in L and one in R. By the single-layer-difference property of BFS, there can't be any edges between different layers within L or R. The only possibility we have to worry about is an edge between two vertices in the *same* layer.

So, suppose $(x, y) \in E$ and $x, y \in L_j$; we will derive a contradiction. Let z be the least common ancestor of x and y in the BFS tree (draw a picture), and suppose z has height h above x and y. Then there is a cycle of length $2h + 1$, contradiction.



Let $(x,y) \in E$ such that $x,y \in L_j$, we will derive a contradiction.

Let z be the lowest common ancestor of x,y in BFS tree.

Say z is h levels above $x+y$.
But this makes an odd cycle of length $2h+1$. ↯

SDG

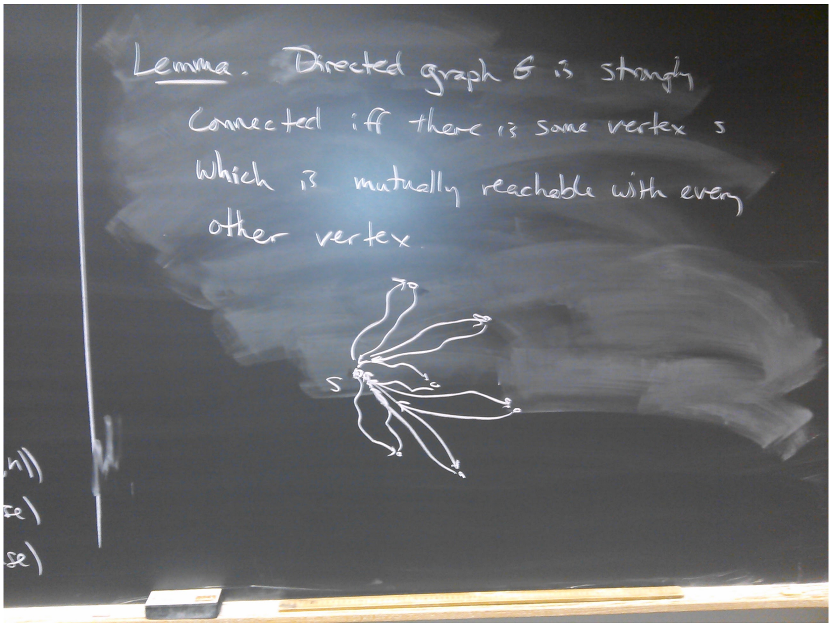
In fact, we can use this as an algorithmic *test* for bipartiteness: run a BFS from any vertex. Then G is bipartite iff there is no edge within some layer. If there is a cross-edge within a layer, we have found an odd cycle; otherwise, we have found a 2-coloring of the graph.

Another application of BFS: finding strongly connected components.

Definition 12.4. A *directed graph* is like an undirected graph except the edges are *ordered* pairs of vertices, $E \subseteq V \times V$.

Lots of things generalize naturally to directed graphs: instead of paths we have directed paths. Instead of degree we have indegree and outdegree. In a directed graph "connected" is "weakly connected". "Strongly connected" means between any two vertices there is a directed path in *both* directions. BFS extends naturally to directed graphs as well: only follow edges in the direction they are supposed to go.

Lemma 12.5. *A directed graph G is strongly connected iff there is some vertex s such that every other vertex in G is mutually reachable with s (that is, for each $v \in V$ there is a directed path from s to v and another directed path from v to s).*



<u>Lemma</u>. Directed graph G is strongly connected iff there is some vertex s which is mutually reachable with every other vertex.

Proof. ($\Longrightarrow$) If G is strongly connected we can pick any vertex as s.

($\Longleftarrow$) Let $u, v \in V$ and suppose all vertices are mutually reachable with s. Then we can construct a directed path from u to v by following the path from u to s and then from s to v; and vice versa. SDG

Definition 12.6. Given a directed graph G, its *reverse graph* G^{rev} is the graph with the same vertices and with all edges reversed.

Theorem 12.7. *A directed graph G is strongly connected if and only if all vertices are reachable from some vertex s in both G and G^{rev}.*

Proof. A vertex v is reachable from s in G^{rev} if and only if s is reachable from v in G. (A directed path from x to y in G turns into a directed path from y to x in G^{rev}.) So this is really just saying the same thing as the previous lemma. SDG

Corollary 12.8. *We can decide whether a directed graph G is strongly connected in $\Theta(V + E)$ time.*

Proof. Pick a vertex s, and run a BFS from s in G and then run another BFS from s in G^{rev}. Each BFS takes $\Theta(V + E)$ time, and computing G^{rev} takes $\Theta(E)$ time. SDG

13 (L) DAGs and topological ordering (KT 3.6)

Definition 13.1. A *directed acyclic graph* (DAG) is a directed graph with no *directed* cycles.

In general represents precedence/prerequisites. Courses; compilation; production pipeline; etc.

Definition 13.2. A *topological ordering* (*topological order*, *topological sort*, *topsort*) of a directed graph is an ordering of nodes $v_1, \dots, v_n$ such that for every $(v_i, v_j) \in E$, we have $i < j$.

In other words, we can line up the vertices so that edges only point to the right. (Draw a picture.) This corresponds to an order in which classes can be taken, tasks can be done, etc. so prerequisites are always met.

Theorem 13.3. *A directed graph G has a topological ordering iff G is a DAG.*

Proof. ($\Longrightarrow$) Suppose G has a topological ordering $v_1, \dots, v_n$. We must show that G has no directed cycles. Intuitively, this is because any directed cycle must have at least one edge pointing backwards. More formally, suppose there is a cycle C, whose lowest-numbered vertex is v_i, with the previous vertex in the cycle being v_j. But then there is an edge $v_j \to v_i$ with $i < j$, a contradiction since $v_1, \dots, v_n$ is a topological ordering. Hence G has no cycles.

($\Longleftarrow$) Proof by algorithm! SDG

Lemma 13.4. *A DAG has a vertex with indegree 0.*

Proof. Proof by algorithm. Pick any starting vertex v, and keep following incoming edges backwards until finding a vertex with no incoming edges. This process must stop: if not, by the pigeonhole principle, since there are only finitely many vertices, we must eventually visit a vertex twice, but this would form a directed cycle, and we assumed the graph is a DAG. SDG

Proof. Now we prove that if G is a DAG, it has a topological ordering. Proof by induction on n / recursive algorithm.

- Base case: if $n = 1$ there is only one vertex and no edges, so there is a trivial topological ordering.
- If $n > 1$, find a vertex v with indegree 0 by the previous lemma/algorithm. Note that $G - \{v\}$ (delete vertex and any connected edges) is also a DAG since deleting things cannot create any cycles. Then by the induction hypothesis, $G - \{v\}$ has a topological ordering. Adding v at the beginning then makes a topological ordering for G since v has no incoming edges.

SDG

This is $\Theta(V^2)$: searching for an indegree-0 vertex takes $\Theta(V)$ in the worst case, and we have to do it once for each vertex. But we can do better when the graph is sparse. The idea is to maintain some extra information that allows us to quickly find a vertex with indegree 0 on each iteration without searching.

Theorem 13.5. *A topological sorting algorithm can be implemented in* $\Theta(V + E)$ *time.*

Proof. (Assume adjacency list representation.)

We need to be able to quickly find the next remaining vertex with indegree 0 (don't want to re-run a search every time), and also quickly delete a vertex (updating the indegrees of other vertices appropriately).

Maintain:

- Array/dictionary $in[v]$ = number of incoming edges (indegree) of v. Initialize in $\Theta(V + E)$ time (just look at each vertex and count number of incoming edges).

- Queue/stack/whatever Z of vertices with indegree 0. Initialize in $\Theta(V)$ after building *in*: if $in[v] = 0$ then add v to Z.

At each iteration, dequeue a vertex v from Z in $\Theta(1)$. To delete v, decrement $in[u]$ for each edge (v, u), and add u to Z if $in[u]$ becomes zero. This is $\Theta(1)$ per edge and only looks at each edge once in total. Hence $\Theta(V + E)$ overall. □

The resulting algorithm is known as *Kahn's Algorithm*, and was first published by Arthur Kahn in 1962.

Algorithm 4 TopSort(G)

```
Require: Directed graph $G = (V, E)$.
$T \leftarrow$ empty list
$Z \leftarrow$ empty queue/stack/whatever (with $\Theta(1)$ add/remove)
$in \leftarrow$ dictionary mapping all vertices to 0
for each $v \in V$ do
    for each $u$ adjacent to $v$ do
        increment $in[v]$
for each $v \in V$ do
    if $in[v] = 0$ then
        add $v$ to $Z$
while $Z$ is not empty do
    $v \leftarrow Z.dequeue$
    append $v$ to $T$
    for each $u$ adjacent to $v$ do
        decrement $in[u]$
        if $in[u] = 0$ then
            add $u$ to $Z$
```

14 (P/L) Dijkstra's algorithm

Though we will continue studying graph algorithms, we will now specifically study several *greedy algorithms*. The basic idea of a greedy algorithm is to pick the *locally* best thing at each step. For some problems, we can prove that this leads to a *globally* best solution.

So far we have considered undirected and directed graphs, but each edge either exists or not. We will now consider *weighted* graphs, where each edge has some sort of cost.

Definition 14.1. A *weighted* (directed or undirected) graph is a graph where each edge is assigned a *weight*. We will denote the weight of edge (u, v) by w_{uv}.

The weight or length of a path is just the sum of the weights of its edges.

For now, we will consider weights in $\mathbb{R}^+$, that is, nonnegative real numbers; later, we will consider $\mathbb{R}$; in general, one can use any semiring.

Note that we can think of an unweighted graph as a weighted graph where all edges have weight 1.

Problem 1 (*s*-*t* shortest path)**.** Given vertices s, t, find the shortest (weighted, directed) path from s to t.

Note, almost all solutions actually end up solving a more general problem:

Problem 2 (single-source shortest path (SSSP))**.** Given a vertex s, find the shortest paths from s to *every other* vertex. Intuitively, you can't find shortest path to just t without exploring the rest of the graph.

In an unweighted graph, we would use BFS, but now we need to take edge weights into account. Intuition: BFS searches outwards one layer at a time, by increasing distance. We'll do the same thing: search outward by increasing distance. Imagine turning on a source of water at vertex s, and watching the water flood the whole graph. Each edge is a (directed) pipe, and the weight of an edge says how long the water takes to traverse that pipe.

Note that this only makes sense when the edge weights are *positive*—if water can "travel back in time" it invalidates this whole scheme; we can't necessarily find the next vertex to be reached by the water with a *greedy* approach. If we want to find shortest paths when edge weights can be negative, we need a different algorithm (which we will study later).

Here's the basic algorithm, which we can think of as a generalization of BFS to weighted graphs. Recall that BFS keeps track of

- a set of visited vertices,
- a list of layers,

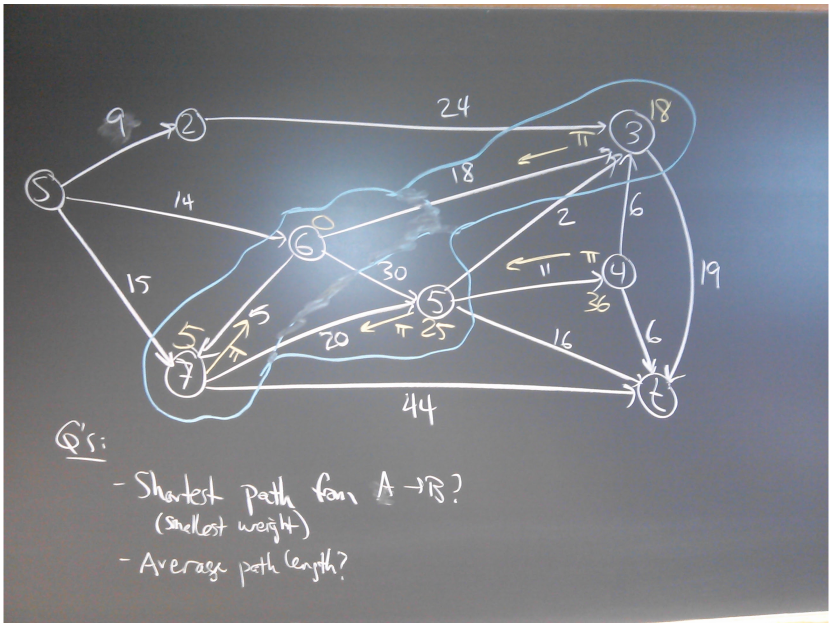
- the parent map π.

The list of layers is really telling us the shortest distance to the nodes in each layer, since the distance is just the number of edges. The appropriate generalization will thus keep track of:

- the set S of vertices that the water has reached;
- the shortest distance $d[v]$ from s to d (*i.e.* how far did the water have to go before it first reached v?),
- the predecessor $\pi[v]$ of each vertex in the shortest path from s to v (*i.e.* where did the water come from when it first reached v?) We can use π to reconstruct the shortest path from s to any vertex v (just start at v and use π to follow edges backwards).

Algorithm 5 BasicDijkstra(G,s)

```
Require: Weighted graph $G = (V, E)$, vertex $s \in V$.
$S \leftarrow \{s\}$
$d[s] \leftarrow 0$, all other $d[v] \leftarrow \infty$
while $S \neq V$ do
    Pick $u \in S$, $v \notin S$ such that $d[u] + w_{uv}$ is as small as possible.
    $\pi[v] \leftarrow u$
    $d[v] \leftarrow d[u] + w_{uv}$
    Add $v$ to $S$
return $\pi, d$
```



Q's:

- Shortest path from A → B?
(Smallest weight)
- Average path length?

How long does this take? It depends a lot on how we implement line 4. A brute-force approach simply iterates over all edges (u, v), filters out only those with $u \in S$ and $v \notin S$, and picks the one with the smallest $d[u] + w_{uv}$. Since the while loop adds one vertex to S on each iteration, it executes $|V|$ times, making the whole algorithm $\Theta(VE)$ (which could be $O(V^3)$ if the edges are dense). It turns out we can do better than this if we use a more clever scheme for quickly finding the best edge on line 4.

15 Dijkstra proof and asymptotics

Theorem 15.1. *Dijkstra's algorithm correctly solves the SSSP problem for a weighted graph with nonnegative edge weights.*

Proof. We will prove the loop invariant that for all $v \in S$, $d[v]$ is the length of the shortest path from s to v.

The proof is by induction on the number of loop iterations.

- At the start of the algorithm, $S = \{s\}$ and $d[s] = 0$.
- Now suppose the invariant holds and the loop executes one more time. Let u, v be the vertices picked by the algorithm. We will show that any other path from s to v must be at least as long as the path from s to v via u. [Draw a picture.] Any other path from s to v must exit S at some point, say the edge where it exits S is $x \to y$. But by the way u, v were chosen, we know that the shortest path from s to y via x is at least as long as the path from s to v (since by the invariant we know the shortest path from s to x), plus there may be extra distance from y to v as well. (Notice how the assumption of nonnegative edge weights is important here—if the distance from y to v could be negative it invalidates this proof!)

  Hence, the loop updates S and d appropriately and the invariant still holds.

$\boxed{\text{SDG}}$

How fast can we make Dijkstra run? While loop obviously executes n times. The crux of the issue is how long it takes to pick the best u and v. Brute-force: just consider every edge and find the minimum. Then the whole algorithm would be $\Theta(VE)$ which could be as high as $O(V^3)$. Can we do better?

The key, as usual, is to use some data structures to keep track of enough information so that we can pick u and v quickly without having to search through the whole graph every time.

- We will expand d to keep track of not just the shortest distances to vertices in S, but the *current shortest known* distances to other vertices as well. So $d[v]$ will always be an *upper bound* on the shortest distance from s to v. We may have to update $d[v]$ every time we add a vertex u to S with an edge (u, v).
- We will expand π similarly: $\pi[v]$ is the predecessor of v along the *current shortest known* path from s.

Idea: at each iteration we want to pick the vertex $v \notin S$ with the *smallest* $d[v]$. We will store the vertices $v \notin S$ in some kind of data structure so that we can quickly remove the one with the smallest $d[v]$. We then need to be able to update d and π appropriately.

So we need a data structure that supports the following operations, *i.e.* a priority queue:

	Array/dict	Sorted array	Binary Heap	Fibonacci Heap
Remove min	$\Theta(n)$	$\Theta(1)$	$\Theta(\lg n)$	$\Theta(\lg n)$
Decrease key	$\Theta(1)$	$\Theta(n)$	$\Theta(\lg n)$	$\Theta(1)$

(Annoying note: Java's standard library does not have a priority queue implementation capable of doing a fast `decreaseKey` operation. One can simply remove an element and re-insert it, but this takes $\Theta(n)$ since there is no way to find an element other than scanning over the entire heap. An implementation supporting a fast `decreaseKey` could, for example, keep a hash table on the side which keeps track of the location of each item within the heap.)

Algorithm 6 Dijkstra(G,s)

```
Require: Weighted graph $G = (V, E)$, vertex $s \in V$.
$S \leftarrow \{\}$
$d[s] \leftarrow 0$, all other $d[v] \leftarrow \infty$
Create priority queue $Q$ containing all nodes, using $d[v]$ as the key for $v$.
while $Q$ is not empty do
    $u \leftarrow Q.removeMin$
    Add $u$ to $S$
    for each outgoing edge $(u, v)$ from $u$ do
        if $v \notin S$ & $d[u] + w_{uv} < d[v]$ then
            $d[v] \leftarrow d[u] + w_{uv}$
            $Q.decreaseKey(v, d[v])$
            $\pi[v] \leftarrow u$
return $\pi, d$
```

Time complexity? Again, we can't just simplistically look at loops and so on. Instead, we count the total time taken by various operations.

- For any reasonable implementation of Q, creating it in the first place takes $\Theta(V)$ time since we are inserting vertices with known keys all at once.

- We end up calling *removeMin* once for each vertex, and Q has size at most $|V|$, which contributes $\Theta(V \cdot T_{remove}(V))$ (the PQ decreases each time, but half the calls are on a priority queue of size at least $V/2$, so we are still justified in saying it takes $\Theta(V \cdot T_{remove}(V))$).

- We also call *decreaseKey* once for each edge, which contributes $\Theta(E \cdot T_{decrease}(V))$.

- All other operations (adding to S, checking for membership in S, setting values in d and π) take $\Theta(1)$, which adds $\Theta(V + E)$ overall.

So in total, the algorithm takes $\Theta(V \cdot T_{remove}(V) + E \cdot T_{decrease}(V))$. We'll generally assume that $|E| \geq |V|$ (otherwise the graph is either a tree, in which case we don't need this algorithm, or there are degree-zero vertices which we can throw away). Total running time depends on implementation of Q:

- Sorted array: $\Theta(V + EV) = \Theta(EV)$.
- Array/dictionary: $\Theta(V^2 + E) = \Theta(V^2)$ (since $E < V^2$).
- Binary heap: $\Theta(V \lg V + E \lg V) = \Theta(E \lg V)$ (since we assume $|E| \geq |V|$).
- Fibonacci heap: $\Theta(V \lg V + E)$.

Fibonacci heap is fastest known implementation. But for simplicity of implementation and speed in practice, binary heap is best all-around.

16 (L) Minimum Spanning Trees (MSTs)

Input: a weighted, undirected graph $G = (V, E)$ (with weights in $\mathbb{R}^+$, *i.e.* nonnegative).
Output: A *minimum-weight spanning subgraph*: that is, a set of edges $T \subseteq E$ such that (V, T) is connected and T has the smallest total weight among all such spanning subgraphs.

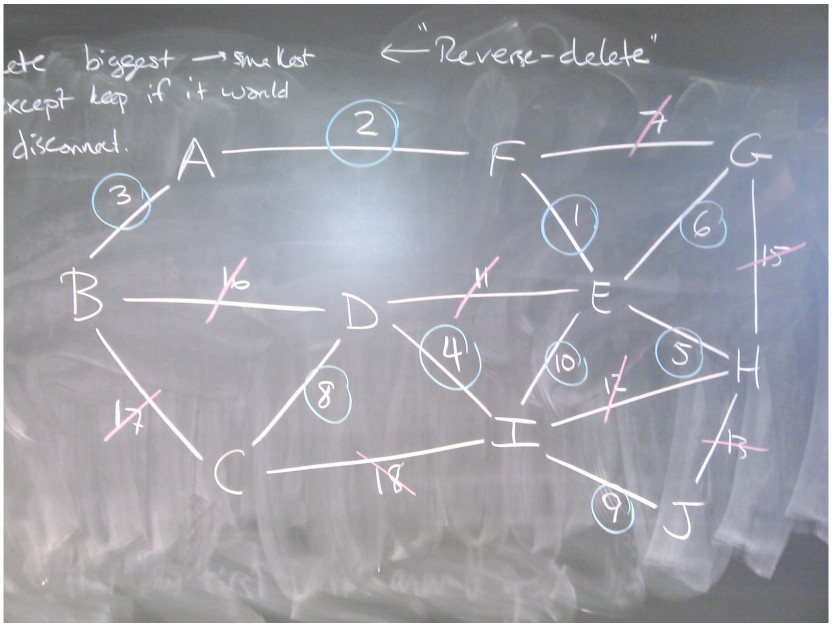
Applications: connect things with minimum cost (assuming no redundancy is needed), *e.g.* transportation or communication networks.

Observation 4. Any minimum-weight spanning subgraph (MWSS) is a tree.

Proof. A MWSS is connected by definition. If a spanning subgraph has a cycle, we can remove any edge from the cycle, resulting in a spanning subgraph that is still connected but with smaller weight. Hence any MWSS must be acyclic.

A MWSS is thus usually referred to as a *minimum spanning tree* (MST).

Given a graph, how can we compute a MST? Do an example, come up with greedy algorithms. Draw a second copy (have a student make the copy while drawing the first copy) and try a different algorithm.



- Kruskal: repeatedly pick the shortest edge that doesn't make a cycle.
- Prim: repeatedly pick the shortest edge that expands the growing tree.
- (Reverse-delete: repeatedly delete the biggest edge that doesn't disconnect the graph.)

Lots of greedy algorithms work! It's almost like we can't mess it up. How to prove this?

Lemma 16.1 (Cut Property)**.** *Let X, Y partition V and let $e = (x, y)$ be the shortest edge crossing the (X, Y) cut (that is, the shortest edge with $x \in X$ and $y \in Y$). Then e must be in any MST.*

Proof. Suppose we have a spanning tree T that does not include edge e; we will show that it is not a MST (and hence every MST must include e). Consider the unique path in T from x to y. It must cross the cut at least once, say at $e' = (x', y')$. Exchange e for e': the resulting graph is still connected, since any path that used to go through e' can now go through e. The resulting graph also has lower total weight, so T was not a minimum spanning tree. SDG

This is an **exchange argument**, which is a general technique when proving something is not minimal—find appropriate things to exchange so the total weight becomes smaller (while preserving any relevant properties). Note we can't exchange e with *any* edge across the cut! For example... We particularly found the edge on the path from u to v since that guarantees we can exchange it with e while preserving connectivity.

Theorem 16.2. *Kruskal's algorithm is correct.*

Proof. Suppose at some step the algorithm picks $e = (x, y)$. Take X to be the set of nodes connected to x so far (not including e); $x \in X$ by definition. $y \notin X$ since e would then make a cycle, and Kruskal wouldn't have picked it. By definition Kruskal picks the *smallest* such e. So the chosen edge must be in a MST by the cut property. SDG

The proof for Prim's algorithm is very similar; left as an exercise.

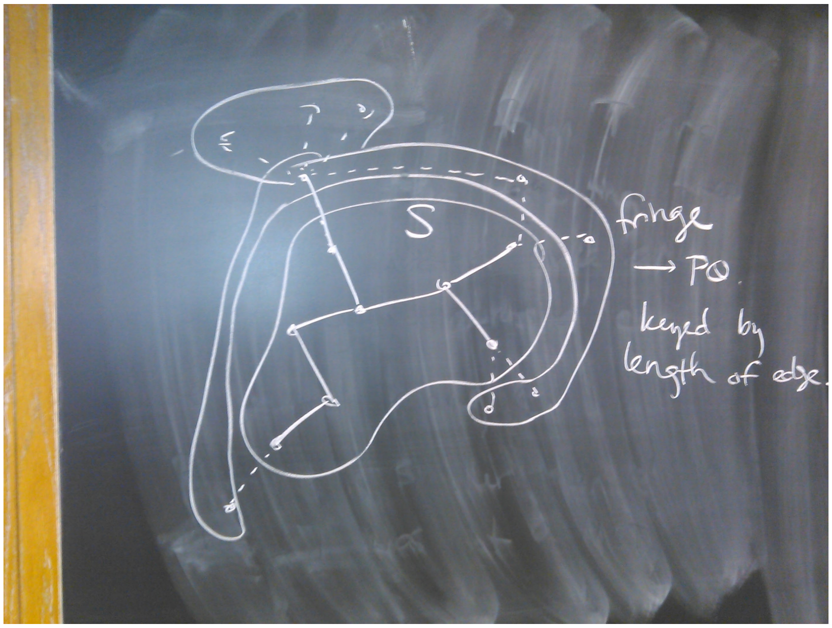
17 (L*) Implementing MST

Let's implement Prim's algorithm. We'll build the tree T. S is the set of vertices connected by the tree so far.

Algorithm 7 HighLevelPrim(G)

```
Require: Weighted, undirected, connected graph $G = (V, E)$.
$T \leftarrow$ empty tree
$S \leftarrow \{v\}$ (pick arbitrary starting vertex $v$)
while $|S| < n$ do
    $e \leftarrow$ smallest edge with one end in $S$ and one end not in $S$.
    Add $e$ to $T$.
    Add $v$ to $S$.
```

This is simple enough; by induction we can see that T will always be a tree and has S as the vertices connected by T. Clearly the important line is the one about picking e. How can we do that efficiently? Use a priority queue! Store "fringe" vertices (connected to S by one edge) keyed by weight of shortest edge to them from an edge in S.



Algorithm 8 Prim(G)

```
Require: Weighted, undirected, connected graph $G = (V, E)$.
 1: $S \leftarrow \{s\}$ (pick arbitrary starting vertex $s$)
 2: $fringe \leftarrow$ empty priority queue of vertices
 3: $\pi \leftarrow$ empty parent map
 4: for each neighbor $v$ of $s$ do
 5:     Add $v$ to $fringe$ using $w_{sv}$ as priority
 6:     $\pi[v] \leftarrow s$
 7: while $|S| < |V|$ do
 8:     $u \leftarrow fringe.removeMin$
 9:     Add $u$ to $S$
10:     for each edge $(u, v)$ with $v \notin S$ do
11:         if $v \in fringe$ then
12:             if $w_{uv} < fringe.priority(v)$ then
13:                 $fringe.decreaseKey(v, w_{uv})$
14:                 $\pi[v] \leftarrow u$
15:         else
16:             $fringe.add(v, w_{uv})$
17:             $\pi[v] \leftarrow u$
```

What's the running time of this algorithm?

- Lines 1–4 are all constant time.
- The loop at line 5 takes $O(V)$ time: lines 6 and 7 are constant-time operations, and s may have $O(V)$ neighbors.
- The while loop at line 9 executes $|V|$ times. Line 10 therefore contributes a total of $O(V \cdot T_{remove}(V))$ (depending on the priority queue implementation). Lines 11 and 12 are constant so they contribute a total of $O(V)$.
- Line 16 executes at most once per edge, so it contributes a total of $O(E \cdot T_{decrease}(V))$.
- Line 20 executes at most once per vertex, so it contributes a total of $O(V \cdot T_{add})$.

All together, then, this algorithm is $O(V + V \cdot T_{remove}(V) + E \cdot T_{decrease}(V) + V \cdot T_{add})$. If we use a binary heap-based priority queue implementation, $T_{remove}(V) = T_{decrease}(V) = T_{add} = \Theta(\log n)$, so this becomes $O(V + V \log V + E \log V) = O(E \log V)$. If we use a Fibonacci heap, $T_{remove}(V) = \Theta(\log n)$ but $T_{add} = T_{decrease}(V) = \Theta(1)$, so it becomes $O(V + V \cdot \log V + E + V) = O(E + V \log V)$. These running times are the same as Dijkstra's algorithm.

18 (L) Kruskal's Algorithm, Union-Find data structure

Recall Kruskal's algorithm for computing MST: consider edges in order from smallest to biggest, keep each edge unless it would create a cycle with edges already chosen. How to implement this?

First idea:

- Initialize T to be an empty graph.
- Sort the edges.
- For each edge (u, v), run a DFS from u in T in and see if we can reach v. If so, adding (u, v) would create a cycle so discard it; otherwise, add (u, v) to T.

How fast is this? Sorting the edges takes $\Theta(E \log E) = \Theta(E \log V)$. ($E$ is $O(V^2)$ so $\log E$ is $O(\log V^2) = O(2 \log V) = O(\log V)$.) Running a DFS in T takes $\Theta(V + E) = \Theta(V)$ (since T will never have more edges than vertices), and in this algorithm we run a DFS once for each edge, for a total of $\Theta(VE)$, which dominates the $\Theta(E \log V)$ time to sort the edges. Of course, $\Theta(VE)$ can be as big as $\Theta(V^3)$ if there are a lot of edges. Can we do better?

Yes, we can! Notice that for each edge (u, v), doing a DFS is sort of overkill, in the sense that it actually tries to *find a path* from u to v, but we don't care about the path, only whether u and v are already connected or not. In this case—when we only care about which vertices are connected and not about the actual paths between them—we can test for connectivity much faster.

The basic idea is to come up with some sort of data structure to maintain a set of connected components. Given such a data structure, the algorithm looks something like this:

- Start every vertex in its own connected component.
- For each edge, test whether its vertices are in the same component (which means it would form a cycle) or in different components.
- If the endpoints are in different components, add the edge to T, and merge the two components into one.

This kind of data structure is called a *union-find* structure, and from the algorithm above we can see what operations it needs to support:

- MakeSets(n): create a union-find structure containing n singleton sets.
- Find(v): return the *name* of the set containing v. ("Name" could be anything, typically we will use integers.) To check whether two vertices u and v are in the same connected component, we can test if Find(u) = Find(v).
- Union(x, y): merge the two sets whose names are x and y.

This has lots of applications! (See homework.)

Given such a data structure, we can implement Kruskal's algorithm as follows:

Algorithm 9 KRUSKAL(G)

```
Require: Weighted, undirected, connected graph $G = (V, E)$.
T ← empty set of edges
Sort the edges of $G$ by weight
$U \leftarrow$ MAKESETS($n$)
for each edge $e = (u, v)$ from smallest to biggest do
    if $U$.FIND($u$) $\neq$ $U$.FIND($v$) then
        Add $e$ to $T$
        $U$.UNION($u, v$)
```

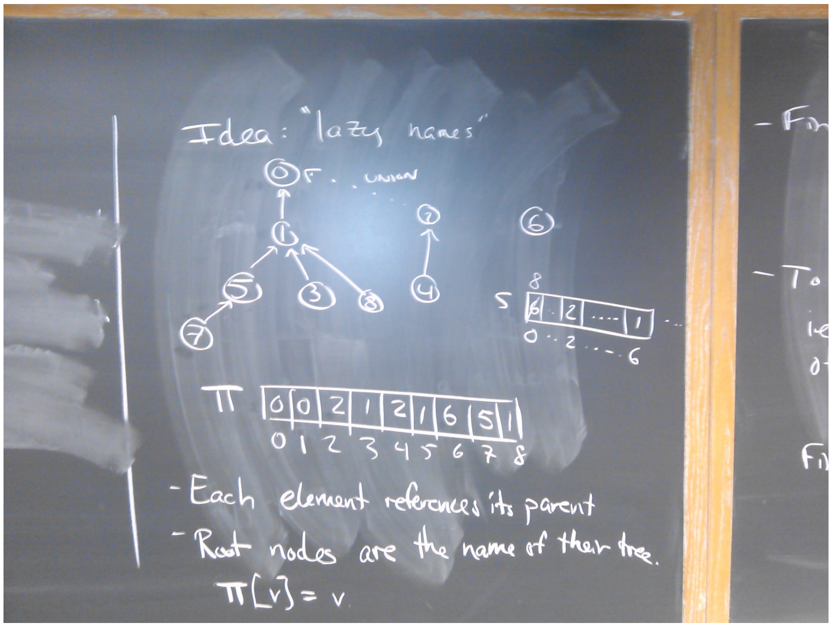
- Sorting edges by weight takes time $\Theta(E \log E) = \Theta(E \log V)$.
- We do $2|E|$ FIND operations.
- We do at most $|V| - 1$ UNION operations.

Hence, the overall time is $O(E \log V + T_{\text{MakeSets}} + ET_{\text{Find}} + VT_{\text{Union}})$. We would really like for T_{Find} and T_{Union} to be $O(\log V)$ (or better), which would make the whole thing $O(E \log V)$.

First idea: just keep a dictionary that maps each node to an "id" which identifies its set.

- FIND is $\Theta(1)$. (Assuming $\Theta(1)$ dictionary lookup.)
- But to do UNION we have to go through and change all the ids of one of the sets. This could be $O(n)$. Not good enough!

$\log n$, eh? This should make us think of trees. Idea: *forest* (multiple trees) of vertices where each points to its parent. (Parents don't need to know about their children.) We can represent this simply with an array/dictionary where $\pi[v] = p$ means p is the parent of v; by convention, $\pi[v] = v$ means v is a root. Each tree is a set; the root of the tree will be used as the name of the set. Nodes are given an id "lazily"—might not point directly to its id.



- Each element references its parent
- Root nodes are the name of their tree.
  $\pi[v] = v$

- To do Find, just follow pointers up to the root. That is, given v, look up $\pi[v]$, then $\pi[\pi[v]]$, and so on, until finding a root.
- To do Union(x, y), just merge the trees by setting one to be the parent of the other, that is, $\pi[x] \leftarrow y$. Note that this only works if x and y are already roots. (For convenience, one might also want to implement a version of Union which can work on any nodes, by calling Find on each first and then doing the union.)

Clearly Union is $\Theta(1)$, hooray! But what about Find? It seems like it might be $O(n)$ in the worst case, if we end up with an unbalanced tree. But if we are clever/careful in how we implement Union, this won't happen!

- Keep track of the size of each set (*i.e.* in a separate array/dictionary).
- When doing Union, always make the smaller set a child of the larger (and update the size of the larger in $\Theta(1)$).

Theorem 18.1. Find *takes* $\Theta(\log n)$ *time.*

Proof. The distance from any node up to its root is, by definition, the number of times its set has changed names. But the name of a node's set only changes when it is unioned with a *larger* set. So each time a set changes names, its size must at least double. The total size of a set can't be larger than n; hence the most time any element can have its set change names, and therefore its maximum depth, is $O(\log n)$. SDG

One can also implement *path compression*: when doing Find, update every node along the search path to point directly to the root. This does not make Find asymptotically slower, and it speeds up subsequent Find calls. One can show (although the proof is somewhat involved—it would probably take two lectures or so) that Find then takes essentially $\Theta(\lg^* n)$ on average, where $\lg^* n$ is *iterated logarithm* of n, defined as the number of times the lg function must be iterated on n before reaching a result of 1 or less. This means that the largest number for which $\lg^* n = k$ is $n = 2^{2^{2^{\cdots}}}$ with k copies of 2 stacked up in a tower of exponents! So, for example, $\lg^* n \leq 5$ for all $n \leq 2^{2^{2^{2^{2}}}} = 2^{2^{2^{4}}} = 2^{2^{16}} = 2^{65536}$, a number with 19729 decimal digits (for comparison, the number of particles in the entire universe is estimated at around 10^{80}, a number with a measly 81 decimal digits). So although *in theory* $\lg^* n$ is not a constant—it does depend on n, and can get arbitrary large as long as n is big enough—*in practice*, in our universe, it is essentially a constant (and a rather small constant at that). This means that both Find and Union can be implemented to run in (essentially) constant time!

Note, however, that this does not change the asymptotic running time for Kruskal's algorithm, which is still dominated by the $\Theta(E \log V)$ time needed to sort the edges.

19 Huffman coding

20 (P/L) Divide and Conquer: Master Theorem

Basic idea of divide & conquer:

- Break input into subproblems.
- Solve subproblems recursively.
- Combine subproblem solutions into overall solution.

When it works, this is a beautiful technique and very amenable to analysis:

- Implementation: recursion
- Correctness proof: induction
- Asymptotic analysis: recurrence relations

Classic example: mergesort. Look at call tree of mergesort to see why it is $\Theta(n \log n)$: the tree has height $\lg n$, and we do $\Theta(n)$ work merging at each level.

Integer multiplication

Input: two n-bit integers A, B.
Output: $A \times B$.

Note that *adding* two n-bit numbers takes $\Theta(n)$. The naive grade-school algorithm to multiply them takes $\Theta(n^2)$: multiply A by each bit of B in $\Theta(1)$ time (for each bit of B we either get 0 or a shifted copy of A), and then add the results—there are $\Theta(n)$ shifted copies of A to add, and each addition takes $\Theta(n)$, for a total of $\Theta(n^2)$.

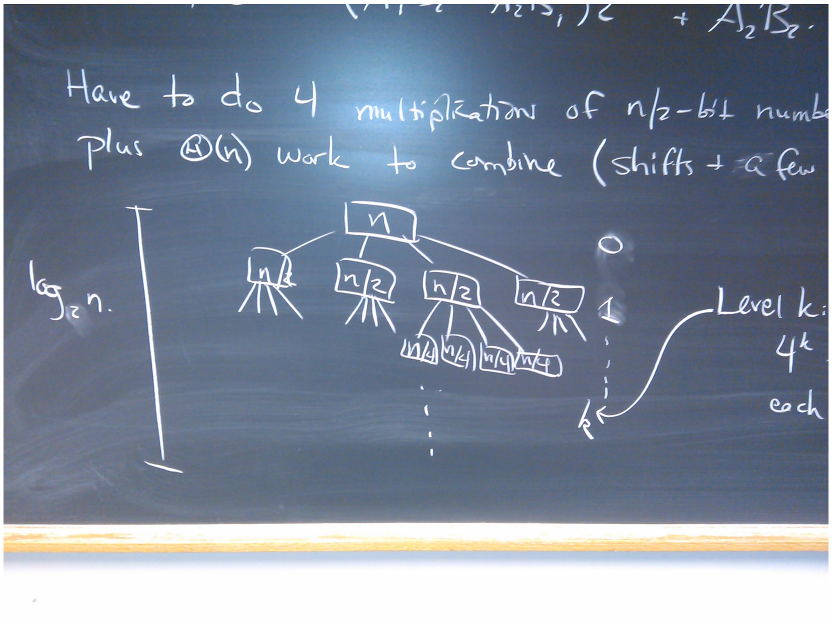
Let's try a divide and conquer approach. Divide each integer into two $n/2$-bit halves. (Assume n is a power of two—if not, we could always left-pad the numbers with zeros, which only increases the size by at most a factor of 2.) For example, if $A = 105_{10} = 01101001_2$ then $A_1 = 0110_2 = 6_{10}$ and $A_2 = 1001_2 = 9_{10}$. Note $6 \cdot 2^4 + 9 = 105$. In general, $A = A_1 \cdot 2^{n/2} + A_2$ and $B = B_1 \cdot 2^{n/2} + B_2$. Multiplying,

$$AB = (A_1B_1)2^n + (A_1B_2 + A_2B_1)2^{n/2} + A_2B_2.$$

So we have broken the original problem (multiplying two n-bit numbers) into four subproblems of size $n/2$ (*i.e.* four $n/2$-bit multiplications) plus some extra work to combine the results. Note the combining takes $\Theta(n)$ time (three additions at $\Theta(n)$ each, plus two shifts which take constant time). So if $T(n)$ represents the amount of time needed to multiply two n-bit numbers, we have the recurrence

$$\begin{aligned} T(1) &= \Theta(1) \\ T(n) &= 4T(n/2) + \Theta(n) \end{aligned}$$

We can unroll this into a recursion tree to figure out how much total work happens:



$+ A_2B_2.$
Have to do 4 multiplications of n/2-bit numbe
plus Θ(n) work to combine (shifts + a few
$\log_2 n.$
n
n/2
n/2
n/2
n/2
n/4
n/4
n/4
n/4
0
1
k
Level k:
4^k
each

- The tree has height $\log_2 n$.
- At depth k, there are 4^k recursive calls, each on a problem of size $n/2^k$.
- At each recursive call of size $n/2^k$ we do $\Theta(n/2^k)$ work.
- Hence the total amount of work at level k is $4^k\Theta(n/2^k) = 2^k\Theta(n)$.

Hence the total amount of work in the whole tree (noting that $2^{\log_2 n} = n$) is

$$\begin{aligned}\sum_{k=0}^{\log_2 n} 2^k\Theta(n) &= \Theta(n) + 2\Theta(n) + 4\Theta(n) + \cdots + n\Theta(n)\\ &= (1 + 2 + 4 + \cdots + n)\Theta(n)\\ &= (2n-1)\Theta(n) = \Theta(n^2).\end{aligned}$$

Argh! This turns out to be no better than our original naive algorithm. BUT it was a good exercise, and gives us insight into more general situations—not to mention this approach can be salvaged by doing something a bit more clever when we split up the problem into subproblems. But first, let's prove a more general theorem.

Lemma 20.1. $a^{\log_x b} = b^{\log_x a}$.

Lemma 20.2. *For some positive constant r and some variable x, let*

$$S = 1 + r + r^2 + \cdots + r^x = \frac{1 - r^{x+1}}{1 - r}.$$

- *If* $r < 1$*, then* S *is* $\Theta(1)$*.*
- *If* $r = 1$*, then* S *is* $\Theta(x)$*.*
- *If* $r > 1$*, then* S *is* $\Theta(r^x)$*.*

Theorem 20.3 (Master Theorem)**.** *If*

$$T(n) \leq aT(\lceil n/b \rceil) + O(n^d)$$

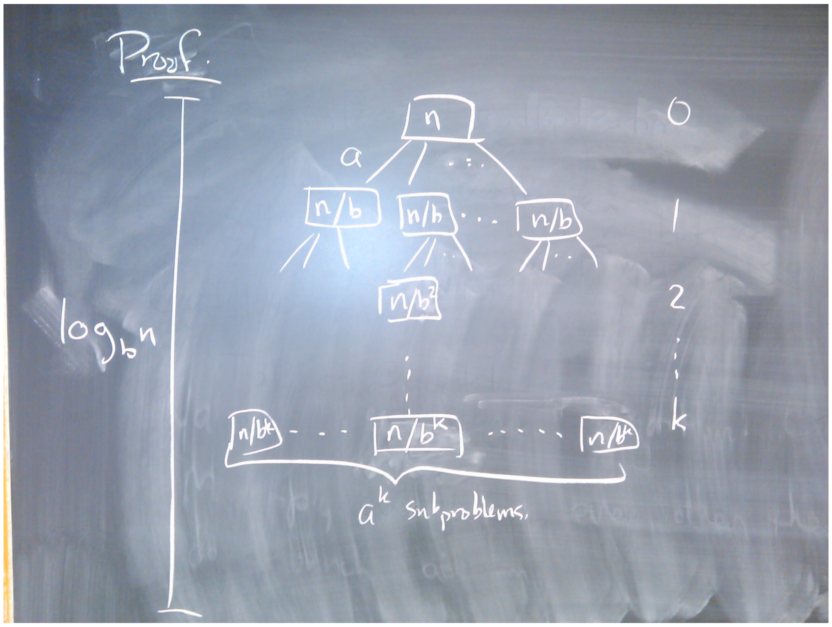
for positive constants a*,* b*, and* d*, then*

$$T(n) = \begin{cases} O(n^d) & a < b^d \\ O(n^d \log n) & a = b^d \\ O(n^{\log_b a}) & a > b^d. \end{cases}$$

Intuitively, a tells us how fast the number of recursive calls grows; b^d tells us how fast the problems are getting smaller. The ratio of these will end up being the common ratio of a geometric sum.

- If $a < b^d$, then the problems get smaller faster than the number of problems increases, and the total amount of work is dominated by the work done at the very top of the recursion tree.
- If $a > b^d$, then the number of nodes is growing faster than the work decreases, so the amount of work is dominated by the bottom level of the tree (as in our integer multiplication example).
- If $a = b^d$, then the growth of the number of subproblems is exactly balanced by the decrease in the amount of work, so there is exactly the same amount of work done in total at each level of the recursion tree (namely, $O(n^d)$), and the overall total is hence this amount of work per level times the number of levels (as in merge sort).

Proof. We begin by drawing the recursion call tree:



- The tree has height $\log_b n$.
- A node at level k has size n/b^k and does $O((n/b^k)^d)$ work.
- There are a^k nodes at level k, so the total work at level k is

$$a^k \cdot O\left(\left(\frac{n}{b^k}\right)^d\right) = O(n^d)\left(\frac{a}{b^d}\right)^k.$$

Hence the overall total work is

$$\sum_{k=0}^{\log_b n} O(n^d)\left(\frac{a}{b^d}\right)^k = O(n^d)\sum_{k=0}^{\log_b n}\left(\frac{a}{b^d}\right)^k.$$

This is $O(n^d)$ times a geometric series with ratio a/b^d.

- If $a < b^d$ then the ratio is < 1 and the sum is a constant; hence the total work is $O(n^d)$.
- If $a = b^d$ then the ratio is 1 and the sum is $O(\log_b n)$; hence the total work is $O(n^d \log n)$.
- If $a > b^d$ then the ratio is > 1 and the sum is proportional to its final term, $(a/b^d)^{\log_b n}$. In that case the total work is

$$O\left(n^d \cdot \left(\frac{a}{b^d}\right)^{\log_b n}\right) = O\left(n^d \cdot \frac{a^{\log_b n}}{(b^{\log_b n})^d}\right) = O(a^{\log_b n}) = O(n^{\log_b a}).$$

SDG

21 (L*) Applications of divide & conquer, intro to FFT

Karatsuba's algorithm

Now, back to integer multiplication! In 1960 it seemed "obvious" that integer multiplication was $\Omega(n^2)$; Andrey Kolmogorov (a really huge name in mathematics) posed it as a conjecture. Then Anatoly Karatsuba disproved the conjecture by coming up with a faster algorithm!

Break up A and B into two pieces of size $n/2$ as before. Now for Karatsuba's clever insight. Define

$$\begin{aligned} P_1 &= A_1 B_1 \\ P_2 &= A_2 B_2 \\ P_3 &= (A_1 + A_2)(B_1 + B_2) \end{aligned}$$

Note $P_3 - P_1 - P_2 = A_1 B_2 + A_2 B_1$. Hence

$$AB = P_1 2^n + (P_3 - P_1 - P_2) 2^{n/2} + P_2.$$

This requires only *three* multiplications of size $n/2$! (Along with two additions of size $n/2$, four additions or subtractions of size n, and two shifts—more work than before, but all still $\Theta(n)$ in total.)

Hence $T(n) = 3T(n/2) + \Theta(n)$, so we can apply the Master Theorem with $a = 3$, $b = 2$, and $d = 1$. $3 > 2^1$ so we are in the third case of the theorem (the work is concentrated at the bottom of the call tree), and we conclude that this algorithm is $O(n^{\log_2 3}) \approx O(n^{1.59})$.

This is not even the fastest known algorithm—the fastest algorithm actually used in practice is the Schönhage-Strassen algorithm, which can multiply two n-bit integers in $O(n \log n \log \log n)$. [Q: what does a theoretical computer scientist say when drowning? A: log log log . . .]

Matrix multiplication

Recall how matrix multiplication works. Given $n \times n$ matrices X and Y, we want to compute XY where

$$(XY)_{ij} = \sum_k X_{ik} Y_{kj}.$$

(Incidentally, this is much more important than it might seem: there are a whole host of linear algebra operations which can be reduced to doing matrix multiplication; there are lots of algorithms, *e.g.* graph algorithms (remember adjacency matrices?) that can be similarly reduced. . .)

How long does this take? Obvious algorithm is $\Theta(n^3)$: three nested loops (for each of the n^2 elements of the output array, we do n multiplications and n

additions). So we can say matrix multiplication takes $O(n^3)$. Also, it is clearly $\Omega(n^2)$ since the output has size n^2. But it seems "obvious" that we can't do any better than n^3... can we?

In fact, lots of people used to think $\Theta(n^3)$ was the best possible, until Volker Strassen made a surprising discovery—a divide-and-conquer algorithm faster than the naive $\Theta(n^3)$!

The basic idea boils down to a "trick" (similar in spirit to Karatsuba's algorithm) for computing the product of 2×2 matrices using only 7 multiplications instead of 8.

$$XY = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

Computing the result directly, as above, obviously requires eight multiplications (and four additions).

Now define:

$$\begin{aligned} p_1 &= a(f - h) \\ p_2 &= (a + b)h \\ p_3 &= (c + d)e \\ p_4 &= d(g - e) \\ p_5 &= (a + d)(e + h) \\ p_6 &= (b - d)(g + h) \\ p_7 &= (a - c)(e + f) \end{aligned}$$

Computing all the p_i requires 10 additions and 7 multiplications. Now, as you can check,

$$XY = \begin{bmatrix} p_5 + p_4 - p_2 + p_6 & p_1 + p_2 \\ p_3 + p_4 & p_1 + p_5 - p_3 - p_7 \end{bmatrix}.$$

All told, we have now computed XY using 18 additions and 7 multiplications. This is a terrible algorithm for multiplying actual 2×2 matrices! But we can turn it into a recursive divide-and-conquer algorithm for multiplying large matrices.

Assume X, Y are $n \times n$ matrices where n is a power of 2. Break each one into four $n/2 \times n/2$ submatrices ("blocks"). That is,

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \qquad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

where $A, \dots, H$ are $n/2 \times n/2$ matrices. It turns out that matrix multiplication works the same way on these blocks as it does for actual 2×2 matrices, that is,

$$XY = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

(For proof, see any linear algebra textbook, or just think about it a bit.)

If we do the obvious thing here and make 8 recursive calls, note that we have $T(n) = 8T(n/2)+O(n^2)$, so we can apply the Master Theorem with $a = 8$, $b = 2$, and $d = 2$. Since $8 > 2^2$ the algorithm takes $O(n^{\log_b a}) = O(n^{\log_2 8}) = O(n^3)$: this is just the naive algorithm.

But instead, we can use the above method for multiplying 2×2 matrices: this results in only seven recursive calls, and a constant number of extra matrix additions which still take $O(n^2)$ overall. So now $a = 7$, $b = 2$, and $d = 2$: we still have $7 > 2^2$ so the algorithm is $O(n^{\log_2 7}) \approx O(n^{2.81})$.

Although asymptotically faster than "naive" matrix multiplication, Strassen's algorithm is

- numerically less stable
- only faster for $n > 1000$ or so, because of the overhead of extra additions and so on.

But it's actually used in practice, especially for multiplying very large matrices when numerical stability is not an issue (*e.g.* over finite fields).

Strassen's breakthrough spurred a lot more research into the problem. The currently best known asymptotic complexity is about $O(n^{2.37})$, but such algorithms are not used in practice because they would only be faster for astronomically large matrices.

Introduction to the Fast Fourier Transform

Problem: multiplying two *polynomials* of degree n,

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n,$$

where the a_i are complex numbers (really, any field will do).

How fast can we multiply two such polynomials?

- Using a simple naive algorithm we can clearly achieve $O(n^2)$.
- We can use Karatsuba's trick to achieve $O(n^{\log_2 3})$.

But it turns out we can do even better, using the *Fast Fourier Transform* (FFT). The FFT is one of the most important algorithmic developments of the 20th century, and has tons of applications in engineering, physics, chemistry, astronomy, geology, signal processing... Particular applications you may be familiar with include encoding and decoding DVDs, JPEGs, and MP3s, as well as speech processing (*i.e.* every time you speak to your phone or your computer, it probably runs FFT).

Operations on polynomials, represented in the usual way by a list of $n + 1$ coefficients $a_0, a_1, \ldots, a_n$:

- Addition: $O(n)$.
- Evaluation: $O(n)$ using *Horner's method*: $a_0 + x(a_1 + x(a_2 + \cdots + x(a_n) \ldots))$
- Multiplication (*convolution*): $O(n^2)$ brute force.

$$a_0 b_0 + (a_0 b_1 + a_1 b_0)x + (a_0 b_2 + a_1 b_1 + a_2 b_0)x^2 + \ldots$$

Let's consider a different representation of polynomials, based on the

Theorem 21.1 (Fundamental Theorem of Algebra)**.** *Any nonzero degree-n polynomial with complex coefficients has exactly n complex roots.*

Corollary 21.2. *A degree-n polynomial is uniquely specified by its value at $n+1$ distinct x-coordinates.*

Proof. If f and g are two degree-n polynomials which have the same value at each of $n+1$ distinct x-coordinates, consider the polynomial $f-g$: it has $n+1$ roots but degree $\leq n$; the only way for this to happen is if $f-g=0$, that is, $f=g$. $\boxed{\text{SDG}}$

So another way to "represent" a degree-n polynomial is by a list of $n+1$ pairs $(x_i, f(x_i))$, *i.e.* some choice of $n+1$ distinct x-coordinates along with the value of the polynomial at each.

How fast can we do operations given this representation?

- Addition: $O(n)$. Just add corresponding y-values.
- Multiplication: $O(n)$. Just multiply correpsonding y-values! Actually, there is a subtlety here: the resulting polynomial may have degree $2n$, so we need to make sure we have values of the polynomials for at least $2n+1$ points to start. (No big deal.)
- Evaluation: actually $O(n^2)$ using Lagrange's formula.

The point is that these two representations represent a tradeoff: do we want multiplication to be fast, or evaluation?

And this raises a natural question: *how fast* can we convert between the two representations? If we can convert faster than $O(n^2)$ then we win!

$$\begin{tikzcd} a_0, a_1, \dots, a_n \arrow[r, shift left] & (x_0, y_0) \dots (x_n, y_n) \arrow[l, shift left] \end{tikzcd}$$

fast eval   fast multiply

This is what FFT does: it can convert in $O(n \log n)$. So, for example, to multiply two polynomials represented by their coefficients, we can convert to the set-of-points representation in $O(n \log n)$, multiply in $O(n)$, and convert back in $O(n \log n)$, for a total time of $O(n \log n)$. The key will be to cleverly pick the x_i we will use as our points at which to evaluate the polynomial.

22 (L*) FFT

Details of FFT; see slides.

23 (L) Median/select

Remember the 3 components of divide & conquer:

- *Divide* input into subproblems somehow (split, partition, . . .)
- *Recursively* solve the subproblems (*have faith* they are correct!)
- *Combine* the answers

Don't forget base case(s) as well.
To prove (by induction on size of input!):

- Prove base case(s) are correct
- Prove: *if* answers to subproblems are correct, *then* answer to whole problem is correct.

Array Median

Input: array of n integers.
Output: the median element, *i.e.* the $\lfloor n/2 \rfloor$-smallest.

In groups: what's the first algorithm you think of? How long does it take? (Probably sort + return middle, $\Theta(n \lg n)$.)

Can we do better? Get ideas.

Problem: if we *e.g.* split the array in half somehow and get the median of both, that wouldn't really help. The medians of the subarrays are not likely to have any relationship to the median of the whole.

Solution: generalize! Let's find not just the median, but take k as an input and return the kth-smallest. $\textsc{Select}(A, k)$: return the kth-smallest element of array A. We can get median by calling $\textsc{Select}(A, \lfloor n/2 \rfloor)$.

Question: how should we split up the array? Just splitting down the middle doesn't really help. What's a different way we could split up A (think in groups). Hint: think of other divide & conquer algorithms you know. (Quicksort.)

Solution: pick a pivot element p (random!) and partition into A_1 = all elements $< p$ and A_2 = all elements $\geq p$. Then if $k < |A_1|$ return $\textsc{Select}(A_1, k)$, else return $\textsc{Select}(A_2, k - |A_1|)$.

Prove correct. Analyze time complexity.

24 (P/L) Intro to dynamic programming

"Dynamic programming"—not really either. Name chosen by Richard Bellman in 1950 as something that sounded good to government / funding agencies.

Last week we looked at the divide and conquer technique: a problem gets broken down recursively into subproblems. We're still considering the same phenomenon—dynamic programming is about what to do when the recursive subproblems *overlap*. Basic idea: *save* answers to recursive subproblems (*memoization*) so we don't have to compute them more than once. DP has a reputation for being difficult/confusing, but at heart it's just this simple idea.

Example: Fibonacci numbers

Recall $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$, each number is the sum of the previous two.

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

Obvious recursive algorithm is too slow (in fact, it's $O(\varphi^n)$).

```
# Fibonacci #1: naive recursive algorithm
def fib1(n):
    if n <= 1:
        return n
    else:
        return fib1(n-1) + fib1(n-2)
```

Why is that? Draw out recursion tree for F_5. Notice many redundant calls to subproblems. The core idea of dynamic programming is extremely simple: save the results of recursive subproblems so each only needs to be computed once. *Memoization.*

We have two options (show `fib.py`):

- Create an array to hold F_n values and fill it in from 0 to n using a loop. This is the "standard" DP solution. Pros: Efficient, works well even in languages without good support for recursion. Con: we have to manually figure out the correct order to fill in the array, so we have already computed the answer to subproblems when we need them. Simple in this case, but can be tricky in general.

  ```
  # Fibonacci #2: explicitly filling in a table with a loop
  def fib2(n):
      fibs = [0] * (n+1)
      fibs[1] = 1
  ```

```
    for i in range(2, n+1):
        fibs[i] = fibs[i-1] + fibs[i-2]

    return fibs[n]
```

- Keep our recursive function, but every time it is called, check whether the answer for that input has already been computed and saved. If so, return it; if not, compute it recursively, save it, and return it. Pros: simple to code; we don't have to worry about the right order in which to fill things in. Cons: a bit more overhead; can run into recursion limits. (Plug for functional programming: lazy, immutable arrays…)

```
# Fibonacci #3: recursion with memoization

# Keep a global table to remember the results of fib3
fibtable = [0,1]

def fib3(n):

    # Expand the table as necessary
    while len(fibtable) < n+1:
        fibtable.append(-1)

    # Fill in the table recursively (only if necessary)
    if fibtable[n] == -1:
        fibtable[n] = fib3(n-1) + fib3(n-2)

    return fibtable[n]
```

Example: Low/High Stress Jobs

Consider the following table composed of n weeks where each week i has low stress job that pays L_i and a high stress job that pays H_i.

WEEK	1	2	3	$\cdots$	n
low stress	L_1	L_2	L_3	$\cdots$	L_n
high stress	H_1	H_2	H_3	$\cdots$	H_n

Each week you are allowed to pick either a low stress job or a high stress job, however, picking a high stress job at week i means that you must take the week before (i.e. week $i-1$) off. Your goal is to maximize total income.

We've studied greedy algorithms; definitely the thing to try first. What would a greedy algorithm look like? See Algorithm 13 below.

This algorithm basically looks ahead one week and decides if it's worth taking a week off in order to land a higher-paying job. However, this strategy fails, because taking a high stress job at week h_i means that you cannot take week h_i off to take a job at week h_{i+1}. Thus, any strategy employing look-ahead by a constant factor will fail. Here is a counter-example to the greedy algorithm.

Algorithm 10 GreedyJob

```
while $i \leq n$ do
    if $i < n$ and $H_{i+1} > L_i + L_{i+1}$ then
        Take week $i$ off, choose $H_{i+1}$ and continue with $i \leftarrow i + 2$
    else
        Take $L_i$ and continue with $i \leftarrow i + 1$
```

WEEK	1	2	3
Low Stress	2	2	1
High Stress	1	5	10

- Greedy: 5 (week 2) + 1 (week 3) = 6
- Optimal: 2 (week 1) + 10 (week 3) = 12

The key to solving this problem is to come up with a recursive solution, and then use dynamic programming. The key idea (common to many similar problems) is to consider the most we could make if we worked *only* through week i and then stopped. That is, define $OPT(i)$ = maximum revenue for working weeks $1 \ldots i$. Now, we can come up with a recurrence for OPT:

1. Base cases:
    - $OPT(0) = 0$. We don't make any money for working 0 weeks.
    - $OPT(1) = L_1$. We can't take the high-stress job the first week. (Or maybe we can—need to clarify problem parameters!)
2. $OPT(i) = \max\{L_i + OPT(i-1), H_i + OPT(i-2)\}$. First, note that if we are going to maximize the profit for the first i weeks, we should always work the final week. Thus, our decision is between working a high-stress or low-stress job. Choosing a low-stress job at week i means we should add L_i to the optimal profit for the previous $i-1$ weeks and choosing a high-stress job at week i means we had to take week $i-1$ off so we should add H_i to $OPT(i-2)$. The optimal choice is to choose the max of these two options.

Notice that OPT naturally forms a $1 \times (n+1)$ table where each entry requires $\Theta(1)$ operations to fill in. We can fill in the table from 0 to n. So the whole algorithm is $\Theta(n)$. This is the standard way to analyze the running time of a DP solution.

We can also keep a table $JOB(i)$ that tells us which choice we made at week i (the low stress job or the high stress job) to recover which weeks we should actually work. Start with choice at week n and work backwards through the table. This technique is also standard. In more generality, typically we will be taking some sort of max or min, but this gives us only the *value* of the max/min and forgets *which* choice actually leads to it. We can always make a parallel table that records the optimal choice. In many cases (when there are exactly two choices) it will be a table of booleans. But we will see other examples where it is a table of *e.g.* integers.

Do an example:

WEEK	1	2	3	4	5	6	7	8	9	10
Low Stress	2	2	1	7	5	20	3	19	10	13
High Stress	1	5	10	100	23	20	5	21	30	30

Show `jobs.py`:

```
def work_schedule(low, high):
    n = len(low)
    opt = [0] * n
    take_high_job = [False] * n

    # If we’re only working one week, take the low-stress job.
    opt[0] = low[0]
    take_high_job[0] = False

    for i in range(1, n):

        # How much could we make taking the low or high stress job?
        low_total  = low[i] + opt[i-1]
        high_total = high[i] + (opt[i-2] if i > 1 else 0)

        # The optimal for weeks 1..i is the higher of the two
        opt[i] = max(low_total, high_total)

        # Record which choice produced the higher total
        take_high_job[i] = high_total > low_total

    # Finally, produce a work schedule: work backwards from the end
    wk = n-1
    schedule = []
    while wk >= 0:
        if take_high_job[wk]:
            schedule = [’off’, ’HI’] + schedule
            wk -= 2
        else:
            schedule = [’LO’] + schedule
            wk -= 1

    return (opt[-1], schedule)
```

25 (L*) Matrix chain multiplication

Suppose we have matrices A and B, where A is $p \times q$ and B is $q \times r$. (To be able to multiply them, the q has to match.) How many operations are needed to compute the matrix product AB?

- The result AB will be a $p \times r$ matrix, so it has pr entries.
- To compute each entry of AB, we take a row of q entries from A and a column of q entries from B and multiply them, then add the results. So we do about q multiplications and q additions to compute each element of AB.

Thus, the total time to compute AB is $O(pqr)$. Note that we can assume p, q, and r are small enough that fancy matrix multiplication algorithms like Strassen's algorithm don't really help.

Matrix multiplication is associative, $(AB)C = A(BC)$, but these may not take the same time to compute! Let's try an example. Say

- A is 2×10,
- B is 10×3,
- and C is 3×20.

It takes $10 \times 3 \times 20 = 600$ operations to compute BC, resulting in a 10×20 matrix. It would then take another $2 \times 10 \times 20 = 400$ to compute $A(BC)$, for a total of 1000. On the other hand, if we associate the product as $(AB)C$, it takes $2 \times 10 \times 3 = 60$ operations to compute the 2×3 matrix AB, and then only another $2 \times 3 \times 20 = 120$ operations to compute $(AB)C$, for a total of 180—a big difference!

With just three matrices the situation is simple: we can easily just do these computations, compare, and choose the cheaper order. However, if we have a sequence of n matrices we wish to multiply, the situation is more difficult. For example, suppose we have six matrices $A_1 \dots A_6$ and want to compute their product. We could associate them, for example, as $(A_1((A_2(A_3A_4))A_5))A_6$, or as $A_1(A_2(A_3(A_4(A_5A_6))))$, or in fact in any of 42 distinct parenthesizations. We really don't want to check all of them to see which would be the cheapest. In general, the number of parenthesizations for n matrices is the $(n-1)$st *Catalan number*

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

which for large n is approximately equal to

$$C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}},$$

so the number of possibilities becomes very large, very fast. The brute force algorithm—*i.e.* checking all parenthesizations to see which is best—is completely out of the question!

Formally, we have a sequence of n matrices A_1, A_2, ..., A_n, and a sequence of $n+1$ positive integers $p_1, \ldots, p_{n+1}$ such that matrix A_i has size $p_i \times p_{i+1}$. We want to compute the parenthesization of the A_i which minimizes the number of operations needed to compute the product $A_1 \ldots A_n$. (Note that this generalizes readily to any situation where we have a sequence of n things and an associative binary operation where the cost of the operation varies depending on the arguments it is applied to.)

Observation: ultimately, there will be some final two matrices that get multiplied to produce the final answer. These have to come from some splitting of the sequence of matrices into two subproducts

$$(A_1 \ldots A_k)(A_{k+1} \ldots A_n).$$

The total cost to compute the product if we split at index k is then given by the sum of the optimal cost to compute $A_1 \ldots A_k$, the optimal cost to compute $A_{k+1} \ldots A_n$, and the cost to do the final matrix multiplication, which will take $p_1 p_{k+1} p_{n+1}$ operations (because $(A_1 \ldots A_k)$ is a $p_1 \times p_{k+1}$ matrix, and $(A_{k+1} \ldots A_n)$ is a $p_{k+1} \times p_{n+1}$ matrix). The best possible cost for the product $A_1 \ldots A_n$ will then be the minimum cost over all such splitting points k. In general, if we use $m[i, j]$ to denote the minimum cost for computing the product $A_i \ldots A_j$, with $i \leq j$, then we have the recurrence

$$\begin{aligned} m[i, i] &= 0 \\ m[i, j] &= \min_{i \leq k < j} (m[i, k] + m[k+1, j] + p_i p_{k+1} p_{j+1}) \end{aligned}$$

Note that $m[i, i] = 0$ since this corresponds to the base case of already having a single matrix A_i, so we do not need to do any work.

We can store the $m[i, j]$ values in an $n \times n$ matrix (actually, we just need the part above the main diagonal since we assume $i \leq j$). Note that each $m[i, j]$ depends on values to its left ($m[i, k]$ where $k < j$) and below it ($m[k+1, j]$ where $k \geq i$). So we can fill in the matrix by diagonals, beginning by filling in zeros along the entire main diagonal, then filling in values just above the main diagonal, then the second diagonal above the main diagonal, and so on. Note that the dth diagonal above the main diagonal consists of values of the form $m[i, i+d]$, which corresponds to subproducts $A_i \ldots A_{i+d}$ of exactly $d+1$ matrices. So this seems intuitively sensible: first we compute the optimal way to multiply any two adjacent matrices; then we compute the optimal way to multiply any three adjacent matrices; then any four, and so on.

(Alternatively, we could arrange the matrix a bit differently, so that $m[i, d]$ would denote the optimal cost for multiplying $A_i \ldots A_{i+d}$. Then we would simply fill in the matrix by columns.)

The matrix m has size $n \times n$, and we need to fill in half its entries, giving a total of $\Theta(n^2)$ entries to fill in. Each entry is computed as a minimum over at most n costs, each of which takes $O(1)$ to compute (just a few lookups, additions, and multiplications). Thus the whole algorithm is $O(n^3)$.

Notice this has a similar problem as the high/low-stress jobs example: it gives us the optimal cost but doesn't tell us what the actual best parenthesization is. This is a common issue with dynamic programming; the problem is that when we take the minimum over all k for each entry in the matrix m, we forget which k was best, recording only the best cost itself. The solution, therefore, is to maintain another $n \times n$ matrix b which records the information that is otherwise being forgotten: $b[i, j]$ records the "best split", that is, the value of k (where $i \leq k < j$) which results in the minimum cost for the product $A_i \dots A_j$. After filling in the matrices m and b, we can reconstruct the best parenthesization (which is really just a binary tree) by recursively splitting starting from the top: we start by looking up $b[1, n] = k$ which tells us where to do the top-level split $(A_1 \dots A_k)(A_{k+1} \dots A_n)$. We then recursively look up $b[1, k]$ and $b[k+1, n]$ to find out where to make the next splits, and so on, until there are just single matrices left at the leaves of the tree.

See `MatrixChainFull.java` for Java implementation.

26 (L) The Floyd-Warshall algorithm

Recall that Dijkstra's algorithm solves the *single-source* shortest path problem (*i.e.* it finds the shortest path from a single start vertex to every other vertex) for weighted, directed graphs, as long as all edge weights are positive. Today we will consider an algorithm to solve the *all-pairs* shortest path problem (find shortest path between all possible pairs of nodes) on directed graphs with arbitrary (possibly negative) weights.

Of course, negative cycles can pose a problem: if there is a directed cycle whose total weight is negative, then we can keep decreasing the path weight forever by just going around the cycle. In that case "shortest" paths may not be well-defined. The algorithm we consider will also be able to detect this situation.

Input: a directed, weighted graph $G = (V, E)$ with vertices numbered $1 \dots n$. Output: ultimately, we want an $n \times n$ matrix s where $s[u, v]$ records the length of the shortest path from u to v, *or* an indication that the graph has negative cycles. Ideally we also want to be able to recover the actual shortest path between any two vertices u and v.

We want to come up with a recurrence. As is typical, we add another parameter which we use to restrict the problem. In this case let's add another parameter k that means *we are only allowed to use vertices* $1 \dots k$. That is, let $s[u, v, k]$ denote the length of the shortest path from u to v *using only vertices* $1 \dots k$ *as intermediate nodes on the path.*

Base cases?

$$s[u, v, 0] = \begin{cases} 0 & u = v \\ w_{uv} & (u, v) \in E \\ \infty & \text{otherwise} \end{cases}$$

(Note we can't say $s[u, v, k] = 0$ for $k \geq 0$ because of negative edges!)

Now, what about the recursive case? Suppose we already know $s[u, v, k]$ for some k. Then how can we compute $s[u, v, k + 1]$? There are two possibilities: either the best path from u to v goes through $k + 1$, or it doesn't.

- It could be that allowing vertex $k + 1$ does not help, in which case

$$s[u, v, k + 1] = s[u, v, k].$$

- If it does help, the best path from u to v will consist of first taking the best path from u to $k+1$ (using only vertices $1 \dots k$), followed by the best path from $k + 1$ to v:

$$s[u, v, k + 1] = s[u, k + 1, k] + s[k + 1, v, k].$$

The optimal cost will simply be the minimum of these two.

To be able to reconstruct paths after the fact, we could do something similar to what we usually do: at each step we take a minimum over two choices, so we could keep a parallel 3D table $\mathit{use}[u, v, k]$ that records whether we should use vertex k when trying to go from u to v using only vertices $1 \dots k$. We could then use this to reconstruct the optimal path from u to v as follows:

$$\mathit{bestpath}(u, v, k) = \begin{cases} \mathit{bestpath}(u, k, k-1) + [k] + \mathit{bestpath}(k, v, k-1) \\ \mathit{bestpath}(u, v, k-1) \end{cases}$$

depending on whether $\mathit{use}[u, v, k]$ is true (with some appropriate base cases as well).

However, for this problem it turns out we can do something a bit more clever using only a 2D array. In particular, let $\mathit{next}[u, v]$ denote the *next* vertex after u in the shortest path from u to v that we have found so far. Then in the first case, when we set $s[u, v, k+1]$ to $s[u, v, k]$, we do not change $\mathit{next}[u, v]$; in the other case, we can update it as

$$\mathit{next}[u, v] = \mathit{next}[u, k+1].$$

Also, we don't actually need to keep a 3D array for s. We can simply keep overwriting the same 2D array on each iteration of the loop. If we do that, it is no longer true that $s[u, v]$ is exactly the shortest distance using only vertices $1 \dots k$ on the kth iteration—but if not it can only be *shorter*.

Algorithm 11 Floyd-Warshall

```
$s[u,v] \leftarrow \begin{cases} 0 & u = v \\ w_{uv} & (u,v) \in E \\ \infty & \text{otherwise} \end{cases}$
$\mathit{next}[u,v] \leftarrow v$ for $(u,v) \in E$
for $u \leftarrow 1 \dots n, v \leftarrow 1 \dots n, k \leftarrow 1 \dots n$ do
    if $s[u,k] < \infty \wedge s[k,v] < \infty \wedge s[u,k] + s[k,v] < s[u,v]$ then
        $s[u,v] \leftarrow s[u,k] + s[k,v]$
        $\mathit{next}[u,v] \leftarrow \mathit{next}[u,k]$
```

After the algorithm is finished, to reconstruct the shortest path from u to v, we simply look up each step along the path using the *next* matrix. First we look up $\mathit{next}[u, v] = u_2$, then we look up $\mathit{next}[u_2, v] = u_3$, and so on, until reaching v.

What is the running time? It's obviously $\Theta(V^3)$—just three nested loops from $1 \dots n$. (Which seems kind of amazing given that there could be $\Theta(V^2)$ edges and we need to find shortest paths between *every* pair of vertices, of which there are also $\Theta(V^2)$.)

What about negative cycles? Well, just look at $s[u, u, n]$: u is part of a negative cycle if and only if $s[u, u, n]$ (the length of the shortest path from u back to itself) is negative. Hence we can check whether there are *any* negative cycles in the graph in $\Theta(n)$ time, by scanning along the diagonal of the matrix and looking for any negative numbers.

As a fun aside, this algorithm is really cool because it can actually be generalized to work over any semiring instead of just $(\min, +)$ (technically, any star-semiring), and it turns out that by appropriate choice of semiring, (a slight generalization of) the same algorithm can be used to do a great many things, such as find most reliable paths or largest capacity paths, count the number of shortest paths or total number of paths or even compute regular expressions for all possible shortest paths, compute transitive closures, invert matrices, solve linear systems of equations, or convert a DFA into a regular expression.

27 (P/L) Amortized analysis: intro

Consider the following problem.

Input: an array $B[0\dots]$ representing a binary number n. (Each $B[i]$ is a single bit representing the coefficient of 2^i. Assume B is sufficiently large so we don't have to worry about problems with overflow.)
Output: *Increment* the binary number, that is, modify B so that it represents the number $n+1$.

Here is an algorithm to accomplish this:

Algorithm 12 Binary Increment

```
$i \leftarrow 0$
while $B[i] = 1$ do
    $B[i] \leftarrow 0$
    $i \leftarrow i + 1$
$B[i] \leftarrow 1$
```

How long does this take?

- The best case is $\Theta(1)$: if the last bit of n is 0 then all we have to do is flip it. The while loop never executes at all.
- The worst case is when n is of the form $2^k - 1$, that is, all 1 bits, and to increment it we have to flip all the 1's to 0s and then set the next 0 bit to 1. Since n requires $\Theta(\lg n)$ bits to represent it, this means that incrementing n takes $\Theta(\lg n)$ in the worst case.

In practice, we rarely just increment a binary number a single time. More realistically, we will be repeatedly incrementing it (perhaps it is a counter of some sort). So how long does it take to start at 0, and increment n times?

- We can definitely say it is $O(n \lg n)$: we do n increment operations and each increment takes $O(\lg n)$ in the worst case.
- We can also definitely say it is $\Omega(n)$: we do n increment operations and each takes at least $\Theta(1)$.

This is a common situation: we have some operation that we want to do repeatedly, and the operation does not always take the same amount of time. We can easily give an upper bound and a lower bound, but we want to be able to say something more precise about how long the entire sequence of operations will take. For this we turn to *amortized analysis*. The idea of "amortization" is to spread out big one-time costs more evenly (*e.g.* repaying a car loan or a mortgage in monthly installments).

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
bit flips		1	2	1	3	1	2	1	4	1	2	1	3	1	2	1	5
cumulative cost	0	1	3	4	7	8	10	11	15	16	18	19	22	23	25	26	31

Table 1: Cost of repeated increments

The first step is almost always to just try small examples, play around, make a table, and notice patterns to come up with a guess. Let's try it for incrementing a binary counter. We'll make a table with the counter value n and the cumulative number of bit flips on the right-hand side. Remember we don't need a *formula* for the number of bit flips[2]; we are just trying to figure out how fast it grows in relation to n.

We notice some patterns: whenever n is a power of two, the total number of bit flips seems to be one less than the next power of two. In fact, in general it seems that the cumulative number of bit flips is never more than $2n$. Based on this evidence, we conjecture that a sequence of n increment operations starting from 0 actually takes $\Theta(n)$, the lower bound we derived from the best case, rather than the worst-case upper bound of $\Theta(n \lg n)$. Intuitively, it seems like the "expensive" increment operations happen infrequently enough that they don't add too much to the total—we can "average out" their cost over the whole sequence. In this case we say that a single increment operation takes $\Theta(1)$ *amortized time*: each increment takes $\Theta(1)$ "on average", even though an individual increment operation could take longer.

So how can we prove this?

Accounting method

Imagine that each constant-time operation "costs" \$1. The idea is to *overcharge* for some operations and "save up" the extra money, so that we have enough saved up to pay for expensive operations. In general, if we charge $c \cdot f(n)$ for some operations and always have enough left over to pay for the other operations, then we can say that each operation takes $O(f(n))$ *amortized* time.

In this example, we imagine that flipping a bit costs \$1. Let's charge \$2 every time we flip a bit from 0 to 1. \$1 will pay for the flip itself, and the other \$1 we imagine being saved next to the 1 bit. Later, when we need to flip the bit back to 0, we will have \$1 sitting there which we can use to pay for the flip.

$$000 \xrightarrow{\$2} 00\underset{\$}{1} \xrightarrow{\$2} 0\underset{\$}{1}0 \xrightarrow{\$2} 0\underset{\$}{1}\underset{\$}{1} \xrightarrow{\$2} \underset{\$}{1}00$$

Notice how every increment operation does exactly one $0 \to 1$ flip, for which we pay \$2. And it might do a bunch of $1 \to 0$ flips, but we get to do those "for free" using the money we saved up from previous increments. All in all, if we pay \$2 for each increment we always have enough money to pay for all the bit

[2] Though it turns out in this case it is possible to come up with one!

flips (without ever going negative). Therefore, we conclude that the *amortized* cost of a single increment operation is \$2, that is, $\Theta(1)$.

As a fun aside, this analysis actually shows us how to come up with a precise formula for the total number of bit flips needed to increment from 0 to n: after doing n increment operations, we have paid \$2n, but not all of that \$2n has actually been used. There is \$1 sitting next to each 1 bit, and the rest of the \$2n has been used to pay for bit flips. So the total number of bit flips needed to increment from 0 to n is exactly

$$2n - \#n$$

where $\#n$ is the number of 1 bits in the binary representation of n.

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
incr cost	0	1	3	1	7	1	3	1	15	1	3	1	7	1	3	1	31
total cost	0	1	4	5	12	13	16	17	32	33	36	37	44	45	48	49	80

Table 2: Incrementing a counter when flipping bit i costs 2^i

28 (L) Amortized analysis II

Direct counting method

Here's another way we can prove that incrementing a binary counter takes $\Theta(1)$ amortized time. AKA "just do some math". Let's add up the total number of bit flips and see what we get. Notice that when doing a sequence of increments, $B[0]$ flips every single time (either 1 to 0 or 0 to 1). Then $B[1]$ flips every other time, $B[2]$ flips every fourth time, ... in general $B[i]$ flips every 2^ith time. So when incrementing from 0 to n, bit $B[i]$ flips a total of $\lfloor n/2^i \rfloor$ times. Thus, the total number of bit flips is

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < \sum_{i=0}^{\infty} \frac{n}{2^i} = n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n.$$

This is a sort of "brute force" method of proof which is straightforward when you can get it to work. But sometimes it's easier to use the accounting method.

Another binary counter example

Take the same binary counter incrementing example, but now suppose it costs 2^i to flip bit i. The worst case for a single increment operation is now $\Theta(n)$, since if we have to flip all the bits it will cost $2^0 + 2^1 + 2^2 + \dots$ which is approximately equal to n. But once again, the total time to do n successive increment operations is actually less than $\Theta(n^2)$. Let's again make a table (Table 2) and look for patterns.

Looking at just the powers of 2 might reveal a useful pattern: $2 \to 4$, $4 \to 12$, $8 \to 32$, $16 \to 80$. It looks like each power of 2 is being multiplied by successive integers (2×2, 4×3, 8×4, $16 \times 5 \dots$ in particular 2^k is being multiplied by $k+1$. So we conjecture that the total cost to increment from 0 to n is no greater than $n \times (\lg n + 1)$, which would mean that the amortized cost of each increment operation is $\Theta(\lg n)$.

We could also use the accounting method: every time we do an increment, just put \$1 on *every* bit, whether we flip it or not. (We have to decide up front how many bits we are going to use.) Since bit i is flipped every 2^i increment operations, by the time we flip it we will have accumulated exactly enough money to pay for its cost of 2^i. Therefore, the amortized cost of a single increment is the amount we actually pay: we pay \$1 for each bit and there are $\Theta(\lg n)$ bits.

[We could also use a direct counting method. Note that bit i is flipped only once every 2^i increments, and flipping it costs 2^i. So each bit contributes a total cost that is at most the total number of increment operations. There are $\lg n + 1$ bits, and each contributes a cost of at most n, for a total of at most $n(\lg n + 1)$.]

Extensible arrays

Another example that should be familiar from Data Structures: extensible arrays (*e.g.* `ArrayList` in Java). As our cost model, suppose accessing or modifying an array entry costs 1, and allocating a new array and copying the contents of an old array into it costs n (the length of the old array). Every time we do an `append` operation, it might cost $\Theta(1)$ (if there is still enough space in the underlying array) or it might cost $\Theta(n)$ if we have to allocate a bigger array, copy all the elements over from the old array, and then insert the new element. So we can definitely say that a sequence of n `append` operations is $\Omega(n)$ and $O(n^2)$.

The question is, what strategy should we use for allocating a new array when we run out of space? This turns out to have a big impact on the amortized time of `append`.

- Strategy 1: increase the size by some constant c. That is, when our array of size n becomes full, allocate a new array of size $n + c$ and copy the contents of the old array into it.
- Strategy 2: double the size. That is, when our array of size n becomes full, allocate a new array of size $2n$.

Let's consider the amortized time for a single `append` operation using these strategies.

- Strategy 1: suppose we start with an array of size c. (The starting size of the array does not really change the analysis at all.) The first c `append` operations will cost 1 each, and then the next will cost $c + 1$ (c to copy the now-full array of size c, and 1 more to insert the new element). Then there will be another $c - 1$ operations that cost 1 each until the new array is full. The next will cost $2c + 1$ (copy the full array of size $2c$ plus 1 to insert). And so on.

  In general, if we do n successive `append` operations, we will end up paying n for the actual inserts, plus

  $$c + 2c + 3c + 4c + \cdots + \lfloor n/c \rfloor c$$

  for array allocations. This is

  $$c(1 + 2 + 3 + \cdots + \lfloor n/c \rfloor) = c \cdot \Theta((n/c)^2) = \Theta(n^2).$$

  So the total cost for n successive `append` operations is $\Theta(n) + \Theta(n^2) = \Theta(n^2)$, and a single `append` has an amortized cost of $\Theta(n)$. In this case, the cost of the expensive operations ends up dominating, even though most of the calls to `append` are just $\Theta(1)$.

- Strategy 2: suppose we start with an array of size 1. If we do a sequence of n `append` operations, of course we will still pay n for the actual insertions; the question is how much we pay for array allocations. After the initial array becomes full (immediately), we pay 1 to allocate a new array of size 2 and copy the element over. When that becomes full, we pay 2 to allocate an array of size 4 and copy over the old elements. And so on. In total, we will pay

  $$1 + 2 + 4 + 8 + \cdots + 2^k,$$

  where 2^k is the biggest power of 2 which is less than n. But this sum is $2^{k+1} - 1$ which is approximately $2n$, that is, $\Theta(n)$. So the total cost of n calls to `append` is $\Theta(n) + \Theta(n) = \Theta(n)$, and the amortized cost of a single `append` is only $\Theta(1)$.

  We can also prove this using the accounting method. Charge \$3 for each `append`. \$1 goes to paying for the actual array insertion; the other \$2 is saved along with the inserted element. Each time the array becomes full, it will look something like this:

  $$\begin{array}{cccccccc} 1 & 3 & 19 & -5 & 6 & 8 & 21 & 0 \\ & & & & \$\$ & \$\$ & \$\$ & \$\$ \end{array}$$

  Since the array capacity was doubled the last time it was resized, for each element already in the array there will be a newly inserted element with \$2 stored next to it. Therefore the total amount of money available is equal to the number of elements in the array, and we can pay to allocate a new array and copy all the elements over.

  Thus, the amortized cost of a single `append` operation is \$3, that is, $\Theta(1)$.

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
incr cost	1	1	1	4	1	1	1	1	9	1	1	1	1	1	1	16
total cost	1	2	3	7	8	9	10	11	20	21	22	23	24	25	26	42

Table 3: Table of costs with $f(k) = k$ when k is square, 1 otherwise

29 (P*) More amortized analysis examples

Let's consider another example. Suppose we are doing some sequence of operations where the kth operation takes $f(k) = k$ time when k is a perfect square, and $f(k) = 1$ otherwise. We can make a table and look for patterns (Table 3). No obvious pattern jumps out at us. So let's try a more direct approach. Suppose we add up all the costs from 1 up to some $n = m^2$:

$$\text{total} = 1 + 1 + 1 + 4 + 1 + 1 + 1 + 1 + 9 + \cdots + m^2$$

We can separate out all the 1's and the squares:

$$(1 + \cdots + 1) + (1 + 4 + 9 + \cdots + m^2)$$

We know there are exactly $m^2 - m$ copies of 1 (though it actually doesn't matter all that much: we want an upper bound so it would be enough to say the sum of all the 1's is *less than* m^2). So the sum is

$$(m^2 - m) + \sum_{j=1}^{m} j^2.$$

We can look up a formula for the sum of squares, and find this is equal to

$$m^2 - m + \frac{m(m+1)(2m+1)}{6}.$$

We could do a bunch of algebra at this point, but we don't need to: all we care about is that this is clearly $\Theta(m^3)$. Note, however, that $n = m^2$, so the sum is $\Theta(n^{3/2})$. Hence the amortized time for a single operation is $\Theta(\sqrt{n})$.

We said earlier that no pattern jumped out at us from the table—but really that's just because we didn't try hard enough. We are computer scientists, we don't have to just stare at things with our eyeballs! Make a spreadsheet, or write a program to generate a bunch of data points, and fit a curve to it. This gives us a good guess to start from.

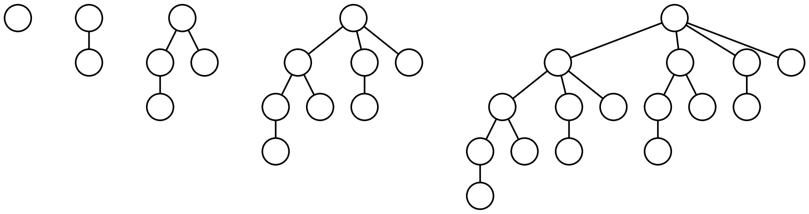
30 (L) Binomial Heaps

Recall the *heap* data structure—a complete binary tree where the value at each node is less than the values at both its children. This lets us implement priority queues with $\Theta(\lg n)$-time insert, delete minimum, and change-key operations.

Consider the *merge* operation: given two priority queues, merge them into a single combined priority queue. This is a natural and useful operation in many applications. Unfortunately, heaps do not support this operation at all. The best one can do is to just list all the elements in the two heaps and then build a new heap out of the elements; this takes linear time. Can we do better?

Definition 30.1. *Binomial trees* are defined as follows:

- A binomial tree of order 0 is a single node.
- A binomial tree of order n consists of a root node with n subtrees, which are binomial trees of order $n-1$, $n-2$, $\ldots$, 0.



Lemma 30.2. *A binomial tree of order n has 2^n nodes.*

Proof. By induction on n. An order-0 binomial tree has $2^0 = 1$ node. If we assume this is true for binomial trees of order $< k$, then a binomial tree of order k has

$$1 + 2^{k-1} + 2^{k-2} + \cdots + 2^0 = 2^k$$

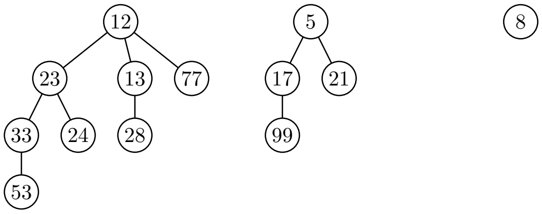
nodes. SDG

Remark. Binomial trees are called *binomial* since a binomial tree of order n has $\binom{n}{k}$ nodes at depth k. This can also be proved by induction, using some facts about binomial coefficients (just think about adding a row of Pascal's triangle shifted against itself to get the next row).

Lemma 30.3. *We can make a binomial tree of order n out of two binomial trees of order $n-1$, by attaching one of the trees as the leftmost child of the other tree's root node.*

Definition 30.4. A *binomial heap* is a list of binomial trees such that:

- Each binomial tree satisfies the heap property (each node is smaller than all its children)
- There is at most one binomial tree of any given order.



We usually keep the list of binomial trees sorted from smallest to biggest order, stored using a linked list. (I will draw them sorted from right to left.) Notice that a binomial heap acts a lot like a binary number: since binomial trees have sizes that are powers of two, and a binomial heap has at most one tree of any given order, it is not hard to see that a binomial heap with n total elements has a binomial tree of order k if and only if the 2^k bit of the binary representation of n is a 1. For example, the heap illustrated above has 13 nodes, $13_2 = 1101$, and indeed the heap has one tree of order 1, one of order 4, and one of order 8. This also means a binomial heap of size n contains $O(\lg n)$ trees, from order 0 up through order $\lfloor \lg n \rfloor$.

To merge two binomial trees of the same order, just see which one has the bigger root, and make it the leftmost child of the smaller root. This takes $\Theta(1)$ time.

To merge two binomial heaps A and B, we can do what amounts to addition of binary numbers. Iterate through them in parallel. For each order k we have possibly a binomial tree of order k from A, possibly a tree of order k from B, and we also may possibly have a *carry* tree of order k, just like we can carry a 1 bit when adding binary numbers. (When we start out at $k = 0$ we have no carry tree.)

- If there are no trees of order k, the output has no trees of order k. ($0+0 = 0$.)
- If there is only one tree of order k, just copy it to the output. ($0+1 = 1$.)
- If there are two trees of order k, merge them into a single tree of order $k+1$ and make it the carry tree for the next iteration; the output has no trees of order k. ($1+1 = 10$, so output 0 and carry the 1.)
- If there are three trees of order k, copy one of them to the output, and merge the other two into an order-($k+1$) carry tree. ($1+1+1 = 11$: output 1 and carry 1.)

Each iteration of the above algorithm takes constant time: we may have to merge two trees but that takes constant time. So merging two binomial heaps of size n can be done in $\Theta(\lg n)$ time, since we do $\Theta(1)$ work for each order from 0 up to $\lfloor \lg n \rfloor$.

Now, let's see how we can implement the other priority queue methods as well:

- Insert: to insert a new element x into a binomial heap H, make x into a size-1 binomial heap (containing a single order-0 binomial tree) and then merge it with H. This is just like incrementing a binary counter! So although it is $O(\lg n)$ in the worst case, it takes $\Theta(1)$ amortized time. (This is even better than a standard binary-tree based heap, which always takes $\Theta(\lg n)$ to insert!)
- Delete-min: iterate through the roots of all the trees to find the smallest in $\Theta(\lg n)$ time (the smallest element in the entire heap is guaranteed to be one of the roots). Removing it leaves its children, which are binomial trees of order $k-1$, $k-2$, …, 0: a binomial heap! Just merge this remaining binomial heap with the rest of the heap in $\Theta(\lg n)$ time. So overall delete-min takes $\Theta(\lg n)$. (Note we can also keep track of the minimum root so that *finding* the minimum can be $\Theta(1)$, even though removing it still takes $\Theta(\lg n)$.)
- Change-key: just bubble the key up or down within its binomial tree until it is at the right spot. This takes $O(\lg n)$.

31 (L*) Potential method and splay trees

There is one more method we can use to prove amortized time bounds, called the *potential method*, which comes from an analogy with physics. The idea is to think of a data structure as having some “potential energy”. Operations on the data structure will sometimes increase the potential energy (storing up some extra energy that can be used later) and sometimes decrease it (releasing some of the stored energy). The amortized cost of an operation is the actual cost of the operation *plus* the change in energy. The change in energy can be positive (in which case the amortized cost is more than the actual cost, and some energy is stored for later) or negative (in which case the amortized cost is less than the actual cost, using up some of the stored energy to pay for the difference).

Consider a sequence of operations on a data structure, and let D_i denote the state of the data structure after the ith operation, and Φ_i the potential energy of D_i. Then the amortized cost of the ith operation (which turned D_{i-1} into D_i) is defined as

$$a_i = c_i + \Phi_i - \Phi_{i-1}$$

where c_i is the actual cost of the operation. Therefore the amortized cost of the sequence of operations is

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = \left(\sum_{i=1}^{n} c_i \right) + \Phi_n - \Phi_0.$$

We need the total amortized cost to be an upper bound on the actual cost, and we can see this will happen as long as $\Phi_n - \Phi_0 \geq 0$, which motivates the following definition:

Definition 31.1. Φ is *valid* if $\Phi_i - \Phi_0 \geq 0$ for all i.

If a potential energy function Φ is valid, it means that the total amortized cost will always be at least the total actual costs, that is,

$$\sum_{i=1}^{n} a_i \geq \sum_{i=1}^{n} c_i.$$

So the general technique is to define a potential energy function Φ which starts out as 0 on the initial data structure, and always remains nonnegative. Given such a valid potential function, the amortized cost of an operation (defined as the actual cost plus the change in potential energy) gives a valid bound on the average cost of a single operation.

Note that this can be seen as a generalization of the accounting method: the amount of "extra money" stored in a data structure can be thought of as "potential energy". But the potential method makes it more clear that the energy does not have to be an integer, nor does it have to be "stored" anywhere in particular. In some cases this leads to an easier analysis even though in theory we *could* do the analysis using the accounting method (with weird fractional amounts of money stored in a "bank account").

Splay trees

As a fascinating application of the potential method we will study *splay trees*, a sort of "self-balancing" variant of binary search trees. Recall that binary search trees give us $\Theta(\lg n)$ insert, lookup, and delete for ordered data, *as long as* the trees remain balanced. But if the tree becomes unbalanced these turn into $O(n)$ instead. In some applications it does not matter; if a binary search tree is built randomly it is very likely to be balanced. However, in many applications the data may have characteristics that lead to an unbalanced tree (for example, the data may be already sorted or close to sorted, in which case inserting it sequentially into a naive BST creates an unbalanced tree). There are various ways to solve this problem. What most people think of are sophisticated BST variants, such as red-black trees and AVL trees, which work by storing extra information in the trees and then using this extra information to make sure the trees remain balanced. Practically speaking, these are some of the best BST implementations, but they are complex, and require reimplementing all the BST methods to ensure that the extra information is kept up-to-date and that the tree is rebalanced as necessary.

Splay trees are an intriguing alternative: a splay tree is just a normal binary search tree, with no extra information stored anywhere. The twist is that we implement an extra *splay* operation which works to make the tree a little more balanced, and we use this operation in conjunction with other operations such as lookup and insert, in such a way that the tree tends to stay balanced.

In particular, the Splay operation works by bringing a particular element to the root of the BST (while keeping all the same elements in the tree and preserving the BST properties). It uses a particular algorithm that we will explore shortly. However, given such an operation that brings a chosen node to the root of a BST, let's see how we update the other BST operations to use it:

- INSERT: Whenever we insert a new value into the splay tree (which works just like normal BST insertion), we then splay the newly inserted element to the root.

- LOOKUP: Similarly, lookup works just like in a normal BST; but after doing a lookup we splay the looked up element (or the closest element, in the case that the element being searched for is not in the tree).

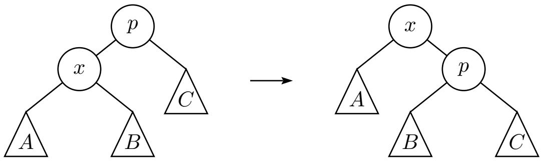
- JOIN: If all the elements in T_1 are less than all the elements in T_2, we can join them into a single BST by first splaying the largest element in T_1 to the root, then adding T_2 as its right child (if the largest element is at the root of a BST, by definition it has no right child, so there is a clear space to add T_2).

- SPLIT: To split a BST into two BSTs which contain all the elements less and greater than a particular target element, respectively, splay the target element to the root and then return its two children.

- DELETE: Splay the item to be deleted to the root, then JOIN the two subtrees. (This is only a minor point, but notice how much nicer this is than the usual BST delete implementation!)

Let's see how SPLAY actually works. We find the element x in the tree and then keep moving it up the tree until it reaches the root, according to the following three rules. In general we will use p to denote the parent of x and g to denote its grandparent.

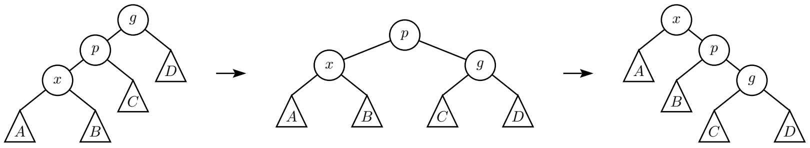
- **ZIG**. If x is the child of the root of the whole tree, then just do one rotation to move it to the root.

  A, B, and C denote arbitrary subtrees. We don't do anything to them at all, so the rotation takes constant time; we just need to update a few references. In the diagram, x is the left child of the root, so we rotate right, but there is also a second entirely symmetric case when x is the right child of the root and we rotate left.

  Note how doing a rotation moves x up while preserving the binary search tree properties: all the elements in the subtree A are still to the left of x; all the ements in B remain in between x and p; and C remains to the right of p.

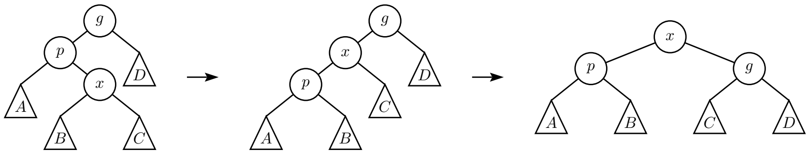
- **ZIG-ZIG**. If x and p are either both left children or both right children, then do two successive rotations: first rotate around $g \to p$ then around $p \to x$.



  Some things to note:

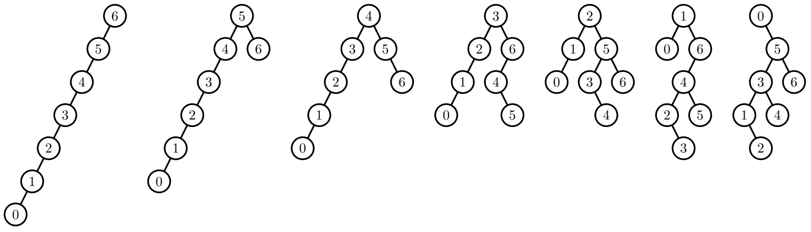
  - This rule applies *anywhere* in the tree, unlike **ZIG** which only applies when x is at a depth of 1. After applying **ZIG-ZIG**, x has moved two levels closer to the root.
  - Like **ZIG**, this takes constant time, since we just do two rotations.
  - Again, in the picture x and p are illustrated as both being left children, but there is an analogous symmetric case when x and p are both *right* children.

- **ZIG-ZAG**. If p is a left child and x a right child, or vice versa, first rotate around $p \to x$ and then around $g \to x$.

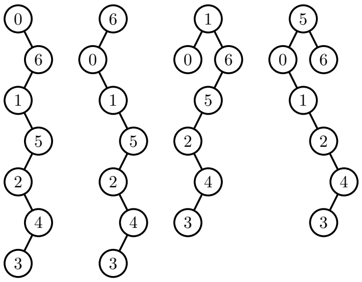


32 (L*) Analysis of splay trees

Let's start with some interactive examples to see how splay trees work in practice, and to gain some intuition for what we are trying to prove. For example, if we start with the tree on the left below, the next tree shows what we get if we splay 5 to the root; the tree after that shows what we would get if instead splay 4 to the root (starting from the tree on the left again); and so on. You can see that as we splay an element from deep in the tree, the other elements tend to "clump up" in twos due to the way we iterate the **ZIG-ZIG** rule.

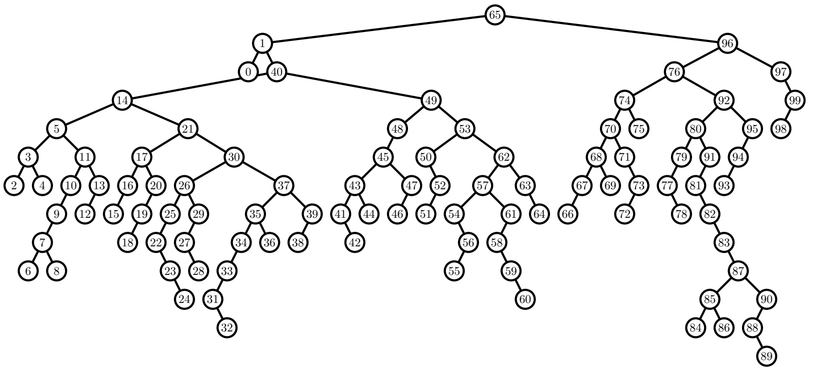


And here is what happens when we iterate the **ZIG-ZAG** rule: starting from the tree on the left, splaying different elements results in the trees on the right:



In this case you can see that zigzags tend to get "unzipped" into two subtrees, one with the small nodes and one with the large nodes.

We think of unbalanced trees as being "tense"/having a high potential energy; the splay operation tends to "relax" them, *i.e.* use some of the stored potential energy to do its work and then leave the tree in a lower-energy state. Doing lots of random splay operations tends to result in a balanced-ish tree. For example, if we start with a (very unbalanced) linear chain of 100 nodes numbered 0 to 99, and do 500 random splay operations, here is (one possible) result:



In fact, it turns out this still works even if the splay operations aren't random: even if we have an evil adversary who is trying to choose splay operations which are as bad as possible for us, the tree will still tend towards balance! Intuitively, splaying an item that is already near the top of the tree doesn't take very long; on the other hand, splaying an item that is deep in the tree takes a while but tends to make the rest of the tree more balanced. So the evil adversary can't win.

Theorem 32.1. *The* Splay *operation takes amortized* $\Theta(\log n)$ *time.*

We note first that this implies all the other methods (in particular Insert, Lookup, Delete) will take amortized $\Theta(\log n)$ time as well. For Insert and

LOOKUP, this is because they have exactly the same cost as splaying: when we lookup or insert, we travel down the tree, paying a cost proportional to the depth of the node we find or insert; then splaying that node to the root has exactly the same cost, since we travel back up exactly the same path, doing one rotation per level of depth. For DELETE, this is simply because it works by doing two SPLAY operations.

To prove this, we will use the potential method.

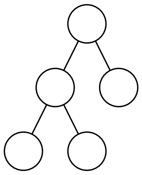
Definition 32.2. If r is some node in a binary search tree, let $S(r)$ be the *size* of the subtree rooted at r, that is, the number of nodes which have r as their ancestor (including r itself).

Definition 32.3. The *rank* of a node r is defined by $R(r) = \log_2(S(r))$.

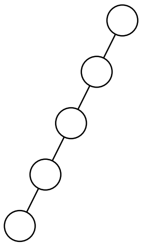
Finally, let Φ be the sum of the ranks of all the nodes in the tree, that is,

$$\Phi = \sum_{r \in T} R(r).$$

(Note this is kind of weird and definitely not an integer! So thinking of this in terms of money would probably not work very well.) For example, consider this tree:



Its three leaves all have size 1 and hence rank 0 (since $\log_2 1 = 0$); the internal node has size 3, and the root has size 5, so Φ for this tree is $\log_2 3 + \log_2 5$. On the other hand, this tree:



has a Φ value of $\log_2 5 + \log_2 4 + \log_2 3 + \log_2 2 + \log_2 1 = 3 + \log_2 5 + \log_2 3$. In general, Φ will be higher for more unbalanced trees.

We note that Φ is a *valid* potential function (according to the definition given previously): Φ for an empty tree is 0, and by definition it is clearly always positive. So the amortized time for an operation, that is, the sum of the actual cost and the change in potential,

$$a = c + \Delta\Phi,$$

will be a valid upper bound on the average cost of a single operation.

Before starting in on the proof proper, we will need a couple observations/lemmas about properties of the $\log_2$ function.

Observation 5. $\log_2$ is an increasing function. So, if $S(x) > S(y)$ then $R(x) > R(y)$. In particular this means that $R(x) > R(y)$ whenever x is an ancestor of y.

Lemma 32.4. *On the interval* $0 < x < 1$*, the function* $\log_2(x) + \log_2(1 - x)$ *attains a maximum value of* -2 *(namely, when* $x = 1/2$*).*

Proof. Straightforward application of standard calculus techniques. SDG

Remark. To get a more intuitive idea of why this is true, recall what the graph of $\log_2(x)$ looks like on the interval $0 < x < 1$: it lies below the x-axis, passing through the points $(1/2, -1)$ and $(1, 0)$, with a vertical asymptote as it diverges to $-\infty$ as $x \to 0$. We are adding up the $\log_2$ of two values which are symmetric about the point $x = 1/2$. If we pick $x = 1 - x = 1/2$ then the sum is -2. As we move the two points farther away from each other, the one moving left moves much faster in the negative y direction than the one moving to the right moves in the positive y direction, so the sum gets more negative.

We want to talk about the rank of nodes before and after applying the splay rules. In general, for a node x we will use $R(x)$ to refer to the rank of x *before* applying a rule, and $R'(x)$ to refer to its new rank *after* applying the rule.

Important note: to understand the following proofs you really have to be looking at the pictures for the different splay rules!

Lemma 32.5. *The amortized cost of a* ***ZIG*** *operation is* $\leq 3(R'(x) - R(x)) + 1$.

Proof. The amortized cost is defined as the actual cost plus the change in potential. In this case the actual cost of **ZIG** is 1 rotation. As for the change in potential, the ranks of x and p may change, but note that the ranks of all nodes in the subtrees A, B, and C do not change. Note also that the old rank of p is the same as the new rank of x (since p used to be the root of the whole tree, but afterwards x is, so the new x has the same number of nodes under it as the old p did). So the amortized cost is

$$
\begin{array}{ll}
\quad 1 + R'(p) - R(p) + R'(x) - R(x) & \\
= & \{ \quad R'(x) = R(p) \quad \} \\
\quad 1 + R'(p) - R(x) & \\
< & \{ \quad R'(p) < R'(x), \text{ since } p \text{ is now a child of } x \quad \} \\
\quad 1 + R'(x) - R(x) & \\
< & \{ \quad R'(x) > R(x), \text{ hence } R'(x) - R(x) \text{ is positive} \quad \} \\
\quad 1 + 3(R'(x) - R(x)) &
\end{array}
$$

SDG

Lemma 32.6. *In the context of the* ***ZIG-ZIG*** *operation,* $2 \le 2R'(x) - R(x) - R'(g)$.

Proof.

$$\begin{array}{ll}
 & R(x) + R'(g) - 2R'(x) \\
= & \qquad\qquad\qquad\qquad \{ \quad \text{definition of } R \quad \} \\
 & \log_2(S(x)) + \log_2(S'(g)) - 2\log_2(S'(x)) \\
= & \qquad\qquad\qquad\qquad \{ \quad \text{properties of log} \quad \} \\
 & \log_2(S(x)/S'(x)) + \log_2(S'(g)/S'(x)) \\
\le & \qquad\qquad\qquad\qquad \{ \quad \text{see below} \quad \} \\
 & -2
\end{array}$$

For the final step, note that $S(x) + S'(g) \le S'(x)$ (since the original tree rooted at x and the new tree rooted at g are completely disjoint, and all the elements of both have x as an ancestor in the new tree), so

$$\frac{S(x)}{S'(x)} + \frac{S'(g)}{S'(x)} = \frac{S(x) + S'(g)}{S'(x)} \le 1.$$

Therefore, the bound of -2 follows from Lemma 32.4.

The lemma as stated now follows by negating both sides of the inequality (and flipping the inequality appropriately). SDG

Lemma 32.7. *The amortized cost of a* ***ZIG-ZIG*** *operation is* $\le 3(R'(x) - R(x))$.

Proof. The actual cost of **ZIG-ZIG** is 2 rotations; the only nodes whose ranks change are x, p, and g. So the amortized cost is

$$\begin{array}{ll}
 & 2 + R'(g) - R(g) + R'(p) - R(p) + R'(x) - R(x) \\
= & \qquad\qquad\qquad\qquad \{ \quad R(g) = R'(x) \quad \} \\
 & 2 + R'(g) + R'(p) - R(p) - R(x) \\
< & \qquad\qquad\qquad\qquad \{ \quad R(x) < R(p), \text{ so } -R(p) < -R(x) \quad \} \\
 & 2 + R'(g) + R'(p) - 2R(x) \\
< & \qquad\qquad\qquad\qquad \{ \quad R'(p) < R'(x) \quad \} \\
 & 2 + R'(g) + R'(x) - 2R(x) \\
\le & \qquad\qquad\qquad\qquad \{ \quad \text{Lemma 32.6} \quad \} \\
 & (2R'(x) - R(x) - R'(g)) + R'(g) + R'(x) - 2R(x) \\
= & \qquad\qquad\qquad\qquad \{ \quad \text{algebra} \quad \} \\
 & 3(R'(x) - R(x)).
\end{array}$$

SDG

Lemma 32.8. *The amortized cost of a* ***ZIG-ZAG*** *operation is* $\le 3(R'(x) - R(x))$.

Proof. Omitted; very similar to the proof for **ZIG-ZIG**. SDG

Now we have finally built the tools we need to prove the original theorem: that Splay takes $\Theta(\log n)$ amortized time.

Proof. The amortized cost of a Splay operation is the sum of the amortized costs of each of the rule applications as x makes its way up the tree; we have shown that each of these costs at most $3(R'(x) - R(x))$, except possibly one final **ZIG** application which takes at most $1 + 3(R'(x) - R(x))$. Note that in each case, the $R'(x)$ from one step becomes the $R(x)$ for the next step, and so the sum telescopes, with all of the intermediate R values cancelling, leaving at most

$$3(R(t) - R(x_0)) + 1,$$

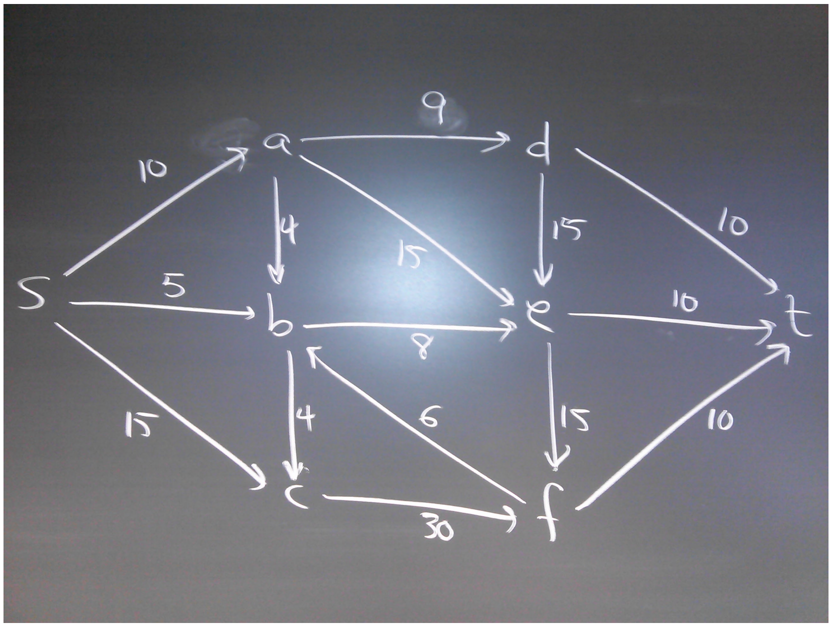
where t denotes the root of the entire tree (where x ultimately ends up) and x_0 denotes the starting location for x. The size of the tree t is of course n, so $R(t) = \log_2(n)$. Also, in the worst case, x had to start all the way down at a leaf, which has $R(x_0) = \log_2(1) = 0$. Therefore the amortized cost is bounded by $3\log_2 n + 1$, which is $O(\log n)$. Of course, it is clear that the cost has to be at least $\Omega(\log n)$ as well, since the height of a tree of size n is at least $\log_2 n$. Therefore Splay takes amortized $\Theta(\log n)$ time. SDG

33 (P/L) Introduction to network flow

This week we will look at **flow networks** which effectively model a diverse set of problems in project selection, airline scheduling, network packet routing, congestion control, baseball elimination, supply chains, image segmentation…

Definition 33.1. A **network** is a

- directed graph $G = (V, E)$,
- a source vertex $s \in V$ (with only outgoing edges),
- a sink vertex $t \in V$ (with only incoming edges), and
- a *capacity function* $c : E \to \mathbb{R}^+$ mapping each edge e to a non-negative capacity $c(e)$.



Definition 33.2. A **flow** on G is a function $f : E \to \mathbb{R}^+$ mapping edges to real numbers such that

1. $0 \le f(e) \le c(e)$ for each edge $e \in E$ (that is, the flow along an edge is limited by the edge's capacity).
2. What flows in must flow out: for each vertex other than s or t,

$$\sum_{e \text{ entering } v} f(e) = \sum_{e \text{ leaving } v} f(e).$$

We will abbreviate the above as

$$f^{\text{in}}(e) = f^{\text{out}}(e).$$

Definition 33.3. The **value** of a flow f is defined as the amount of flow leaving the source:

$$v(f) = f^{\text{out}}(s).$$

Note that because the total flow must be preserved at each vertex, it should be intuitively clear that

$$f^{\text{out}}(s) = f^{\text{in}}(t)$$

so we could equally well define the value of a flow $v(f)$ as the amount of flow arriving at the source. In fact, proving this formally is a good exercise.

Lemma 33.4. $f^{\text{out}}(s) = f^{\text{in}}(t)$*, that is, "what leaves* s *must eventually arrive at* t*".*

Proof. Note first that

$$\sum_{v \in V} f^{\text{out}}(v) = \sum_{e \in E} f(e) = \sum_{v \in V} f^{\text{in}}(v).$$

The middle sum is just the sum of the flows along every edge. But since every edge has exactly one start vertex, the flow along any given edge gets included exactly once in the left-hand sum (as part of $f^{\text{out}}(v)$ for its start vertex). Likewise, each edge is included exactly once in the right-hand sum, with its end vertex.

Since the left- and right-hand sums are equal, we have

$$0 = \left(\sum_{v \in V} f^{\text{out}}(v) \right) - \left(\sum_{v \in V} f^{\text{in}}(v) \right) = \sum_{v \in V} (f^{\text{out}}(v) - f^{\text{in}}(v)).$$

But by definition of a flow (property 2), $f^{\text{out}}(v) - f^{\text{in}}(v) = 0$ for every vertex v other than s or t. Note also that $f^{\text{in}}(s) = 0$ and $f^{\text{out}}(t) = 0$. Hence everything in this sum cancels except

$$f^{\text{out}}(s) - f^{\text{in}}(t).$$

Thus we have shown $f^{\text{out}}(s) - f^{\text{in}}(t) = 0$, that is, $f^{\text{out}}(s) = f^{\text{in}}(t)$. $\boxed{\text{SDG}}$

Max Flow Problem

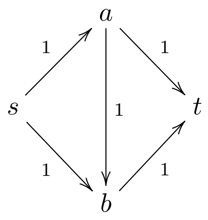
Definition 33.5. The **max flow problem** asks, given a network G, what is the flow with maximum value?

How should we design an algorithm for the maximum flow problem? Well, let's try a greedy strategy, which looks for an *unsaturated* path from s to t (that is, a path for which the flow along every edge is *less* than the edge's capacity), increases (*augments*) the flow along that path as much as possible, and then repeats until no unsaturated path is left.

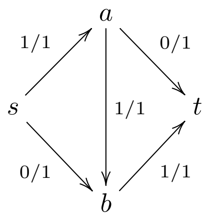
Algorithm 13 GreedyFlow

```
Initialize $f(e) \leftarrow 0$ for all $e \in E$
repeat
    Find an unsaturated path $P$ from $s$ to $t$
    $a \leftarrow$ minimum excess capacity $c(e) - f(e)$ among all edges $e \in P$
    $f(e) \leftarrow f(e) + a$ for each edge $e \in P$
until 1no more unsaturated $s \to t$ paths
```

Unfortunately this does not work. We may get stuck in a "local optimum" where there are no more unsaturated paths, but we have not found the globally maximum flow. Here is an example:



Clearly the max flow in this network is 2 (send one unit of flow along the top and another unit along the bottom). However, if the algorithm happens to pick the path $s \to a \to b \to t$ the first time through the loop, we end up with this:



and there are no longer any unsaturated paths from $s \to t$. The problem is that we shouldn't have picked the middle edge, but there's no way to know that ahead of time, and we have no way to "undo" our choice once we have made it.

However, all is not lost! We can actually keep the basic outline of the greedy algorithm. We really just need to allow ourselves to *undo* or "push back" flow if we find somewhere better for it to go.

Residual Networks

For a given network G and flow f, we define the *residual network* G_f. The residual network has the same vertices as G and essentially has two edges for each edge $e = (u, v)$ in G: one from u to v with capacity equal to the *remaining* capacity $c(e) - f(e)$, and one "backwards" edge from v to u with capacity equal to the flow flow $f(e)$ along e. The backwards edge allows us to "retract" some of the flow along e. The only wrinkle is that we don't include edges with zero capacity, so some edges of G only have one corresponding edge in G_f: those with zero flow (which have only a corresponding forward edge) or those at maximum capacity (which have only a corresponding backward edge).

[Show example.]

Formally,

Definition 33.6. Given a network $G = (V, E)$ and a flow f on G, we define the *residual network* G_f as $G_f = (V, E_f)$, with

$$E_f = \{e \in E \mid c(e) - f(e) > 0\} \cup \{e^R \mid e \in E, f(e) > 0\}$$

(where e^R denotes the reverse of edge e) and we define the capacities of edges in E_f by

- $c_f(e) = c(e) - f(e)$, and
- $c_f(e^R) = f(e)$.

[Show example.]

The Ford-Fulkerson algorithm uses the *residual network* instead of the *original network* to find augmenting paths.

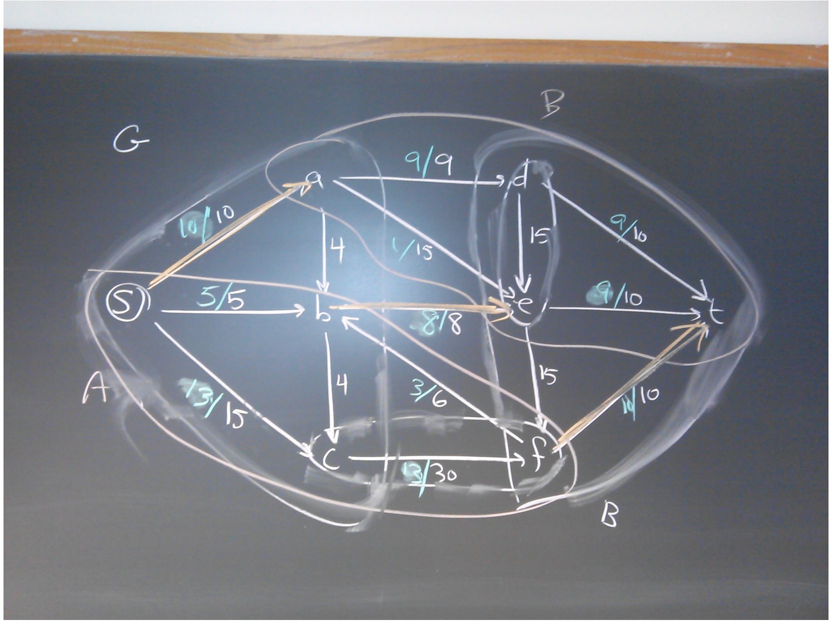
Algorithm 14 FORD-FULKERSON

```
1: $f(e) \leftarrow 0$ for all $e \in E$
2: **while** there exists any path $P$ from $s$ to $t$ in $G_f$ **do**
3:     $\alpha \leftarrow min\{c_f(e) \,|\, e \in P\}$
4:     $f(e) \leftarrow f(e) + \alpha$ for each $e \in P$ such that $e \in E$
5:     $f(e) \leftarrow f(e) - \alpha$ for each $e \in P$ such that $e^R \in E$
```

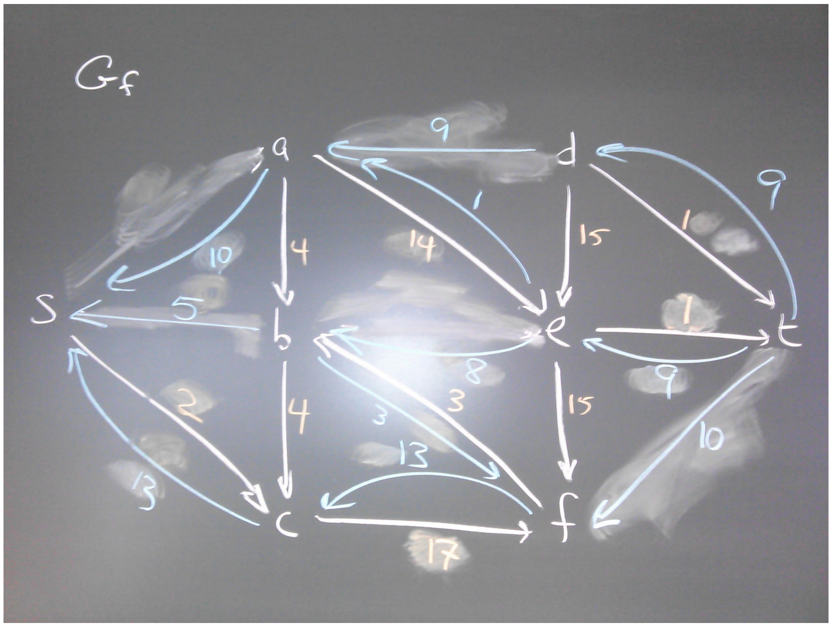
We execute lines 3, 4 and 5 because we have found an *augmenting path* in the residual network.

34 (L) Network flow: max flow/min cut examples and intuition

Start with example of running Ford-Fulkerson on big graph, to refresh our memories. Here's what the graph ends up looking like at the end:



(The min capacity cut (A, B) is also highlighted—see later notes.) And here's what the residual network looks like:



Notice there are no remaining s-t paths in the residual network.

Observation 6. Augmenting along an s-t path in the residual network always preserves the flow properties. (The changes to the flow values adjacent to a given vertex always cancel out.)

Observation 7. Augmenting along an s-t path in the residual network always *increases* the value of the flow.

This is because any s-t path in G_f has to start along some outgoing edge from s, which corresponds to an outgoing edge from s in G. So when we augment along the path we increase the flow along that edge, which by definition increases the value of the flow.

Corollary 34.1. *The Ford-Fulkerson algorithm terminates (the flow can't keep getting bigger forever).*

OK, but we still don't know if it will terminate with the max flow! For all we know it could still get stuck in some local maximum, which can't be augmented even though there is some other flow with a bigger value.

Today we will start in on one of the great duality results in computer science—that the maximum flow of a network is equivalent to the minimum cut of a network. Along the way, we will also show that f is a max flow if and only there are not augmenting paths in G_f, which proves the correctness of Ford-Fulkerson.

Definition 34.2. An *s-t cut* is a partition of V into two sets (A, B) where $s \in A$ and $t \in B$.

Definition 34.3. The *capacity* of an s-t cut is the total capacity of all edges crossing the cut, that is,

$$c(A,B) = \sum_{e \text{ leaving } A} c(e).$$

(Of course if an edge leaves A then it must enter B, that is, cross the cut.)

Show some examples—one "nice" and one crazy. Examine capacity of each.

Definition 34.4. The *min cut* is the s-t cut with minimum capacity.

Definition 34.5. The *net flow* of a cut (A,B) with respect to a flow f is

$$\sum_{e \text{ leaving } A} f(e) - \sum_{e \text{ entering } A} f(e),$$

i.e. the difference between the flow leaving A and the flow entering A. (We could thus call it A's "net exports".)

If we imagine the vertices of A being "clumped together" around the source vertex, then it's clear what should happen: there will be no flow coming in to A, and the flow going out will just be equal to the value of the flow. But A could be crazier—it could consist of a bunch of vertices scattered all through the graph. They don't even have to be connected. So what can we say about the net flow in general? Do some examples!

35 (L) Network flow: max flow/min cut

Lemma 35.1 (Flow Value Lemma)**.** *Let f be any flow and let (A,B) be any s-t cut. Then the net flow of (A,B) with respect to f equals the value of f.*

Proof.

$$\begin{aligned}
v(f) &= f^{\text{out}}(s) && (1)\\
&= \sum_{v\in A}(f^{\text{out}}(v) - f^{\text{in}}(v)) && (2)\\
&= \sum_{e \text{ leaving } A} f(e) - \sum_{e \text{ entering } A} f(e) && (3)\\
&= \text{net flow of } (A,B) && (4)
\end{aligned}$$

Line 2 follows because $f^{\text{out}}(v) = f^{\text{in}}(v)$ for all $v \in A$ besides s. (Remember that $t \notin A$.)

To go from line 2 to line 3, consider all the edges with at least one endpoint in A.

- If an edge has both endpoints in A, its flow contributes twice to the sum in line 2, once positively and once negatively, and the two cancel out. (Intuitively: moving stuff around within A does not affect its net exports.)
- Edges leaving A contribute positively to the sum.
- Edges entering A contribute negatively to the sum.

Hence we are left with line 3. Line 4 follows since line 3 is just the definition of net flow. SDG

Lemma 35.2 (Bottleneck Lemma)**.** *Let f be any flow and (A, B) be any s-t cut. Then $v(f) \le c(A, B)$.*

Proof.

$$\begin{aligned} v(f) &= \sum_{e \text{ leaving } A} f(e) - \sum_{e \text{ entering } A} f(e) \\ &\le \sum_{e \text{ leaving } A} f(e) \\ &\le \sum_{e \text{ leaving } A} c(e) \\ &= c(A, B) \end{aligned}$$

SDG

Remark. From the proof we can see that $v(f) = c(A, B)$ exactly when (1) there is no flow entering A and (2) every edge leaving A is at max capacity.

Corollary 35.3. *Given some flow f and s-t cut (A, B), if $v(f) = c(A, B)$ then f is a max flow and (A, B) is a min cut.*

Proof. No other flow f' could be bigger than f since $v(f') \le c(A, B)$. Likewise, no other cut could have smaller capacity than (A, B), since it has to be at least as big as the value of f. SDG

Theorem 35.4 (Max flow/min cut)**.** *Let f be any flow on a network G. The following are equivalent:*

1. *$v(f) = c(A, B)$ for some cut (A, B).*
2. *f is a max flow.*
3. *There are no s-t paths in the residual network G_f.*

Proof. We will prove that $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 1$.

- $1 \Rightarrow 2$ is just Corollary 35.3.
- $2 \Rightarrow 3$ because of Observation 7: if there were an s-t path in G_f, then we could increase the flow along that path, so f would not be a max flow.

($3 \Rightarrow 1$) is more interesting. Define an s-t cut (A, B) as follows:

- $A =$ all nodes reachable from s in G_f.
- $B =$ everything else.

Notice that

- $s \in A$ (s is trivially reachable from itself).
- $t \in B$ (t can't be in A, since we assumed there are no s-t paths in G_f).

By the Flow Value Lemma,

$$v(f) = \text{net flow of } (A, B) = \sum_{e \text{ leaving } A} f(e) \quad - \sum_{e \text{ entering } A} f(e)$$

Note that each edge (u, v) leaving A must be filled to capacity, since otherwise there would be a forwards edge from u to v with the remaining capacity in G_f, but then by definition we would have $v \in A$. Similarly, each edge entering A must have 0 flow since if it didn't, there would be a backwards edge from v to u in G_f, and then u would be in A.

Hence, we have

$$v(f) = \sum_{e \text{ leaving } A} f(e) \quad - \sum_{e \text{ entering } A} f(e) = \sum_{e \text{ leaving } A} c(e) = c(A, B).$$

SDG

Corollary 35.5. *The Ford-Fulkerson algorithm is correct.*

Proof. We already know Ford-Fulkerson terminates, and it ends when there are no s-t paths in G_f. By the theorem, this means it has found a max flow f. SDG

The proof of the max flow/min cut theorem actually shows something more: the Ford-Fulkerson algorithm can also be used as an algorithm to find a min cut. Just run the algorithm until it stops, and then do a DFS or BFS from s in the residual network G_f to find all the connected vertices, which make up one side of the min cut.

Bonus section: applications of network flow

- Maximum bipartite matching: given an arbitrary bipartite graph with edges between two sets of vertices L and R, we want to find a matching with the most number of edges. Solution:
  - Add a new source vertex s with an edge to each vertex in L

  - Add a new sink vertex t with an edge from each vertex in R
  - Give every edge a capacity of 1.

  The maximum flow now gives us a maximum matching.

  Do an example—see how finding augmenting paths corresponds to finding zig-zagging paths and swapping which edges have flow and which don't, so the number of edges with flow increases by 1 on each iteration.

- Assigning people to teams:

  - Each pereson has teams they are willing to be on
  - Each team has a max size
  - Assign as many people as possible.

Bonus section: max flow algorithms in practice

Ford-Fulkerson is actually a kind of *meta-algorithm*: it specifies the method of repeatedly augmenting along s-t paths in the residual network (*augmenting paths*) but it doesn't actually specify *how* to find augmenting paths. Without specifying how to find augmenting paths at all, we can prove (on the HW) that Ford-Fulkerson will run in $O(EC)$ time, where E is the number of edges and C is the total capacity leaving the source (or the total capacity entering the sink, whichever is smaller). But if we are more careful about how we find augmenting paths we can do better.

- The most straightforward way is to simply use a DFS from s to t in the residual network. This works fine but not particularly well; depending on the graph it can exhibit the worst case $O(EC)$ behavior.

- We can use BFS in the residual network to find the *shortest* augmenting path on each iteration. This is called the *Edmonds-Karp* algorithm, and it can be shown (though the proof is outside the scope of this course) that it runs in $O(VE^2)$—in particular, the running time does *not* depend on the capacities. This algorithm works very well in practice, and often runs much faster than the worst case, behaving more like $\Theta(E^{3/2})$.

- We could use a variant of Dijkstra's algorithm to find the *maximum capacity* augmenting path on each iteration. This seems like it would be a good idea, but in practice it is usually slower than Edmonds-Karp.

- Finally, there is a clever method, *Dinitz' Algorithm*, which instead of finding just a single augmenting path each iteration, sends flow along multiple augmenting paths at once. (Note: it is often known as *Dinic's Algorithm*, but this is due to a misspelling of the name of its inventor, Yefim Dinitz.) It works as follows:

  1. Do a BFS from s in the residual network and label every vertex with its level in the BFS tree.

2. Now do a DFS from s in the residual network, only taking edges which go from level i to $i + 1$; as we recurse through the graph, keep track of the available flow and the minimum residual capacity along the path to each point. Once we reach s we can unroll the recursion and update flow values with the minimum capacity seen along the path, and subtract the augmenting flow from the available flow. Instead of stopping, however, we keep going to try to augment along as many paths as we can. If we encounter a vertex from which it's impossible to reach t, we mark it so we know not to visit it if we ever encounter it again.

It can be shown that this takes $O(V^2E)$, an improvement over Edmonds-Karp for dense graphs; Dinitz also works very well in practice and typically runs much faster than $O(V^2E)$. Also, when using max flow to compute a maximum matching in a bipartite graph, it can be shown that Dinitz runs in $O(E\sqrt{V})$ (this special case of Dinitz for maximum bipartite matching was invented independently and is known as Hopcroft-Karp).

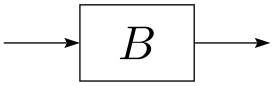
There are many other approaches to finding maximum flows as well (preflow-push, push-relabel, simplex. . .), which can have even better asymptotic running times, work better on certain types of graph, or solve more complex max flow variants (max circulation, min-cost max flow, . . .), but for most practical max flow problems you will encounter, Edmonds-Karp or Dinitz should work just fine!

36 (P*) Intro to reductions: Independent set and vertex cover

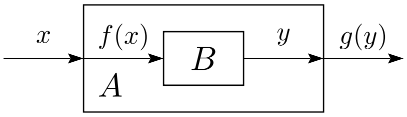
37 (L) Intro to intractability: reductions

This week we will begin studying the limits of computation, from an algorithmic perspective. What problems can computers solve, and what problems can't they solve? Which problems can be solved efficiently and which can't?

The main tool we will use is the process of *reduction*. Suppose we have an algorithm to solve problem B, which we can think of as a black box that takes some inputs describing a problem instance and yields an appropriate output:



We can think of our algorithm as a "subroutine" to turn it into a solution to another problem:



- The function f translates inputs for problem A into inputs for problem B.
- The function g translates outputs for problem B into outputs for prolem A.
- Note we can even allow multiple "calls" to B.

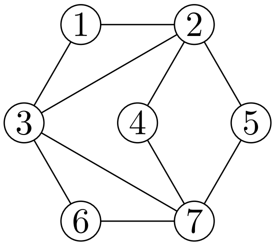
Of course, this is only useful if this really does solve problem A. That is, we need the property that if y is a solution of problem B for input $f(x)$, then $g(y)$ is a solution of problem A for input x. In this case we write $A \leq B$ ("A is reducible to B"). Note we can prove such a reduction without having an actual algorithm to solve problem B: we just show that *if we had* an algorithm to solve B then we could also use it to solve A.

Let's do an example.

Independent set

Definition 37.1. An *independent set* in an undirected graph $G = (V, E)$ is a set of vertices $S \subseteq V$ such that no two nodes in S are joined by an edge.

For example:



What independent sets can you find? Finding *small* independent sets is relatively easy: for example, any set consisting of just a single vertex is an independent set by definition. Any two vertices that are not connected by an edge also form an independent set, like $\{1, 7\}$. Clearly, the interesting thing is to try to find *large* independent sets. In this particular example, we can convince ourselves that the largest independent set is of size 4, namely, $\{1, 6, 4, 5\}$. How hard is it in general to find the largest independent set? At this point, it's not clear! A brute force solution (look at every subset of vertices and check whether each is independent) will obviously take $\Omega(2^n)$ if n is the number of vertices.

Instead of simply asking for the largest independent set, we are going to phrase the problem as a *decision problem*, which asks for a yes/no answer. (We'll explore the reasons for this later.)

> INDEPENDENT-SET: Given a graph G and a natural number k, does G contain an independent set of size *at least* k?

Vertex cover

Now let's consider another problem.

Definition 37.2. A *vertex cover* in a graph $G = (V, E)$ is a set $S \subseteq V$ such that every $e \in E$ has at least one endpoint in S. (In other words, each $e \in E$ is "covered" by some $v \in S$.)

Look at our example graph from before. It's easy to find large vertex covers (for example, the set of all vertices is obviously a valid cover) but hard to find small ones. For our example graph we can convince ourselves that the smallest vertex cover has size 3 (namely, $\{2, 3, 7\}$).

> VERTEX-COVER: Given a graph G and a natural integer k, does G contain a vertex cover of size *at most* k?

As you may have intuited from our example, there is a close relationship between INDEPENDENT-SET and VERTEX-COVER.

Theorem 37.3. *Let $G = (V, E)$ be an undirected graph. Then $S \subseteq V$ is an independent set if and only if $V - S$ is a vertex cover.*

Proof. ($\Longrightarrow$) Let S be an independent set. We must show $V - S$ is a vertex cover. So let $e = (u, v) \in E$. We must show at least one of u or v is in $V - S$. But since S is an independent set, u and v can't both be in S; so at least one of them is not in S, that is, in $V - S$.

($\Longleftarrow$) Let $V - S$ be a vertex cover; we must show S is an independent set. So let $u, v \in S$; we must show there is no edge connecting them. Since $u, v \in S$ then neither one is in $V - S$. Hence there cannot be an edge (u, v), since $V - S$

is a vertex cover, and so every edge must have at least one of its endpoints in $V - S$. SDG

Corollary 37.4. Independent-Set $\leq$ Vertex-Cover *and* Vertex-Cover $\leq$ Independent-Set.

Proof. Suppose we have an algorithm to solve Vertex-Cover. Then to solve Independent-Set(G, k), just solve Vertex-Cover$(G, n - k)$: G has an independent set of size at least k if and only if it has a vertex cover of size at most $n - k$. The other direction is similar. SDG

38 (P/L) Satisfiability

Suppose we are given a set $X = \{x_1, \dots, x_n\}$ of Boolean variables.

Definition 38.1. A *term* is either a variable x_i, or the logical negation of a variable, which we write $\overline{x_i}$ (the negation of x_i is also often written $\neg x_i$).

Definition 38.2. A *clause* is a disjunction of one or more terms, $t_1 \vee \cdots \vee t_l$.

For example, $x_1 \vee \overline{x_3} \vee x_4$ is a clause. (Note there's nothing in the definition precluding repeated variables like $x_1 \vee x_1 \vee x_1 \vee x_3$, but such repetition does not add anything from a logical point of view, so we usually think of the variables as being distinct.)

Definition 38.3. A *truth assignment* is a function $X \to \{\mathsf{T}, \mathsf{F}\}$ assigning a value of true or false to each variable x_i. A truth assignment *satisfies* a clause C if it causes C to evaluate to true (under the usual rules of Boolean logic).

For example, $\{x_1 \mapsto \mathsf{T}, x_2 \mapsto \mathsf{F}, x_3 \mapsto \mathsf{F}, x_4 \mapsto \mathsf{F}\}$ satisfies $x_1 \vee \overline{x_3} \vee x_4$, but $\{x_1 \mapsto \mathsf{F}, x_2 \mapsto \mathsf{F}, x_3 \mapsto \mathsf{T}, x_4 \mapsto \mathsf{F}\}$ does not.

Definition 38.4. A truth assignment $v : X \to \{\mathsf{T}, \mathsf{F}\}$ satisfies a collection of clauses $C_1, \dots, C_k$ if it satisfies all of them (that is, it satisfies $C_1 \wedge C_2 \wedge \cdots \wedge C_k$). We say that v is a *satisfying assignment* for $\{C_1, \dots, C_k\}$. When a collection of clauses has some satisfying assignment, we say it is *satisfiable*.

For example, $(x_1 \vee \overline{x_2}), (\overline{x_1} \vee \overline{x_3}), (x_2 \vee \overline{x_3})$ is satisfiable: just set all x_i to F. (This is not the only possible satisfying assignment.)

On the other hand, as you can verify, $x_1, (x_3 \vee \overline{x_1}), (\overline{x_3} \vee \overline{x_1})$ is not satisfiable: there is no way to assign truth values to the x_i which will simultaneously satisfy all the clauses.

Given these definitions, we can state the following problems:

> Satisfiability (SAT): Given clauses $C_1, \dots, C_k$ over the variables $X = \{x_1, \dots, x_n\}$, is the collection of clauses satisfiable?

Here is another variant of SAT which seems like it might be easier:

> 3-SAT: Given a collection of clauses $C_1, \dots, C_k$ *each containing exactly three variables*, is it satisfiable?

Obviously we have

Theorem 38.5. 3-SAT $\leq$ SAT.

Proof. Since 3-SAT is just a special case of SAT, if we could solve SAT then we could solve 3-SAT as well; we wouldn't even have to do any conversion at all. SDG

Remark. It turns out (but is extremely nonobvious!) that the other direction is true as well: SAT $\leq$ 3-SAT (we will prove this eventually). So in some sense even though the special case of 3-SAT seems like it might be "easier" than the general case of SAT, it is not.

Why 3, you ask? Because 3 is the smallest k for which k-SAT is just as hard as SAT. It turns out that 2-SAT, where all clauses have exactly two variables, is much easier, and can even be solved in linear time. (1-SAT, of course, is trivial (though not as trivial as 0-SAT).)

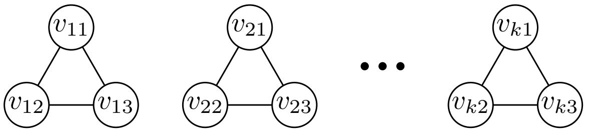
Now let's connect these problems back to some of the problems we have considered previously.

Theorem 38.6. 3-SAT $\leq$ Independent-Set.

Proof. We want to exhibit a polynomial-time reduction from 3-SAT to Independent-Set. That is, given a black box to solve Independent-Set, we want to show how we could use it to solve 3-SAT, using only a polynomial amount of additional work.

Given a set of 3-SAT clauses, we want to find a satisfying assignment, which has to make each clause true. In order to make a clause true it suffices to choose exactly one term from each clause which will be set to T. (Note that "setting $\overline{x_i}$ to True" of course means setting x_i to F.) However, we have to choose consistently: we can never pick both x_i and $\overline{x_i}$.

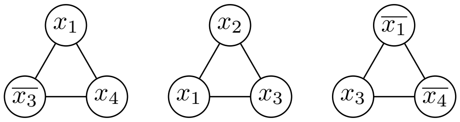
We start by constructing a graph G with $3k$ vertices arranged in k triangles, like so:



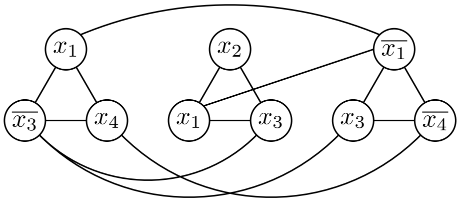
Each triangle corresponds to a clause, and each vertex corresponds to a term; in particular, label v_{ij} by the jth term from clause C_i. So, for example, given the clauses

$$(x_1 \vee \overline{x_3} \vee x_4), (x_2 \vee x_1 \vee x_3), (\overline{x_1} \vee x_3 \vee \overline{x_4}),$$

we would start by constructing the graph



This graph always has independent sets of size k (but no bigger): just pick exactly one node from each triangle. However, we don't want to allow picking just any old combination of nodes. So we add more edges: in particular, add an edge between every pair of nodes labelled by x_i and $\overline{x_i}$ for some i. So our example graph becomes:



An independent set can't contain both endpoints of an edge. These edges therefore encode the constraint that we have to choose variables consistently: we can never choose both x_i and $\overline{x_i}$.

The claim is now that this graph has an independent set of size $\geq k$ iff the original 3-SAT instance is satisfiable—and hence we can solve the original 3-SAT instance by asking our black box for Independent-Set whether this graph has an independent set of size $\geq k$. If the graph has an independent set of size $\geq k$, it must in fact have size exactly k and consist of a choice of one term from each clause. Just making these terms T will result in a satisfying assignment (variables which do not correspond to any chosen term can be set to anything, say, F); this is well-defined because we will never have both x_i and $\overline{x_i}$ selected (because of the extra edges we added). Conversely, if there is a satisfying assignment, just choose one vertex from each triangle whose label is T under the assignment (since it is a satisfying assignment, each clause must have at least one term which is T). Any two vertices connected by an edge are either in the same triangle (and we only pick one vertex from each triangle) or are labelled with inverse terms (and hence can't both be labelled T), so this is an independent set of size k.

SDG

At this point, we have shown

$$\text{3-SAT} \leq \textsc{Independent-Set} \leq \textsc{Vertex-Cover}.$$

39 (L) Decision problems and the $\leq_P$ relation

Definition 39.1. We say an algorithm takes *polynomial time* if it runs in $O(n^c)$ for some constant c, where n is the size of the input.

Remark. Note this doesn't mean the running time has to look like a polynomial! For example, an algorithm that takes time $\Theta(n^2 \log n)$ is polynomial-time, since it runs in, *e.g.*, $O(n^3)$.

Definition 39.2. If $A \leq B$ and the translation functions f and g take polynomial time to compute, and the reduction makes only a polynomial number of subroutine calls to B, then we say that A is *polynomial-time reducible* to B, and write

$$A \leq_P B.$$

Remark. Why focus on *polynomial* time in particular? Several reasons:

- Polynomials are closed under a lot of natural operations:
    - addition (corresponding to running algorithms in sequence),
    - multiplication (corresponding to calling one algorithm as a subroutine of another),
    - and even substitution (corresponding to algorithm composition, that is, using the output of one algorithm as the input to another).

  This means that if we have two polynomial-time algorithms and combine them in some way, the result will also take polynomial time.
- Empirically, polynomial time corresponds well to what is feasible in practice.

Decision problems

Notice how Independent-Set and Vertex-Cover are formulated to give a yes/no answer, *i.e.* a single bit of information. Such problems are called *decision problems*. Note that when reducing decision problems we don't have to worry about translating the output (except perhaps to negate it).

It might seem limiting to restrict ourselves to only decision problems, but actually it is not as restrictive as you might think. As an example, consider another variant of Independent-Set:

Independent-Set -Opt: Given an undirected graph G, find the largest natural number k such that G has an independent set of size k.

Clearly Independent-Set $\leq_P$ Independent-Set-Opt: to decide whether G has an independent set of size at least k, just find the size of the largest independent set and compare it to k. But actually, Independent-Set-Opt $\leq_P$

Independent-Set as well! To find the largest size of an independent set, we can call Independent-Set as a subroutine for each k from 1 up to n until we find the biggest for which we get a "yes" answer. Or we could even be a bit (haha) more sophisticated and do a binary search: Independent-Set returns a single bit as an answer, and using binary search we can reveal a single bit of the desired solution k with each subroutine call to Independent-Set. This results in only a polynomial number ($\Theta(n)$ or $\Theta(\log n)$) of subroutine calls.

This sort of relationship holds often. So, because they are simpler to deal with and don't actually restrict us that much, we will just work in terms of decision problems.

Lemma 39.3. *Suppose A and B are decision problems with $A \leq_P B$. If B is solvable in polynomial time, then so is A.*

Proof. Suppose we have an input x to A of size n. Suppose the translation function f (which converts x into an input suitable for B) takes time $p(n)$ to compute, for some polynomial p. Note that this means the *size* of f's output is also $O(p(n))$, since it takes $p(n)$ time and it has to produce the entire output in that time.

Now suppose B can be solved in time $q(m)$, where q is a polynomial and m is the size of its input. Then the total time to solve problem A on input x, using the reduction to B, is the time to convert the input plus the time to run B on an input of size $O(p(n))$, that is,

$$p(n) + q(p(n)).$$

Since p and q are polynomials, and polynomials are closed under addition and substitution, this is a polynomial. $\boxed{\text{SDG}}$

Remark. It is not too hard to extend this in a couple ways:

- If A and B are not decision problems then we must also take into account the time needed to run the output conversion.
- We can also take into account the case where B is called a polynomial number of times as a subroutine—it essentially involves multiplying by another polynomial, though to do it formally one has to be a bit more careful to state that we have not just a single input conversion function but many, which are all bounded by some polynomial runtime.

Corollary 39.4. *Suppose $A \leq_P B$. If A is not solvable in polynomial time, then neither is B.*

Intuitively, if $A \leq_P B$ then B is "at least as hard as" A (that is, with respect to solvability in polynomial time).

Lemma 39.5. *$\leq_P$ is reflexive and transitive.*

Proof. We can easily reduce a problem to itself using identity conversion functions, which are obviously polynomial-time. Also, if $A \leq B$ and $B \leq C$ it is easy to see that $A \leq C$ by nesting the reductions. A bit more work is required to show that the resulting nested reduction will still be polynomial, but it again boils down to the fact that polynomials are closed under substitution. $\boxed{\text{SDG}}$

40 (L) NP

Often, problems which seem hard to *solve* can nonetheless be easy to *verify* if we are handed a (potential) solution. For example, consider the Independent-Set problem. We don't yet know how hard it is to solve, but if someone hands you a set which they claim is an independent set of size $\le k$, it is easy to verify whether it has the desired size and is, in fact, an independent set—just check each pair of vertices in the set to see whether they are connected by an edge. 3-SAT is similar: if someone hands you a variable assignment, all you have to do is plug it in and evaluate the clauses to see whether they are all satisfied.

Let's be a bit more formal. We will suppose that the input to a problem is encoded as a binary string s. Formally, the "size" of the input will now be the number of bits, $n = |s|$. We will also identify a decision problem X with the set of all inputs s for which the answer should be "yes". The problem then boils down to deciding whether a given bitstring s is contained in X or not. (This is another reason why studying decision problems is particularly nice.)

Definition 40.1. An algorithm A *solves* X if for all binary strings s, $A(s) = \mathsf{T}$ iff $s \in X$.

(If we wanted to be even more formal, we should identify an "algorithm" with some concrete model of computation, typically a Turing machine.)

Definition 40.2. If there is a polynomial $p(n)$ such that $A(s)$ always terminates in at most $p(|s|)$ steps, we say A is a *polynomial-time* algorithm and write $A \in \mathcal{P}$.

Now let's formally define what it means to be able to *verify* the solution to a decision problem. The idea is that we need some sort of extra information to verify a positive answer to a decision problem. For example, given some large graph, if someone were to claim that there was a vertex cover of size ≤ 20, you would challenge them to prove it to you *by showing you the vertex cover.* Given an actual set of vertices, it would be easy for you to check that the set has size ≤ 20 and also that every edge in the graph is covered. Conversely, if there is no vertex cover of that size, then no one is going to be able to fool you by giving you some weird set of vertices. The extra information required to verify a solution is called a *certificate.*

Definition 40.3. An algorithm B is a *polynomial-time certifier* for a decision problem X if:

- B is a polynomial-time algorithm taking some inputs s and t
- There is a polynomial p such that for all bitstrings s, we have $s \in X$ if and only if there is some bitstring t (called a *certificate*) with $|t| \le p(|s|)$ and $B(s, t) = \mathsf{T}$.

In other words, $s \in X$ if and only if there is a (not too big) certificate "proving" it, which can be efficiently checked. That is, if the answer to the decision problem $s \in X$ is YES, then there is a certificate t that proves it: the certifier algorithm B, when run on s and t, will return T. Conversely, if $s \notin X$, there is no t that works: B will never return T for a certificate t unless s really is in X.

For example, again thinking about VERTEX-COVER, s is of course a description of a graph and a number k; a certificate t is simply a list of vertices; the algorithm B checks whether the given set of vertices is $\leq k$ and whether every edge is covered by some vertex in the set, which clearly takes time polynomial in the size of the graph.

Definition 40.4. If X has a polynomial-time certifier, we say $X \in \mathcal{NP}$.

For example, 3-SAT, VERTEX-COVER, and INDEPENDENT-SET $\in \mathcal{NP}$ according to our discussion above.

Remark. $\mathcal{NP}$ stands for "nondeterministic polynomial time". The idea is that if we could make nondeterministic choices (*i.e.* choose multiple things at once and try them in parallel), we could solve any $\mathcal{NP}$ problem in polynomial time: just nondeterministically make all possible choices for the certificate t and check them all at the same time.

We can also observe that

Proposition 40.5. $\mathcal{P} \subseteq \mathcal{NP}$.

Proof. Given a decision problem which can be solved in polynomial time, we can construct a polynomial-time verifier B which simply ignores the certificate and runs the polynomial-time algorithm to solve the problem. SDG

Question. *Does $\mathcal{P} = \mathcal{NP}$?* This question is literally worth \$1 million (it is one of the seven "Millenium Prize Problems" published by the Clay Mathematics Institute). It seems too good to be true, and most believe the answer is no. But this seems extremely difficult to prove.

Definition 40.6. A decision problem X is *$\mathcal{NP}$-hard* if for all $Y \in \mathcal{NP}$, $Y \leq_P X$.

Intuitively, a problem is $\mathcal{NP}$-hard if it is "at least as hard as everything in $\mathcal{NP}$". It seems reasonable that there might be some extremely hard problems which fit the bill—though how on earth would we go about proving it? To show that X is $\mathcal{NP}$-hard, we have to somehow show that *every* $\mathcal{NP}$ problem can be reduced to X, which at first glance seems impossible.

Lemma 40.7. *If X is $\mathcal{NP}$-hard and $X \leq_P Y$ then Y is $\mathcal{NP}$-hard.*

Proof. Transitivity of $\leq_P$. SDG

So if we can find just *one* $\mathcal{NP}$-hard problem, we can potentially use this lemma to find others.

Definition 40.8. A decision problem X is *$\mathcal{NP}$-complete* if both $X \in \mathcal{NP}$ and X is $\mathcal{NP}$-hard.

It is *not* at all clear that any such problems exist at all! We could just as easily imagine a situation where there are just a bunch of problems in $\mathcal{NP}$ which are "maximally hard"—that is, no other problems in $\mathcal{NP}$ are harder—but are still not harder than *everything* in $\mathcal{NP}$. Or we could imagine that the only problems harder than everything in $\mathcal{NP}$ are themselves so hard that they are not in $\mathcal{NP}$.

Proposition 40.9. *If a decision problem X is $\mathcal{NP}$-complete, then X is solvable in polynomial time if and only if $\mathcal{P} = \mathcal{NP}$.*

Proof. ($\Longrightarrow$) If X is solvable in polynomial time, then by a previous lemma, any Y with $Y \leq_P X$ is also solvable in polynomial time. But if X is $\mathcal{NP}$-complete then by definition every problem in $\mathcal{NP}$ is reducible to X and hence solvable in polynomial time.

($\Longleftarrow$) If X is $\mathcal{NP}$-complete then by definition $X \in \mathcal{NP}$, so if $\mathcal{P} = \mathcal{NP}$ then $X \in \mathcal{P}$. $\boxed{\text{SDG}}$

Corollary 40.10. *If you come up with a polynomial-time algorithm to solve an $\mathcal{NP}$-complete problem, you win \$1 million.*

Good luck. Note that this is not just theoretical; it turns out that there are a great many $\mathcal{NP}$-complete problems that people actually care about on a practical level. So lots of smart people have been trying hard for a long time to develop good algorithms to solve them, but no one has ever come up with any polynomial-time algorithms for any of them.

41 (L) The first NP-complete problem

Definition 41.1. A *boolean circuit* is a directed, acyclic graph G such that:

- Each indegree-0 vertex ("input") is labelled with either F, T, or a distinct variable.
- All other vertices have indegree 1 or 2, and are labelled with either $\neg$ (if indegree 1) or $\vee$ or $\wedge$ (if indegree 2)
- Only one vertex has outdegree 0, which we call the "output".

(Note this is not really a "circuit" as we would usually conceive it, since in particular there are no loops!)

Given a truth assignment for the variables labelling the inputs, we think of the circuit as resulting in a T/F value in the obvious way: each edge corresponds to a wire which carries a single T/F value to the input of the next logic gate. Boolean circuits come with a natural decision problem:

> Circuit-SAT: Given a boolean circuit, is there an assignment of the input variables such that the circuit outputs T?

Theorem 41.2 (Cook, Levin (1971))**.** Circuit-SAT *is $\mathcal{NP}$-complete.*

Proof. First, it is easy to see that Circuit-SAT $\in \mathcal{NP}$: a certificate is a truth assignment; to verify it we can just run the circuit and check that it outputs T.

Now we must show Circuit-SAT is $\mathcal{NP}$-hard. Suppose $X \in \mathcal{NP}$, *i.e.* X has a polynomial-time certifier B. We must show $X \leq_P$ Circuit-SAT.

Since B is an algorithm, it can be encoded as a boolean circuit (proof: computers exist, and any boolean function can be encoded using $\wedge$, $\vee$, $\neg$). Well, except for one problem: our boolean circuits have no loops. But we can circumvent that by simply "unrolling" any loops, by making a whole bunch of copies of a circuit that computes a single step. If the algorithm only runs for a polynomial amount of time then we need at most a polynomial number of copies to make this work. So, intuitively, we can build a polynomial-sized boolean circuit that simulates the algorithm B. Given an input s, we fix the inputs corresponding to s, and leave as variables the inputs corresponding to t. Now we ask whether the resulting circuit is satisfiable. If it is, that means there is some t such that $B(s, t) = \mathsf{T}$, that is, $s \in X$. If it is not satisfiable, then there is no such t, so $s \notin X$. Hence $X \leq_P$ Circuit-SAT. $\boxed{\text{SDG}}$

Now that we know there is one $\mathcal{NP}$-hard problem, finding others is a lot simpler!

Theorem 41.3. 3-SAT *is $\mathcal{NP}$-complete.*

Proof. We already know 3-SAT $\in \mathcal{NP}$. We will show it is also $\mathcal{NP}$-hard "by reduction from CIRCUIT-SAT", that is, we will show CIRCUIT-SAT $\leq_P$ 3-SAT. Since we know CIRCUIT-SAT is $NPoly$-hard, by transitivity this will mean that 3-SAT is also $\mathcal{NP}$-hard.

We are given a circuit as input, and have to decide whether it is satisfiable. We will construct a corresponding 3-SAT instance which is satisfiable iff the original circuit is.

First, associate a distinct variable with each edge in the circuit. Now we will define one or more clauses for each node, where each clause has *at most* three terms. To make things a bit more intuitive we will use $a \Rightarrow b$ as shorthand for $\overline{a} \vee b$.

- Given a $\neg$ node with input x_u and output x_v, generate the two clauses $(x_u \Rightarrow \overline{x_v})$ and $(\overline{x_u} \Rightarrow x_v)$.
- Given a $\vee$ node with inputs x_u, x_v and output x_w, generate the three clauses $(x_u \Rightarrow x_w)$, $(x_v \Rightarrow x_w)$, and $((\overline{x_u} \wedge \overline{x_v}) \Rightarrow \overline{x_w})$ (this last expression simplifies to $x_u \vee x_v \vee \overline{x_w}$, so it is a valid clause; remember clauses can contain only $\vee$, not $\wedge$).
- Given a $\wedge$ node with inputs x_u, x_v and output x_w, generate the three clauses $(\overline{x_u} \Rightarrow \overline{x_w})$, $(\overline{x_v} \Rightarrow \overline{x_w})$, and $((x_u \wedge x_v) \Rightarrow x_w)$ (which simplifies to $\overline{x_u} \vee \overline{x_v} \vee x_w$).
- For a constant input node connected to the edge x_u, just generate the clause x_u (if the constant is T) or $\overline{x_u}$ (if the constant is F).
- Finally, generate a clause containing just the variable connected to the output node (since we want the output to be true).

At this point it is clear that the resulting set of clauses will be satisfiable if and only if the original circuit is satisfiable, because the clauses encode exactly what will be computed throughout the entire circuit.

The only remaining detail is that we have to make sure each clause has exactly three terms. To do this, first introduce four new variables, call them $z_1 \dots z_4$. The idea is that we will force z_1 and z_2 to be F (z_3 and z_4 exist just to help us do this), so they can be used to "pad out" any clauses which are too short, without affecting their meaning.

Create eight new clauses of the form $(\overline{z_i} \vee \pm z_3 \vee \pm z_4)$, where we replace z_i by either z_1 or z_2, and replace $\pm z_3$ by either z_3 or $\overline{z_3}$, and so on, and we do this in all possible ways, thus resulting in eight clauses. Notice that these clauses force $z_1 = z_2 = \mathsf{F}$: no matter what values are assigned to z_3 and z_4, there will be one clause of the form $(\overline{z_1} \vee \dots)$ where the $\dots$ terms are false, and hence for the clause to be satisfiable we must have $z_1 = \mathsf{F}$ (and similarly for z_2). Now, for each clause with fewer than three terms, just add $\dots \vee z_1$ or $\dots \vee z_1 \vee z_2$ as appropriate to bring it up to three terms. Since z_1 and z_2 must be F, this has no effect: the new, padded clause will be satisfied iff the original clause is.

Hence we have succeeded in constructing a 3-SAT instance which is satisfiable iff the original boolean circuit is satisfiable. The constructed 3-SAT instance has as many variables as there are edges in the circuit, and no more than $3n + 8$ clauses, where n is the number of vertices in the circuit, so constructing the 3-SAT instance takes time polynomial in the size of the circuit.

Corollary 41.4. 3-SAT*,* SAT*,* Independent-Set*, and* Vertex-Cover *are all* $\mathcal{NP}$*-complete.*

Proof. We have already discussed the fact that they are $\mathcal{NP}$. Since we now know Circuit-SAT $\le_P$ 3-SAT $\le_P$ Independent-Set $\le_P$ Vertex-Cover and 3-SAT $\le_P$ SAT, by transitivity of $\le_P$ they are all $\mathcal{NP}$-hard.

This is amazing. Just by looking at the definitions it wasn't at all clear whether we would be able to find a *single* problem which was $\mathcal{NP}$-complete, and now we have found five. (And you will find yet more on your HW.) In fact, it turns out that we currently know of *hundreds* of "natural" problems which are $\mathcal{NP}$-complete.

So, the "reason" Circuit-SAT is $\mathcal{NP}$-hard is that we can use it to model arbitrary computation, by essentially building a little computer. But think about what this transitive chain of reductions is then showing us: we can take a circuit modelling a computer and reduce it to a 3-SAT instance which models the same computer. But we can then in turn take that 3-SAT instance modelling a computer and reduce it to a graph, such that finding an independent set of a certain size in the graph models the same computation! Computation is sneaky like that; you end up finding it in places you would not expect. Who would have guessed that finding independent sets in a graph turns out to be equivalent to doing arbitrary computation? But in some sense $\mathcal{NP}$-complete problems are like this, and this gives us some intuition as to why they are so hard to analyze: because you can smuggle arbitrary computation into them!

42 (L) Travelling salesman is NP-complete

1. 3-SAT $\leq$ HAMILTONIAN CIRCUIT $\leq$ TRAVELLING SALESMAN.

See K&T for reductions.

43 (L) Super Mario Bros (and Kirby's Dream World) are $\mathcal{NP}$-hard

Show paper by Aloupis et al.

44 (L*) Rabin-Karp pattern matching

Given a *pattern* $P = p_1 \dots p_m$ and a *text* $T = t_1 \dots t_n$ where $p_i, t_i \in \{0, 1\}$, a natural question to ask is, does P occur as a substring within T (and if so, where)? Note that typically n is much larger than m; think of T as a large body of text and P as a word or phrase we are looking for. (If we wanted to do this with actual text instead of bitstrings, we could simply encode the text as bitstrings, or all of the following can be easily generalized to alphabets other than $\{0, 1\}$.)

Let $T_{i,m} = t_i t_{i+1} \dots t_{i+m-1}$ be the substring of T starting at index i with length m (the same length as the pattern P). The question then becomes: does the exists some $1 \le i \le n - m + 1$ such that $T_{i,m} = P$? Note that P must occur contiguously (we are looking for a sub*string*, not a sub*sequence*).

The naïve algorithm would be to just try comparing P to each position in T, but this is $O(nm)$ in the worst case (imagine matching $P = 00001$ against $T = 0000 \dots 0000$). There are algorithms for solving this problem in $O(n + m)$ such as Knuth-Morris-Pratt or Boyer-Moore. These algorithms are deterministic but somewhat complicated. Today we will look at the Rabin-Karp algorithm, which involves randomness and has $O(n + m)$ *expected* time. Even if m is relatively small, this can make a big difference in practice (imagine if Google took 14 times longer to search for an input phrase with 14 letters than it did to search for a single letter).

The idea is to use a *hash function h*. The basic outline of the algorithm is as follows:

Algorithm 15 Naïve Rabin-Karp

```
1: for $i \leftarrow 1 \dots n - m + 1$ do
2:     if $h(T_{i,m}) = h(P)$ then
3:         if $T_{i,m} = P$ then return $i$
   return Not found
```

We iterate through all possible positions, and check whether the hashes of $T_{i,m}$ and P are equal. If the hashes are *not* equal then $T_{i,m}$ and P definitely do not match. But if the hashes *are* equal, we still have to check, since different strings can hash to the same value.

Our hope is that by checking hashes we can avoid actually comparing strings. However, at this point we haven't actually saved any work yet. The outer loop (line 1) of course executes $O(n)$ times, and computing the hash of a length-m bitstring definitely takes at least $\Omega(m)$ time because it at least has to read the whole string! So the whole algorithm is still $O(mn)$. However, we have made some conceptual (if not actual) progress!

The first insight is that if we pick h cleverly, we can avoid doing so much work to compute it: perhaps we can compute $h(T_{i+1,m})$ from $h(T_{i,m})$ in $O(1)$ time. That would bring down the time needed for the check on line 2. The only remaining problem is that we might have too many hash collisions. The check on line 3 is still $O(m)$; if we have to do the expensive equality check too many times, the algorithm could still be $O(mn)$ overall. We'll address this problem later.

To be able to compute h incrementally in this way, we introduce the concept of a *rolling hash function*.

Definition 44.1. Let $I(b_1 \dots b_m) = 2^{m-1}b_1 + 2^{m-2}b_2 + \dots + 2^0 b_m$, that is, the integer value of $b_1 \dots b_m$ considered as a binary number.

Definition 44.2. Let p be a prime number chosen from some range $[1 \dots 2^k]$ (we will pick k later). Then define $h_p(X) = I(X) \bmod p$.

Note that $h_p(X)$ takes $O(m)$ time to compute (where m is the number of bits in X).

Notice that

$$\begin{aligned}
I(T_{i+1,m}) &= I(t_{i+1}t_{i+2}\dots t_{i+m}) \\
&= 2^{m-1}t_{i+1} + 2^{m-2}t_{i+2} + \dots + 2^1 t_{i+m-1} + 2^0 t_{i+m} \\
&= 2^m t_i + 2^{m-1}t_{i+1} + \dots + 2^1 t_{i+m-1} - 2^m t_i + 2^0 t_{i+m} \\
&= 2(2^{m-1}t_i + 2^{m-2}t_{i+1} + \dots + 2^0 t_{i+m-1}) - 2^m t_i + t_{i+m} \\
&= 2I(t_i t_{i+1} \dots t_{i+m-1}) - 2^m t_i + t_{i+m}.
\end{aligned}$$

and hence $h_p(T_{i+1,m}) = (2h_p(T_{i,m}) - 2^m t_i + t_{i+m}) \bmod p$, which can be computed in $O(1)$. So now the initial evaluation of $h_p(T_{1,m})$ takes $O(m)$, and computing subsequent values of h_p takes only $O(1)$.

Now, what about collisions? Note that

$$\begin{aligned}
& h_p(P) = h_p(T_{i,m}) \\
\iff & I(P) \equiv I(T_{i,m}) \pmod p \\
\iff & I(P) - I(T_{i,m}) \equiv 0 \pmod p \\
\iff & p \text{ divides } |I(P) - I(T_{i,m})|.
\end{aligned}$$

Since both P and $T_{i,m}$ have m bits, their difference is at most $|I(P) - I(T_{i,m})| \le 2^m$. So we can quantify the probability of a hash collision by answering two questions:

1. How many prime divisors can a number $\le 2^m$ have?
2. How many primes are there in the range $[1, 2^k]$?

Answering the first question is easy:

Lemma 44.3. *Any number $N \le 2^m$ has at most m distinct prime divisors.*

Proof. All primes are ≥ 2. So if there were more than m distinct prime divisors, their product would be $> 2^m$. SDG

The second question is not so easy to answer, but thankfully it has already been answered for us:

Theorem 44.4 (Prime number theorem)**.** *Let $n \in \mathbb{Z}^+$, and let $\pi(n)$ denote the number of primes $\leq n$.*

$$\pi(n) = \Theta\left(\frac{n}{\ln n}\right).$$

We can now quantify the probability of a hash collision, which is just the probability that the chosen prime p matches one of the prime divisors of $I(P) - I(T_{i,m})$. This probability, in turn, is just the number of distinct prime divisors of $I(P) - I(T_{i,m})$ divided by the total number of primes in $[1, 2^k]$.

$$\begin{aligned}
\Pr[h_p(P) = h_p(T_{i,m,})] &= \frac{\#\text{ of distinct prime divisors of } I(P) - I(T_{i,m})}{\#\text{ of primes } \in [1, 2^k]} \\
&\leq \frac{m}{\pi(2^k)} \\
&= O\left(\frac{m}{2^k / \ln 2^k}\right) \\
&= O\left(\frac{mk}{2^k}\right)
\end{aligned}$$

If we choose $2^k = nm\log(nm)$, then $k = \log 2^k = \log(nm) + \log\log(nm) \leq 2\log(nm) = O(\log(nm))$, in which case

$$O\left(\frac{mk}{2^k}\right) = O\left(\frac{m\log(nm)}{nm\log(nm)}\right) = O(1/n).$$

Therefore the probability of getting a collision at each index i is only $1/n$, which means we expect about 1 collision over the course of the entire algorithm—this is great! The whole algorithm thus runs in $O(m+n)$ ($O(m)$ to compute the initial hash + $O(n)$ loops with $O(1)$ work in each to compute the next hash + $O(m)$ to do the expected 1 equality check).

And finally, how big is p? Actually it's not bad at all: p has $k = O(\log(nm)) = O(\log n)$ bits. For example, a 64-bit prime is enough to efficiently search in a text of 2^{64} bits ≈ 2 exabytes.

45 (L*) Bucket sort

Suppose we are given a list of 10 million records, each representing a person, and we want to sort them by their birthday (*i.e.* all the people born on January 1st come first—regardless of their birth year—then all those born on January 2nd, and so on). What is the best algorithm for doing this?

We could of course use a standard sorting algorithm like merge sort, which would take $\Theta(n \log n)$ time. But in this case we can actually do something better! Make an array with 366 "buckets", one for each day of the year. Now scan through the records and put each record in the bucket corresponding to its birthday. After putting all the records in buckets, simply list all the items in the first bucket, followed by all the items in the second, and so on. Assuming that we can add and remove from the buckets in constant time, and that we can access any given bucket in constant time, this algorithm takes only $\Theta(n)$ time.

So what gives? Haven't we proved that we can't sort any faster than $\Theta(n \log n)$? Well, yes, but that is for *comparison-based* sorting, *i.e.* sorting under the assumption that the only thing we can do with two keys is to compare them. In this case we can do better, since we know in advance how many keys there are and can easily convert them into consecutive integers.

In general, if we have a list of items L with $|L| = n$, and k different keys $0 \dots k-1$, we can use the following algorithm:

Algorithm 16 BUCKET SORT(S)

```
Initialize an array $Q$ of $k$ queues
for $x \in L$ do
    Add $x$ to $Q[x.key]$
for $i \in 0 \dots k-1$ do
    Append the contents of $Q[i]$ to the output list
```

We assume we are using a queue implementation that supports $\Theta(1)$ enqueue and dequeue. This algorithm takes $\Theta(k)$ to create the buckets in the first place, then $\Theta(n)$ to scan through and place everything in buckets, and another $\Theta(n)$ to list the buckets in order, for a total of $\Theta(n+k)$. This works well when

- we know k in advance
- k is relatively small

Of course if $k < n$ this is $\Theta(n)$.

Using queues for buckets ensures that the algorithm is *stable*: we say a sorting algorithm is *stable* if items having equal keys retain their relative order from the input. All the items with a given key i are added to $Q[i]$ in the order they are encountered in the input list L, and then removed from $Q[i]$ in the same order. This stability may or may not be a big deal when using bucket sort by itself—we might not care about the relative ordering of all the people born on February 12—but it will turn out to be crucial in using bucket sort as a building block for more complex algorithms.

46 (L*) Radix sort

Suppose more generally that our keys are strings in Σ^* for some alphabet Σ, that is, keys are sequences of elements from Σ. For example, if $\Sigma = \{0, \dots, 9\}$ then keys would be numbers like 123 or 97268. If the keys have length up to d then there are $|\Sigma|^d|$ possible keys, which is probably too many to do bucket sort. For example, suppose we have ten million records which we want to sort by last name, where each name is a string of letters. Even if we assume that names are at most ten letters long (probably a bad assumption!), that is still $26^{10} \approx 141$ trillion possible names, so bucket sort—which would require creating an array with 141 trillion elements—is completely out of the question.

However, in this situation we can still often do better than a comparison-based sort, using an algorithm called *radix sort*. Radix sort was actually used as long ago as *1887* (yes, 1887, not 1987); look up Herman Hollerith if you want to know more!

The basic idea is to do an independent bucket sort on each key position. As an example, suppose $\Sigma = \{0, \dots, 9\}$, and let's begin with the list

$$126, 328_A, 636, 128, 341, 121, 416, 131, 016, 328_B.$$

The two copies of 328 are marked with A and B so we can see what happens to them.

- First, we bucket sort on the least significant, *i.e.* last, digits. We would put 126 into the 6 bucket, then 328_A into the 8 bucket, and so on, resulting in the buckets

0	1	2	3	4	5	6	7	8	9
	341					126		328_A	
	121					636		128	
	131					416		328_B	
						016			

  So, after listing out all the buckets in order, we get

$$341, 121, 131, 126, 636, 416, 016, 328_A, 128, 328_B.$$

- Next, we bucket sort on the second digit:

0	1	2	3	4	5	6	7	8	9
	416	121	131	341					
	016	126	636						
		328_A							
		128							
		328_B							

  resulting in

$$416, 016, 121, 126, 328_A, 128, 328_B, 131, 636, 341.$$

The bucket sort puts the elements in order by their second digit, but notice that something else has happened: because they were already sorted by their last digit, and bucket sort is stable, elements with the same second digit end up in the correct order with respect to their last digits. So at this point, the list is actually sorted by the *last two* digits of each element $(16, 16, 21, 26, 28, \dots)$.

- One final bucket sort by the first digit now leaves the entire list correctly sorted:

0	1	2	3	4	5	6	7	8	9
016	121		328_A	416		636			
	126		328_B						
	128		341						
	131								

  resulting in

$$016, 121, 126, 128, 131, 328_A, 328_B, 341, 416, 636.$$

  Again, since the elements were already correctly sorted by their last two digits, doing a stable sort on their first digits leaves them in the correct order. Notice also that radix sort itself is stable: for example, 328_A and 328_B ended up in the same relative order they started in.

Theorem 46.1. *Radix sort is a correct sorting algorithm.*

Proof. We claim that after the ith bucket sort pass, the keys are sorted by their length-i suffixes, which we prove by induction on i.

- In the base case, when $i = 0$, the statement is vacuously true: nothing is sorted.

- In the inductive case, suppose the claim holds after the first i bucket sort passes; we will show it continues to hold after the $(i+1)$st. Consider keys X and Y.

  - If $X_{i+1} \neq Y_{i+1}$ then the $i+1$st bucket sort pass will put them in the correct order: they will go in different buckets, and their correct order with respect to their length-$(i+1)$ suffixes is determined by X_{i+1} and Y_{i+1}.
  - If $X_{i+1} = Y_{i+1}$ then by the induction hypothesis they were already in the correct order by their length-i suffixes and hence by their length-$(i+1)$ suffixes as well. The next bucket sort pass will not change their relative order since bucket sort is stable.

□

If we assume all the keys have length at most d, and assume that the size of the alphabet is a constant with respect to the size of the input, radix sort takes $O(dn)$: we do one $\Theta(n)$ bucket sort for each of the d key positions. Notice that if most of the keys have length d then dn is the actual size of the input, so this is really just linear in the size of the input. (There are complications when the keys have very different lengths, a few keys are much longer than all the others, etc.)

This algorithm is called LSD radix sort (LSD stands for Least Significant Digit), since we sort starting with the least significant key position. It works well for sorting numbers, especially when the numbers can be different lengths: we imagine left-padding numbers with zeros, but with LSD radix sort we don't actually need to do any padding, and we don't need to worry in advance about how long the different numbers are.

Alternatively, we can imagine running bucket sorts in order from most to least significant digit (MSD radix sort), using each bucket sort to partition the keys and doing a recursive sort on each bucket. This works well for sorting strings alphabetically.

Algorithm 17 MSD-Radix(A,i)

```
if $|A| = 1$ then return $A$
else
    Sort $A$ into buckets based on the $i$th character
    Recursively call MSD-Radix($B, i + 1$) on each bucket $B$
    Concatenate and return the results.
```

One way to look at this is that it recursively builds an intermediate trie, and then returns an inorder traversal of the trie. The pro of this approach is that it deals efficiently with different length keys. The con is that it uses quite a bit of space and function call overhead.

47 (L*) More radix sort

Radix sorting integers in practice

Consider radix sorting 64-bit integers in practice. Our example from last time involved using the alphabet $\Sigma = \{0, \dots, 9\}$, but converting all the integers to base 10 just to sort them is ridiculous.

Of course the integers are actually stored in base 2, but if we literally used radix sort with $k = 2$, we would do 64 bucket sort passes with only two buckets each time, taking time $O(64n)$. Notice that this is actually slower than $O(n \log n)$ unless $n > 2^{64} \approx 18$ quintillion! We probably will never be sorting a list containing quintillions of integers so this is no good. Sometimes constants really do matter!

One nice solution is to use a bigger value of k. In particular, we can think of the numbers as being stored in base 2^{16} (each 64-bit integer consists of four base-2^{16} "digits"). We make $2^{16} = 65536$ buckets and do four bucket sort passes; on each pass we place the numbers in appropriate buckets by looking at 16 bits at a time. (Note that pulling out a given 16 bits of an integer and using it to index into an array is very fast, using hardware-supported bitwise operations.) Overall this takes $O(2^{16} + 4n)$.

Radix sorting strings in practice

We saw last time that a simple implementation of radix sort takes $O(nd)$ where n is the number of items to sort, and d is the maximum length of the keys (which we assume to be strings over some alphabet Σ). When all the keys have the same length, nd is exactly the size of the input, so this is optimal (assuming it takes $\Theta(d)$ to look at an entire key).

But what if the keys have different lengths? A good example is when we want to sort a collection of strings, which in general probably won't have the same length. One approach is to simply imagine right-padding each string with "null" characters (which sort before any other character). This will give the right behavior; the problem is that it may be too slow. In particular, think of what happens during the algorithm, when we are doing a bucket sort pass on an index that is greater than the length of all but a few strings. We will go through the entire list, placing almost all of the strings into the single "null" bucket, putting only a few long strings into other buckets, and then concatenating them all again. This is mostly wasted work.

In the worst case, suppose we have n strings which are almost all some constant length $d \ll n$, except for one string which has length $\Theta(n)$. In this case a naive implementation of radix sort would take $\Theta(n^2)$, even though the total size of all the strings is only $\Theta(n)$—even worse than a comparison-based sort! Can we do better?

One idea is of course to use a recursive MSD radix sort. But with some cleverness we can do even better, avoiding a lot of the overhead associated with that approach.

The idea is to sort the strings by length, and then deal with them in order from longest to shortest. On each bucket sort pass we can then restrict our attention to only the strings which are long enough; we know all the shorter strings would be put in the "null" bucket so there is no point in doing so explicitly.

But how can we sort the strings by length? With another bucket sort, of course!

Algorithm 18 StringSort1(L)

```
1: $d \leftarrow$ maximum length of any string in $L$
2: $\mathit{ofLen} \leftarrow$ Bucket sort $L$ by length, returning $d$ buckets
3: $T \leftarrow$ empty list
4: Create $k$ buckets, one for each character
5: **for** $i \leftarrow d - 1$ down to 0 **do**
6:     $T \leftarrow \mathit{ofLen}[i] + T$
7:     Bucket sort $T$ by char at index $i$
   **return** $T$
```

The loop maintains the invariant that at the start of the loop, the strings in T are sorted by their suffixes from index $i + 1$ on. Prepending $\mathit{ofLen}[i]$, the strings of length exactly i, maintains this property since we imagine strings of length exactly i have nulls at index $i + 1$, so they come first. Then we know that if the strings are sorted by their suffixes starting at index $i + 1$, doing a bucket sort on index i will leave them sorted on their suffixes starting at index i, so the invariant will be preserved for the next loop. When the loop is done, the strings are sorted starting at index 0, *i.e.* they are sorted.

How long does this take? Let $n = |L|$ be the number of strings in L, N the *total number of characters* in L (*i.e.* the sums of all the lengths), d the length of the longest string, and $k = |\Sigma|$ the size of the alphabet (*e.g.* $k = 256$ if we think of characters as bytes; properly sorting Unicode strings is a morass best avoided if at all possible[3]). Line 1 takes $\Theta(N)$. Bucket sorting by length on line 2 takes $\Theta(N + d)$. Creating the buckets on line 4 takes $\Theta(k)$; we do this to explicitly highlight the fact that we don't have to recreate the bucketes each time through the loop. Line 6 will take a total of $\Theta(n)$ over the course of the whole algorithm (given a suitable choice of data structure for T which supports constant-time prepending), since we prepend each string exactly once.

And what about the bucket sort on line 7? It's helpful to think about breaking this down into two phases: first, distributing the strings in T according to index i; then, concatenating all the buckets back into T. Distributing will take a *total* of $\Theta(N)$ over the course of the entire algorithm, since we will look at each character exactly once. Concatenating takes a total of $\Theta(kd + N)$: each time through the loop we have to look at every single bucket (even the ones that don't contain any strings), since we have no a priori way of knowing which buckets will contain strings and which won't; additionally, to concatenate elements back to T we have to $\Theta(N)$ work.

[3] `http://unicode.org/reports/tr10/`

All in all, then, this is $\Theta(N + (N + d) + k + n + N + kd + N) = \Theta(N + kd)$. But that kd is annoying! It comes from the fact that we have to look through *all* the buckets on every pass, even when only a few of them contain anything (which will probably be the case while i is large, and there are only a few long strings in T). If only we knew ahead of time which buckets will have any strings in them, we could just concatenate elements from those, and avoid having to look at any others.

. . . well, we have one more trick up our sleeves. We can, in fact, figure out ahead of time what characters will show up in each position. First, for each character c in any string in the input, create a pair (i, c), where i is the index of c in its string. So we have exactly N pairs. Now, do two bucket sorts: first, bucket sort all the pairs by the character c; next, bucket sort them by the index i. We will end up with one bucket for each index from 0 to $d - 1$, where each bucket contains a sorted list of the characters that occur at index i in any string.

For example, suppose we have the list

$$L = \{\texttt{cat}, \texttt{catastrophe}, \texttt{cut}, \texttt{cot}, \texttt{car}, \texttt{bar}\}$$

Now, when doing the bucket sort of our strings on index i, we know exactly which characters will occur at that index, so we know which buckets will be nonempty.

Algorithm 19 StringSort(L)

```
$d \leftarrow$ maximum length of any string in $L$
$P \leftarrow$ empty list
for $s \in L$ do
    for $i \in 0 \ldots |s| - 1$ do
        Append the pair $(i, s[i])$ to $P$
Bucket sort $P$ by character (second component)
$charsAt \leftarrow$ Bucket $P$ by index (first component), returning $d$ buckets, each
containing the sorted list of characters occurring at index $i$
$ofLen \leftarrow$ Bucket $L$ by length, returning $d$ buckets
$T \leftarrow$ empty list
Create $k$ buckets, one for each character
for $i \leftarrow d - 1$ down to 0 do
    $T \leftarrow ofLen[i] + T$
    Distribute strings in $T$ into buckets by char at index $i$
    for $c \in charsAt[i]$ do
        Pop one string from bucket $c$ and append it to $T$
return $T$
```

Creating P takes $\Theta(N)$. The first bucket sort of P takes $\Theta(N + k)$, and the second takes $\Theta(N + d)$. Finally, because in the main loop we now only look at buckets which actually contain any characters, the sum of all the bucket sort passes now takes only $\Theta(N)$, because we simply do a constant amount of work for each character in each string. Thus, the whole algorithm is $\Theta(N + k + d)$.

Of course, this probably only makes sense when k and d are both much smaller than N, in which case it is just $\Theta(N)$.