CS 161 NOTES: ALGORITHMS

ARUN DEBRAY
DECEMBER 6, 2012

These notes were taken in Stanford's CS 161 class in Fall 2012, taught by Professor Sergei Plotkin. I live-TEXed them using `vim`, and as such there may be typos; please send questions, comments, complaints, and corrections to `a.debray@math.utexas.edu`. Thanks to Amy Nguyen and Emily Cheng for finding a few mistakes.

Contents

1. Introduction and Big O Notation: 9/25/12

There are several ways of comparing different algorithms.

Experimentation is a fine idea, so you can just test and time them against each other. But testing must be done carefully — some algorithms can be better suited to certain parameters or inputs, and sometimes good algorithms are implemented badly, making testing less meaningful.

Average-case analysis would be nice, but the average depends *very* much on context.

Worst-case analysis (asymptotics) gives a rough idea of performance, but really illustrates how something depends on its parameters.

The best way to analyze algorithms is to combine all of these approaches, though this class will focus on asymptotic analysis.

Best-case analysis does exist, but the best case is sometimes sufficiently improbable that it isn't very useful. However, sometimes best-case analysis can be used to show that algorithms are bad.

Example 1.1 (1.)**.** Insertion sort is a straightforward sorting algorithm in which one:

(1) goes over each number in the array in order, and
(2) in order to insert it in the correct place, shifts the larger numbers to the right and smaller numbers to the left in order to make room for it. ◂

This can be done "in-place" (i.e. there is no need to make a copy of the array), which is an optimization for storage.

Analysis of algorithms focuses on correctness and termination. It will often be necessary to prove both; running time can be used to prove termination, but don't waste time trying to calculate the running time of an interminable algorithm, as any algorithm you use to try this will not terminate either. In general, one will want an upper bound on the worst-case, expected-case (on randomized input or on one input many times for nondeterministic algorithms), and best-case.

The best case for insertion sort is when the array is already sorted; if the array contains n elements, then the algorithm performs n operations. However, the worst case is when it is sorted in reverse order; then, the algorithm performs $\frac{n(n+1)}{2}$ operations. (In general, assume that each operation takes a single unit of time; do not worry about architecture or such.)

In the worst-case scenario, the n^2-term dominates, but how should one formalize this? The time of the algorithm on n terms is denoted $T(n)$, and how does this change as $n \to \infty$? Intuitively, one drops the lower-order terms; call this $\Theta(n)$.

Thus, for insertion sort, the inner loop is $\Theta(j)$, so

$$T(n) \approx \sum_{j=2}^{n} \Theta(j) = \Theta\left(\sum_{j=2}^{n} j\right) = \Theta(n^2).$$

However, this informal calculation raises some questions: since $\Theta(1) = \Theta(1) + \Theta(1)$, then shouldn't $\sum_{i=1}^{n} \Theta(1) = \Theta(1)$? And how does one compare the running times of $\log n$ to $n^{1/10}$? In this case, one can't just drop the lower order terms.

One of the formalisms for this is called Big-O Notation:

Definition 1.2. $f(n) = O(g(n))$ if there exist constants c, n_0 such that if $n \geq n_0$, then $0 \leq f(n) \leq cg(n)$.[1]

This means that for sufficiently large n, f is bounded by a constant times g.

One can write $f(n) = O(n) + n^2$, which means there is some $h \in O(n)$ such that $f(n) = h(n) + n^2$. This is useful when it is known that f has quadratic running time plus some linear factor, but said linear factor is too difficult or unimportant to evaluate exactly, and it can be far easier to prove that $h \in O(n)$.

The constant part of the definition is particularly unintuitive, and allows for $f \in O(g)$ even though $f > g$ (such as $f(x) = x^2 + x$ in $O(x^2)$). Additionally, Big-O Notation isn't necessarily specific; $O(n) \subset O(n^2) \subset \ldots$, so it's possible to provide misleading (but technically true) information in this scheme. Thus there are several others used.

Definition 1.3 (small-o notation)**.** $f(n) = o(g(n))$ if $\forall c \in \mathbb{R}^+$, $\exists n_0 \in \mathbb{N}$ such that if $n \geq n_0$, then $f(n) < cg(n)$.

A subset formalism of this exists as well, defined in the analogous manner. But the key difference between this and Big-O Notation is that it requires that eventually, f is less than *any* positive constant multiple of g; for example, $n = o(n^2)$, which can be shown by letting $n_0 = 2/c$ for any $c > 0$.

2. Omega and Theta Notation: 9/27/12

> "When a woman marries again, it is because she detested her first husband.
> When a man marries again, it is because he adored his first wife." – Oscar Wilde

Definition 2.1 (Big-Omega Notation)**.** $f(n) = \Omega(g(n))$ if there are constants c, n_0 such that for all $n \geq n_0$, $0 \leq cg(n) \leq f(n)$.

More formally, if $g : \mathbb{N} \to \mathbb{R}^+$, $\Omega(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}, n_0 \in \mathbb{N}$ such that $\forall n_0 > n$, $0 \leq cg(n) \leq f(n)\}$. The difference between Big-O Notation and Big-Ω notation is that O indicates an upper bound and Ω is a lower bound. Neither of them are tight bounds, though.

[1] Technically, equality makes little sense here, and a more formal definition exists: suppose $g : \mathbb{N} \to \mathbb{R}^+$ (i.e. g returns a running time for a given input value). Then,

$$O(g(n)) = \{f(n) : \exists n_0 \in \mathbb{N} \text{ and } c \in \mathbb{R}^+ \text{ such that } n \geq n_0 \implies f(n) \leq cg(n)\}.$$

Since $O(g(n))$ is a set rather than a function, one would write $f(n) \in O(g(n))$ and then proceed normally.

Definition 2.2 (small-omega notation)**.** $f(n) = \omega(g(n))$ if for every $c \in \mathbb{R}^+$, there exists an $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $0 \leq cg(n) < f(n)$.

Notice that small-omega implies big-omega, or using the subset notation, $\omega(g(n)) \subset \Omega(g(n))$ for any g. But the opposite is very much untrue; consider $f = g$.

Intuitively, O corresponds to $\leq$ and o to $<$ or $\ll$, and Ω corresponds to $\geq$ and ω to $>$ or $\gg$. This is helpful for intuitively understanding definitions but is not that accurate.

Claim. Big-O Notation obeys transitivity: if $f = O(g)$ and $g = O(h)$, then $f = O(h)$.

Proof. Since $f = O(g)$, there are c_1, n_1 such that if $n \geq n_1$, then $0 \leq f(n) \leq c_1 g(n)$.

Since $g = O(h)$, there are c_2, n_2 such that if $n \geq n_2$, then $0 \leq g(n) \leq c_2 h(n)$.

Then, take $n_0 = \max(n_1, n_2)$ and $c = c_1 c_2$, so that if $n \geq n_0$, then

$$0 \leq f(n) \leq c_1 g(n) \leq c_1 c_2 h(n) = c_3 h(n),$$

so $f = O(h)$. ⊠

However, the intuition breaks down in some other examples. Generally, one has $a \leq b$ or $b \leq a$ for numbers a, b, but if $f(n) = n$ and $g(n) = n^{1+\sin n}$, then $f \notin O(g)$ and $g \notin O(f)$. This is a standard trick question, sometimes popping up on interviews.

Theta notation is used when one wants a function to be bounded both above and below by a constant multiple of a function.

Definition 2.3. $f(n) = \Theta(g(n))$ if there exist constants $c_1, c_2 \in \mathbb{R}^+$ and an $n_0 \in \mathbb{N}$ such that if $n \geq n_0$, then $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$.

Thus, Θ establishes a "corridor," in some sense.

An equivalent definition is that $\Theta(g) = O(g) \cap \Omega(g)$.

There is no small-theta notation, however: $o(g) \cap \omega(g) = \emptyset$.

Example 2.4. $n^2/2 - 2n = \Theta(n^2)$. ◂

Proof. Take $n_0 = 8$, and for $n \geq n_0$, $n^2/2 - 2n \geq n^2/4$, but $n^2/2 - 2n < n^2/2$ (you can do the algebra yourself).

Thus, take $c_1 = 1/4$ and $c_2 = 1/2$ and we're done. ⊠

Exercise 2.5. Prove that for polynomials in general, lower-order terms do not matter; if $f(n) = \sum_{j=0}^{k} a_j x^j$, then $f(n) \in \Theta(n^k)$.

Theorem 2.6. *If $f \in O(g)$ and $g \in O(f)$, then $f \in \Theta(g)$.*

Proof. Plug into the definitions: there exist n_1, c_1 so that for all $n \geq n_1$, $0 \leq f(n) \leq c_1 g(n)$.

Similarly, there exist n_2, c_2 such that for $n \geq n_2$, $0 \leq g(n) \leq c_2 f(n)$.

Let $n_0 = \max(n_1, n_2)$ and $c = 1/c_2$, so that

$$0 \leq cg(n) \leq f(n) \leq c_1 g(n)$$

whenever $n \geq n_0$, so $f \in \Theta(g)$ by definition. ⊠

This can be used to provide yet another equivalent definition of Theta Notation.[2]

Now, a more interesting algorithm.

Example 2.7 (Stable Marriage)**.** Suppose there are n men and n women, and each woman ranks every man and every man ranks every woman.[3]

The goal is to match (marry) all men and women such that there are no two pairs (m, w) and (m', w') such that m prefers w' to w and w' prefers m to m'. In the sense of the name, the goal is to prevent cheating, or technically, instability.

[2] Technically, this requires that the theorem guarantees equivalence rather than just implication, which is thankfully the case.

[3] Arun Debray is not responsible for the ideas about marriage or gender relations stated or implied in the metaphorical interpretation of this algorithm. But it's a lot more interesting to read about than the equivalent statements about bipartite graphs.

This problem can be modelled as a bipartite graph (i.e. it splits into two parts such that every arc connects one node of each part). In this terminology, a matching is a legal set of marriages (no polygamy, analogous to an injection) and a perfect matching means everybody is married (sort of like a surjection). Thus, the goal is to find the perfect matching with an additional stability constraint.

This is similar to the problem of the minimal spanning tree or the shortest-path algorithm.

A perfect matching in this case exists, since everyone ranks everyone. A stable set of marriages also exists, though this is harder to show.

A brute-force approach (i.e. checking all possible perfect matchings for stability) certainly works, but there are $n!$ options, which is unfortunate. On the other hand, this guarantees that it terminates. ◂

Algorithm 2.8 (Gale-Shapley)**.** As long as there is a man who is not engaged,

- Pick some free man m, and
- Have m propose to the next woman w on his list who he has not proposed to.
- If w is free, engage m and w.
- Otherwise, if w is engaged to m', then:
  - If w prefers m' to m, nothing happens.
  - Otherwise, engage m and w, and m' becomes free.

It will be shown that this algorithm has polynomial running time and always finds a stable solution, a significant improvement on the brute-force solution:

Claim 2.1. Once a woman is proposed to for the first time (and becomes engaged), she never becomes free, and the sequence of her partners always improves in terms of her preference list.

Claim 2.2. The sequence of women a man m proposes to decreases relative to his preference list.

Claim 2.3. If at some point m is free, then he has not yet proposed to every woman on his list.

Proof of 2.3. Suppose that m has proposed to all women on his list. Then, by Claim 2.1, then all women are engaged. But that means all the men are engaged. Thus, if a man is free, then he cannot have proposed to all women on his list. ⊠

Termination follows from Claim 2.3; eventually, a man will have proposed to everybody on his list, and at this point there is a perfect matching.

Proof of correctness. Assume the algorithm is incorrect, or there exist two couples (m, w), (m', w') such that m prefers w' to w and w' prefers m to m'. By construction, m last proposed to w.

- If m has not proposed to w' beforehand, then m prefers w to w', according to Claim 2.2. But this is a contradiction.
- ...and if he has, then w' rejected him (at proposal or maybe leaving for someone else), by Claim 2.1, since the algorithm guarantees she upgrades her position. This is another contradiction. ⊠

In order to establish the running time, it is necessary to find some measure of progress. Since each of the n men can propose to n women, there are n^2 possible proposals. And each proposal happens at most once. Thus, there are n^2 iterations of the algorithm, and each iteration takes constant time, so the total running time is $O(n^2)$.

3. Analyzing Insertion Sort as a Recursive Algorithm: 10/2/12

A brief look back to the previous class: it was proven that the algorithm is $O(n^2)$, but we can't use $\Theta(n^2)$ or $\Omega(n^2)$ (yet) because there wasn't a worst case demonstrated. It is in fact $\Theta(n^2)$, but that requires a stronger proof.

The basic idea of a recursive algorithm is a divide-and-conquer strategy:

- Divide each problem into at least 2 subproblems.
- Solve each problem recursively.
- Combine the results.

Insertion sort is just a (not very good) divide-and-conquer algorithm! Its two subproblems are the last element and all of the rest, and combining is where the interesting part happens.

This allows one to calculate the running time recursively:[4]

$$T(n) = \begin{cases} T(n-1) + n, & n > 1 \\ 1, & n = 1 \end{cases}$$

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= T(n-3) + (n-2) + (n-1) + n \\ &= \sum_{i=1}^{n} n = \Theta(n^2). \end{aligned}$$

The above proof is not completely formal; a rigorous proof would be by induction. But in this case, the induction is simple.

Insertion sort is reasonable for small n, but $\Theta(n^2)$ is not ideal when n is larger. A standard rule of thumb for divide-and-conquer algorithms is to try and divide the subproblems into roughly equal chunks.

Consider merge sort, which splits the array into 2 equal segments, then calls merge sort on them. It takes linear time to merge two arrays.

$$T(n) = \frac{n}{2}T(1) + \frac{n}{4}T(2) + \frac{n}{8}T(4) + \dots$$

so the overall time is $\Theta(n \log n)$.[5] (The $\frac{n}{2^n}$ terms represent the number of pairs to merge.)

Once again, the intuitive result needs a formal proof. Given the recursive analysis of merge-sort, which solves 2 subproblems of half the size, and merges in linear time, the recurrence is:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + \Theta(n), & n > 1 \\ 1, & n = 1. \end{cases}$$

Formally solving this requires asking some new questions. Does the specific function in $\Theta(n)$ matter? Is it important that n is not always divisible by 2?

In the appendix of the textbook, there are some useful summations that will make analysis of algorithms easier (such as formulas for e^x, the geometric and harmonic series, etc.)

Learning to recognize standard simplifications and/or working in the opposite direction will help find running time. If all else fails, reach into the bag of tricks for something helpful.

Consider an algorithm given by the following recurrence (e.g. binary search):

$$T(n) = \begin{cases} 1, & n = 1 \\ T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1, & \text{otherwise.} \end{cases}$$

Claim. If $n = 2^k$, then $T(n) = \log n + 1$.

Proof. Proof by induction on k. If $k = 1$, then $T(1) = 1 = \log 1 + 1$. Then,

$$T(2^{k+1}) = T(2^k) + 1 = \log(2^k) + 1 + 1 = k + 2 = \log(2^{k+1}) + 1. \qquad \boxtimes$$

Notice that unlike in previous classes, you don't need to state every obvious part of the inductive process; they are implicit, and will also be implicit in the homework and such. Also, it should be clear that the logarithm is base 2 (sometimes written $\lg n$).

If n is not a power of 2, then by induction one can prove $T(n) \geq T(n-1)$, so

$$T(n) \leq T\left(2^{\lceil \lg n \rceil}\right) = \lceil \lg n \rceil + 1 = \Theta(\log n).$$

But! $T(n) \leq \Theta(\log n)$, so $T(n) \in O(\log n)$. One must be careful, though reversing the direction and using floors leads to the same proof for a lower bound, so $T(n) \in \Theta(\log n)$ after all. In general, one can focus on only powers of 2, though one should be careful with the floors and ceilings.

[4] Implicit in this calculation is that the data structure used for the list is an array. If you use a linked list, for example, inserting an element can be done in constant time, but looking up the n^{th} element takes linear time. Thus, the proof, and very possibly the result too, will be different.

[5] The base of the logarithm is irrelevant; by the change-of-base formula, two logarithms of different bases only differ by a constant. But here most logarithms will be base 2.

So the formal proof requires one to know the solution informally, which was possible, but sometimes this can be difficult. Consider the following divide-and-conquer algorithm for addition: divide the sum into two parts, compute the sum recursively, and then add the results. Then,

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1.$$

The floors and ceilings are because not all numbers are divisible by 2; check with some small examples to see why.

The factor of constant time implicitly states we aren't working with variably-sized numbers; some algorithms (especially in cryptography) do involve sufficiently large numbers that this must be considered, however.

One can guess that $T(n) \leq cn$ for a constant c. Then,

$$\begin{aligned} T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \\ &\leq c\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \\ &\leq cn + 1. \end{aligned}$$

This is not in fact a proof of linear time, because of that pesky $+1$ factor. One could weaken the hypothesis (though $T(n) \leq cn + 1$ fails just as badly), but with induction, strengthening the hypothesis often leads to a rigorous proof. Thus, assume $T(n) \leq cn - b$ for some constant b. Then, the proof shows $T(n) \leq cn - 2b + 1 \leq c_n - b$ if $b \geq 1$.

... but we're not done yet. The base case is simple, though; since $T(1) = 1$, let $b = 1$, which indicates $c = 2$.

The actual values of b and c only matter sometimes. Asypmtotically, they aren't important, but when one actually implements things, a large value of c might not be ideal.

Consider another recursive example, where $T(n) = 4T(n/2) + n$. (Breaks into four subproblems, and takes linear time to combine them.)

First guess: maybe $T(n) \leq cn^3$. The base case can be done later, so let's look at the inductive step:

$$4T\left(\frac{n}{2}\right) + n \leq \frac{cn^3}{2} + n = cn^3 + \left(n - \frac{cn^3}{2}\right).$$

If $c \geq 2$ and $n \geq 1$, then the last part is less than zero, so this works.

The base case is straightforward, but this is not a tight bound.

Exercise 3.1. The next guess, $T(n) \leq cn^2$, does not lead to a valid proof. Figure out why.

A better guess is $T(n) \leq c_1 n^2 - c_2 n$. (Once again, strengthening the hypothesis makes everything work.)

$$T(n) = 4T\left(\frac{n}{2}\right) + n \leq 4\left(c_1\left(\frac{n}{2}\right)^2 - \frac{c_2 n}{2}\right) + n = c - 1n^2 - 2c_2 n + n,$$

and the $n - c_2 n$ can be shown to vanish for sufficiently large c_2.

There are some strange cases that reinforce the idea that initial conditions matter, so it can pay to be careful with $T(1)$.

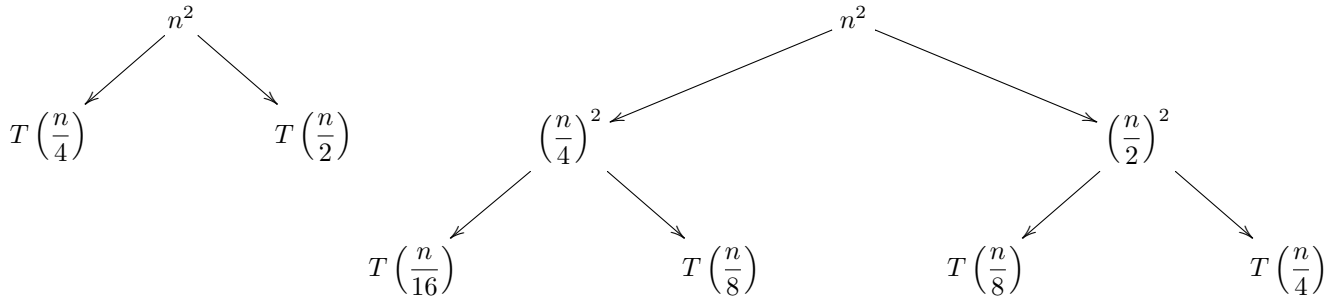
Consider $T(n) = T(n/2)^2$. Then, if $T(1) = 2$, then $T(n) = 2^n$. if $T(1) = 3$, then $T(n) = 3^n \ldots$ but if $T(1) = 1$, then $T(n) = 1$, and the initial conditions are quite important!

Guesses can be obtained by expanding a couple terms out, obtaining a sum that looks informally like some $O(f)$. Of course, such a back-of-the-envelope calculation must be accompanied by a formal proof, but it can serve as a starting point.

4. Recursion Trees and the Master Method: 10/4/12

Though calculating complexity of recursive algorithms always involves some guess-and-check, a recursion tree can add some insight into the problem.

Consider the recurrence $T(n) = T(n/4) + T(n/2) + n^2$. It is possible to write it in a more convenient format:



Then, one can sum up each row to get a guess at the general formula for the row: the first few such terms are n^2, $\frac{5n^2}{16}$, $\frac{25n^2}{256}$, ... At the k^{th} level there is a general formula: there have been i steps to the right and $k-i$ steps to the left (i.e. you followed that many arrows). Thus, if you know the general formula for going left or going right, then you can calculate the general formula of something in a given position and therefore the sums of the rows. In this case, the formula is:

$$n^2 \sum_i \binom{k}{i} \left(2^{-i} 4^{-(k-i)}\right)^2 = n^2 \sum_i \binom{k}{i} \left(4^{-i} 16^{-(k-i)}\right)^2 = n^2 \left(\frac{5}{16}\right)^k.$$

The binomial coefficient stops in because we need to sum over the row, giving a certain number of options, and it goes away because of the Binomial Theorem.[6]

But we're not done yet, since we need to sum over all k. But this is the geometric series, so it sums to $O(n^2)$ (you can overcount since $T(1) = 1$). Then, it must be $\Omega(n^2)$, because of the n^2-term in the formula, so $T(n) \in \Theta(n^2)$.

Another method, called the Master Method, is applicable to the vast majority of recurrences; counterexamples have to be specifically cooked up. Consider a recurrence of the form $T(n) = aT(n/b) + f(n)$, where $a \geq 1$ and $b > 1$.

(1) If $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ for some $\varepsilon > 0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.
(2) If $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
(3) If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$ and $af(n/b) \leq cf(n)$ for some $c < 1$, then $T(n) = \Theta(f(n))$.

Note that the bases of the logarithms do matter here, since powers are taken.

This is something that would be worthwhile to memorize, even though it was presented so quickly that few people had time to write it down. Once we have proved that it works, invoking the method after showing it applies *is* the proof, and no more work needs to be done.

For example, if $T(n) = 2T(n/2) + \Theta(n)$, then $n^{\log_b a} = n^{log_2 2} = n$, so using Case 2, $T(n) = \Theta(n \lg n)$.

As another example, Strassen's matrix multiplication algorithm has a recurrence of $T(n) = 7T(n/2) + \Theta(n^2)$. Using Case 1, $T(n) = \Theta(n^{\lg 7})$, which is measurably better than the obvious matrix multiplication algorithm, which is $\Theta(n^3)$.

If the amount of work to combine the subproblems is high, as in $T(n) = 4T(n/2) + n^3$, then Case 3 is invoked, and this recurrence is $\Theta(n^3)$. This case is rare; Case 2 is by far the most common.

However, this method does not always apply: if $T(n) = 4T(n/2) + n^2/\lg n$, then none of the cases are satisfied. In this case, though, the Master Method can be used to obtain upper and lower bounds. An upper bound is $4T(n/2) + n^2$, which falls into Case 2, so this upper bound is $\Theta(n^2 \lg n)$, and the original recurrence is $O(n^2 \lg n)$.

For a lower bound, consider $4T(n/2) + n^{2-\varepsilon}$ for some $\varepsilon > 0$, which is $\Theta(n^2)$ by Case 1. Thus, the original function is $\Omega(n^2)$. Often this is good enough (though if you're curious, the recurrence is $\Theta(n^2 \lg \lg n)$; the Master Method can be extended to more and more cases, but at some point there's a diminishing return between utility and ease of memorization for the various counterexamples).

The Master Method can be visualized using a recursion tree. In the k^{th} row, there are a^k instances of $f(n/b^k)$, so the sum in that row is $a^k f(n/b^k)$. In the last row, $\Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$ elements, each of which

[6] ... about which, I may add, I am teeming with a lot o' news.

is $\Theta(1)$, since there are $\log_b n$ rows. Then, the total running time is

$$\Theta\left(n^{\log_b a}\right) + \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right).$$

However, this is not entirely useful, as it isn't in closed form. Intuitively, if f is sufficiently small, the sum is negligible, giving $\Theta(n^{\log ba})$, and if f is large, then it dominates, giving about $f(n)$.

Formally, suppose $\frac{n^{\log_b a}}{f(n)} = \Omega(n^\varepsilon)$ (i.e. there is some c such that for large enough n, $f(n) \leq \frac{cn^{\log_b a}}{n^\varepsilon}$, so that f is small). Then,

$$a^j f\left(\frac{n}{b^j}\right) \leq ca^j \left(\frac{n}{b^j}\right)^{\log_b a - \varepsilon} = cn^{\log_b a - \varepsilon} \frac{a^j b^{j\varepsilon}}{b^{j \log_b a}} = cn^{\log_b a - \varepsilon} b^{j\varepsilon}.$$

Thus, summing over all possible j, this becomes (using the geometric series again) $\frac{b^{\varepsilon \log_b n} - 1}{b^\varepsilon - 1} = \Theta(n^\varepsilon)$. Thus, the total is $O(n^{\log_b a})$. In order to prove $\Theta(n^{\log_b a})$, observe that $T(n) = \Theta(n^{\log_b a}) + O(n^{\log_b a})$, which just folds into $\Theta(n^{\log_b a})$.

Thus Case 1 is proven. Case 3 is very similar, and no more interesting. So consider Case 2.

If $f(n) = \Theta(n^{\log_b a} \lg^k n)$, so that the total sum is (analogously)

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right).$$

But since there are $O(\lg n)$ elements in the sum, then it is possible to bound the sum:

$$\sum_{j=0}^{\log_b n - 1} a_j \left(\frac{n}{b^j}\right)^{\log_b a} \log^k \left(\frac{n}{b^j}\right) = O(\log^{k+1} n) n^{\log_b a}.$$

Since there are $\log_b n$ elements in the sum, and each element is at most $\lg^k n$, then their total is $\lg^{k+1} n$, which was probably the most mysterious part of the bound.

The lower bound is an entirely different beast:

$$\sum_{j=1}^{\log_b n - 1} \lg^k \left(\frac{n}{b^j}\right) \geq \sum_{j=1}^{\frac{\lg_b n}{2}} \lg^k \left(\frac{n}{b^j}\right) \geq \sum_{j=1}^{\frac{\lg_b n}{2}} \lg^k \sqrt{n} = K \lg^{k+1} n$$

for some constant K. Though these sums may look cryptic, they are mostly formed by removing some of the terms from the original sum. It is assumed that $k \geq 0$, and it will be formally necessary to deal with $(\log n)/2 \notin \mathbb{Z}$, but this is also straightforward.

5. Conditional Probability and Randomized Algorithms: 10/9/12

Conditional probability is the probability of one event happening given (or on the condition of) another. Venn Diagrams can be used to make this make more sense; the probability of A given B is the probability of $A \cap B$ in B.

$$\Pr[X = i \mid Y = j] = \frac{\Pr[X = i, Y = j]}{\Pr[Y = j]}.$$

Conditional expectation is straigntforward: $E_y[E_x[X \mid Y]] = E[X]$. This is just the expected value of the conditional event.

Example 5.1. Consider tossing one fair die. Let X be the result of the tie, and Y be the event that $X = 2$ (i.e. $Y = 1$ if $X = 2$, and $Y = 0$ otherwise).

This is a simple example of a condition on Y, so $\Pr[Y = 0] = 2/6$ and $\Pr[Y = 1] = 4/6$. Also,

$$E[X \mid Y = 0] = \sum_i i \Pr[X = i \mid Y = 0] = \frac{3}{2}$$
$$E[X \mid Y = 1] = \sum_i i \Pr[X = i \mid Y = 1] = \frac{9}{2}$$
$$E[X] = E[X \mid Y = 0] \Pr[Y = 0] + E[X \mid Y = 1] \Pr[Y = 1]$$
$$= \left(\frac{3}{2}\right)\left(\frac{2}{6}\right) + \left(\frac{9}{2}\right)\left(\frac{4}{6}\right) = 3.5,$$

which is just the unconditional probability as was already known. ◂

Now consider Quicksort, which should be familiar:

Algorithm 5.2 (Quicksort)**.** Quicksort sorts an array in place:

(1) Divide the array into two sub-arrays around the first element.
(2) Recursively Quicksort each array.
(3) Merge and combine (which is trivial, because this is an in-place sort).

The partition routine is as follows:

```
Partition(A,p,r)
    x = A(r)
    i = p - 1
    for j = p to r - 1
        if A(j) <= x
            then i++, exchange A(i) and A(j)
    exchange A(i+1) and A(r)
    return (i+1)
```

Quicksort looks like this:

```
Quicksort(A,p,r)
    if p < r
        q = partition(A,p,r)
        quicksort(A,p,q-1)
        quicksort(A,q+1,r)
```

Suppose for simplicity that the elements of the array are distinct (the algorithm is basically the same otherwise, but that requires more casework).

If Quicksort is lucky, then the partition is always an even split. Then,

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n).$$

On the other hand, if Quicksort is maximally unlucky, then the partition splits Quicksort into an array of one element and an array of $n - 1$ elements. In this case, Quicksort acts more like insertion sort, and

$$T(n) = T(0) + T(n - 1) + \Theta(n) = \Theta(n^2).$$

This is not a proof that this is the worst case, but intuitively it feels like it, and it's worth learning how to avoid the bad cases. In general, partitioning around the middle element does not work; it is in now way guaranteed to be the middle element in value. One should always assume that an adversary might feed your algorithm the worst case. But if you partition around the first entry of the array, the worst case is the already sorted array.

So the only way to avoid this adversary is to randomly decide which element to partition around.[7] This is the basis for randomized algorithms: they make worst cases much harder to hit (though they're still possible, of course).

The ideas behind a randomized algorithm are as follows:

- The algorithm can "toss coins."

[7] In practice, many implementations of Quicksort randomly permute the array and then select the first element. This is equivalent but often easier to implement.

- No specific input always leads to worst-case behavior.
- The data is not expected to be random, but the algorithm, which is a noteworthy distinction.

Analysis of running time is a bit more interesting than in deterministic cases; let $T(n)$ be the expected time to sort.

Consider the case where the partition is $(k, n-k-1)$. Then, conditioned on the probability of this partition, the expected time to terminate is $T(n) = T(k) + T(n-1-k) + \Theta(n)$.

Since every value of k from 0 to $n-1$ is equally likely, then one can use the formula for conditional expectation to write $T(n)$ in terms of only n, not k.

$$\begin{aligned} T(n) &= E_k[T(n \mid (k, n-k-1))] \\ &= \sum_{k=0}^{n-1} \Pr[(k, n-k-1)] T(n \mid (n-k-1))] \\ &= \frac{1}{n} \sum_{k=0}^{n-1} (T(k) + T(n-1-k) + \Theta(n)) \\ &= \left(\frac{2}{n} \sum_{k=0}^{n-1} T(k) \right) + \Theta(n). \end{aligned}$$

So now there's a recurrence, albeit a very nasty one that cannot be solved with the Master Method. Solving it is difficult, though $T(n) = \Theta(n \log n)$ is a natural guess.

Claim. $T(n) \leq an \lg n + b$ for some constants a, b.

Proof. First, choose a b large enough to satisfy $T(1) \varepsilon a \lg 1 + b = b$.

Now, consider the inductive step, and recall that $T(0) = 0$:

$$\begin{aligned} T(n) &= \left(\frac{2}{n} \sum_{k=0}^{n-1} T(k) \right) + \Theta(n) \leq \frac{2}{k} \sum_{k=0}^{n-1} (ak \lg k + b) + \Theta(n) \\ &= \frac{2a}{n} \sum_{k=1}^{n-1} k \lg k + 2b\Theta(n). \end{aligned}$$

It is necessary to show that $\sum_{k=1}^{n-1} k \lg k \leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$; showing that $\sum_{k=1}^{n-1} k \lg k \leq n^2 \lg n$ is insufficient, even though it seems natural. See below for the proof.

$$\begin{aligned} &\leq \frac{2a}{n} \left(\frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \right) + 2b + \Theta(n) \\ &= an \lg n + b + \left(\Theta(n) + b - \frac{an}{4} \right), \end{aligned}$$

and this last part goes to zero for sufficiently large a.

Here is the proof for the bound of $\sum k \lg k$, which uses the standard trick of splitting the sum into two parts and bounding each part. It's also possible to answer it by integration (specifically, integration by parts

in this example).

$$\begin{aligned}
\sum_{k=1}^{n-1} k \lg k &= \sum_{k=1}^{\lceil n/2-1 \rceil} k \lg k + \sum_{\lceil n/2 \rceil}^{n-1} k \lg k \\
&\leq \sum_{k=1}^{\lceil n/2-1 \rceil} k \lg n - \sum_{k=1}^{\lceil n/2-1 \rceil} k + \sum_{\lceil n/2 \rceil}^{n-1} k \lg n \\
&\leq \lg n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2-1 \rceil} k \\
&\leq \lg n \left(\frac{n(n-1)}{2} \right) - \frac{(n/2-1)(n/2)}{2} \\
&\leq \frac{1}{2} n^2 \lg n - \frac{n^2}{8}.
\end{aligned}$$

So there's actually more work yet to do, since this is just showing $O(n \lg n)$, though $\Omega(n \lg n)$ is similar. ⊠

Here are some examples that illustrate how powerful randomization can be on some algorithms.

Example 5.3. Consider some undirected graph G, and try to divide the set V of nodes into two subsets such that at least half of the edges connect a node in one part to a node in another part.[8] ◂

Algorithm 5.4. Assign each node to right or left with probability 1/2.

This seemingly silly algorithm is actually quite powerful. Define an indicator

$$I(u, v) = \begin{cases} 1, & \text{if } uv \text{ crosses} \\ 0, & \text{otherwise.} \end{cases}$$

The probability that an edge crosses from left to right is 50%: $\Pr[I(u, v) = 1] = 0.5$.

The expected number of crossing edges is

$$E\left[\sum_V I(u, v)\right] = \sum_V E[I(u, v)] = \sum_V ((1) \Pr[I(u, v) = 1] + (0) \Pr[I(u, v) = 0]) = \sum_V 1/2 = \frac{|E|}{2}.$$

So the expected value indicates that a solution exists; if there wasn't a solution, then the expected value would have to be less than half.

This algorithm produces an answer in expectation, so more work is needed to make it work in more cases. But this requires some more complicated probability and will be addressed later. Consider another example:

Example 5.5. Given an undirected graph G, what is the smallest number of edges such that deleting those edges separates G into two parts? ◂

This is akin to blocking off some roads in a city to separate it, I guess for a siege or something. Additionally, the solution coul be 0 if the graph is disconnected. This can be checked at the start of the algorithm, however.

Algorithm 5.6 (Karger)**.** (Fun fact: Karger was a Stanford student at some point.)

(1) Pick a random edge, and identify its two nodes as a "supernode."
(2) Repeat this until there are only two nodes left.
(3) Return the number of edges connecting these nodes.

Claim. This algorithm produces a global cut with probability at least $P = \frac{k}{n^2}$ for some constant k, if there are n nodes in the graph.

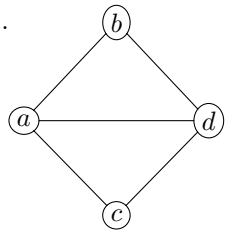
[8] For a related randomized algorithm, consider Nick Wu's proof that a bipartite graph is two-colorable.

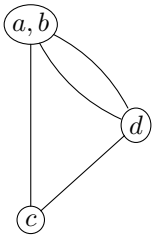
6. More Randomized Algorithms: 10/11/12

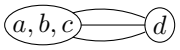
Looking again at the global cut problem, it is clear that the minimum number of cuts needed to disconnect the graph is at most the minimum degree of any node of the graph (because removing all of the edges of a single node disconnects the graph).

Since Karger's algorithm doesn't always return the minimum cut, the actual strategy for producing a solution is to run it many times and find the minimum of the returned results (since it always returns a disconnected graph). This is of course not guaranteed towork, but it will return the correct solution with a known probability: $\Pr\left[\left(1-\frac{1}{n^2}\right)^{n^2}\right] = \frac{1}{e}$ for sufficiently large n.

Consider how the algorithm runs on the following graph:

1. 

2. 

3. 

(Here it actually fails, since the minimum cut is of the two edges that connect node c).

If the global minimum-cut size is k, then each supernode has degree at least k, and the total number of edges is at least $\frac{n_i k}{2}$, where n_i is the number of supernodes. Thus, if m_i is the number of the edges in the cut and m the total number of edges, then the probability that the contraction "kills" a specific cut is $m_i/m \leq 2/n_i$.

Additionally, the number of supernodes decreases by 1 every iteration. Thus, the probability that each of the edges in the cut is still "alive" in the end is

$$\prod_{j=1}^{n-1}\left(1-\frac{2}{n-j}\right) = \frac{2}{n(n-1)} = \binom{n}{2}^{-1}.$$

This example should illustrate that there are more interesting examples of probabilistic algorithms than just Quicksort.

Another question is: given points $(x_i, y_i) \in \mathbb{R}^2$ for $i = 1, \ldots, n$, what two points are closest to each other? The obvious, brute-force approach is $\Theta(n^2)$. But one can make a better divide-and-conquer algorithm:

- Divide the plane into 2 parts using a vertical line L by obtaining the median of the x-values of the points.
- One can then compute recursively the closest pairs on the right and the left.

If this is it (and the algorithm just returns the minimum value of these two), the running time is $\Theta(n \log n)$ — but this isn't even correct! Two points could be separated by an arbitrarily small distance across the line L. Thus, the problem becomes one of correctly combining the results.

If d is the minimum distance between any two points on the right or on the left, then any two points which are less than d apart (on opposite sides of the line) must be less than d away from L. In some sense, they are on a "band" of width d on either side of L.

Unfortunately, this makes the divide-and-conquer approach harder: the left half of the problem is half as large as the original problem, and the right half is similar, but the band in the middle might be just as complicated as the original problem. So some ingenuity is required.

Consider the points in the band, and do the following:

(1) Sort the points by their y-coordinates (realistically, this will be done in the beginning, so that this takes less time in each individual step).
(2) Place a grid of size $d/2$ onto the band.

Then, there must be at most one point per cell; since the cells do not cross the boundary, if there were two cells in the same grid, d would be less than $d/\sqrt{2}$, since both points would be on the same side of the line.

Then, for any particular point in the band, there are only 16 cells that are less than d away from it in the positive y-direction (and the others can be ignored, since if you search in order of increasing y the lower possibilities have been covered), and one of these contains the point itself. So for each point, there are only 15

possible points. (The algorithm looks at points rather than cells because it doesn't compute which cell a point lies in. Since there is at most one point per cell, this still works). Thus, searching the band takes linear time.

Then, the overall running time is $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$.

This leads to the problems of order statistics. In general, one wants to find the i^{th} smallest element in the array (if $i = 1$, then this is the minimum; if $i = n$, this is the maximum, and if $i = n/2$, this is the minimum). A possible solution is to sort and then return $A[i]$, but this is inefficient. But to find one (e.g. the maximum), it just takes linear time (and even for arbitrary i, it can still be done in linear time by removing the lower values).

However, it will necessarily take at least linear time, because the algorithm has to read every element of an array, which will take linear time.

The trick is to used randomized selection, which also involves a divide-and-conquer approach:

```
RS(A,p,r,i)
    if p == r then return A(p)
    q = RandomPartition(A,p,r)
    k = q - p + 1
    if i < k then return RS(A,p,q-1,i)
    if i > k then return RS(A,q+1,r,i-k)
    if i == k then return A(q)
```

By computing where the smallest elements lie, this algorithm decides whether it needs to recurse to the left or to the right.

The proof of correctness is straightforward: assume it is correct for size at most $n = r - p + 1$; after the partition, the arrays are smaller than n, so one can apply induction.

Claim. It is only necessary to search one of the parts of the partition.

The proof will have to consider each of the three cases.

Running time: in the lucky case (in which we have to search in the larger side only 10% of the time), $T(n) \leq T(9n/10) + \Theta(n)$. Using the Master Method, this yields $T(n) = O(n)$.

If the guarantee is 99/100, then it's still $O(n)$, which suggests taking a limit...

7. Order Statistics: 10/18/12

Continuing from the previous lecture, if the random-selection method hits the worst case (i.e. always choosing the largest possible element to pivot around), then $T(n) = T(n-1) + \Theta(n)$, so $T(n) = \Theta(n^2)$.

So condition on the outcome of the partition:

$$\begin{aligned} T(n) &= E_{\text{outcome}}[\text{Expected time conditioned on last partition outcome}] \\ &\leq \frac{1}{n} \sum_{k=0}^{n-1} T(\max\{k, n-k-1\}) + \Theta(n) \end{aligned}$$

because the larger running time will outweigh the smaller one, and the algorithm only looks on one side.

$$\leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} T(k) + \Theta(n)$$

Though it looks ambiguous as to whether the $\Theta(n)$ term is inside or outside of the sum, it doesn't actually matter which is the case.

Though this doesn't fit into the Master Method, it's reasonable to guess that $T(n) \leq cn$ for a sufficently large c:

$$\begin{aligned} T(n) &\leq \frac{2c}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} k + \Theta(n) \\ &\leq \frac{2c}{n} \left(\sum_{k=1}^{n-1} - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k \right) + \Theta(n) \\ &\leq \frac{2c}{n} \left(\frac{n(n-1)}{2} - \frac{(n/2-2)(n/2-1)}{2} \right) + \Theta(n) \\ &= cn + (k_1 + \Theta(n) - ck_2 n) \end{aligned}$$

for constants k_1, k_2, so $T(n) = O(n)$. However, it is also $\Omega(n)$, because picking a first pivot necessarily takes linear time, so it cannot do better. Thus, $T(n) = \Theta(n)$.

Though this is a randomized algorithm, it always produces the correct answer; in the worst case, it is just slow. However, it is very fast in practice (just like Quicksort, and the same additional tricks will help speed it up).

It is a theoretically interesting question to ask if there is a linear-time algorithm for order statistics that is deterministic.

"Such an algorithm exists, but it is a magic:"

- Divide the n elements into groups of 5.
- Brute-force find the median in each group of 5 (which takes constant time).
- Recursively, find the median among these $n/5$ medians, and partition around it.
- Now recurse based on the partition in the same manner as before.

So why 5? For now, think of it as a magic number.

Correctness is as before; the only thing that has changed is the choice of the pivot, so it still works. However, analyzing running time is a bit more convoluted, since there are two recursive calls.

Let x be the median-of-medians chosen. Half of the medians are less than x, or that $\frac{1}{2} \left\lfloor \frac{n}{5} \right\rfloor \geq \left\lfloor \frac{n}{10} \right\rfloor$.

Overall, each median is larger than two of the five elements, so each group of 5 takes 3 elements that might be smaller than the pivot, so overall 30% of the elements are smaller than the pivot (since half of the medians are).

Thus, for $n \geq 50$, $3\lfloor n/10 \rfloor \geq n/4$, so at least $n/4$ elements are less than (or equal to) the pivot. In the same manner, at least $n/4$ of the elements are greater than or equal to x. Thus, the recursion will be on at most 75% of the elements. Specifically, $T(n) \leq T(n/5) + T(3n/4) + \Theta(n)$. (The constant term comes from finding the median in the group of 5).

Choose a c large enough to cover $T(1)$, so that it's possible to show that $T(n) \leq cn$:

$$T(n) \leq \frac{cn}{5} + \frac{3cn}{4} + \Theta(n) = cn + \left(\Theta(n) - \frac{cn}{20} \right).$$

So this is linear — but c is large enough that it multiplies all of your other constants by 20, which is not practical. In particular, 5 was chosen so that this above equation works: not every number works. See the homework for more details.

Thus, this algorithm is $O(n)$. It is also $\Theta(n)$, for the same reasons that the randomized algorithm is.

So this strategy can be applied to get a deterministic variant of Quicksort, since the randomness just involves picking a pivot. But this is incredibly slow, even though it's still $\Theta(n \log n)$.

A data structure can be thought of as a black box which supports queries (of information contained in it) and updates (of the information in it).

Example 7.1. Consider using a priority queue to simulate phone calls of a known length. Each event contains a start time and an end time, so the simulator picks a new event, processes it, and perhaps updates the queue.

Suppose after entering a new event, the algorithm wants to pick the "next" event (i.e. the one with the smallest time key). ◁

There are several possible approaches:

- One could keep all the elements in an unordered linked-list. Then, insertion is $O(1)$, but extraction is $\Omega(n)$.
- One could keep the list in sorted order, so that extraction is $O(1)$, but insertion is $\Omega(n)$ (since binary search doesn't work on a linked-list).

This is a common pattern: one solution is good at one thing and bad at something else, and the data structure in question requires tradeoffs.

It would be nice for insertion and extraction to work in $O(\lg n)$ time, so consider a heap. A heap is a nearly complete binary tree, with the heap property: the parent's key is always greater than the child's key.

By induction on the layer of the heap, the maximum is always at the root of the tree.

A heap can be stored in an array, which is much less cumbersome than the pointer model. The parent of the node at index i is stored at index $\lfloor i/2 \rfloor$. Thus, the children of node i are at $2i$ and $2i+1$ (unless that's outside of the array, in which case that node doesn't have that child).

8. Heaps: 10/23/12

The array representation of a heap is not the only possible one; you could maintain it as a tree of nodes and pointers, but this is simply more inefficient. However, it is important to keep track of the size of the array, so it's clear which nodes have children and which nodes don't.

There is one fundamental idea, one tool, for heaps after which everything else is just details. It is to fix a single incorrectness in the heap property. Without loss of generality, one can assume the problem is at the root (since the children of any node are also heaps, so you can just consider that subheap). If the heap is broken in that manner, just exchange it with its largest child.

This doesn't necessarily fix the heap; it's still possible for this child to be larger than one of its children. But then the same process can be applied to that subheap, and so on. This process is guaranteed to terminate, since the heap is finite and this process keeps the number of problems constant or decreasing. Thus, the algorithm is correct (though formally, this would be accomplished by induction on the height of the heap).

The procedure itself is called Heapify(A, i, n), where A is the heap (represented as an array), i is the index, and n is the number of elements in the heap. Its complexity is $O(\log n)$ (easy proof by induction; not $\Theta(\log n)$ because the heap isn't a complete binary tree). Part of the reason this is so fast is because one can access the children of a given node in constant time.

Insertion is very similar to Heapify:

```
last++
A(last) = new element
i = last while parent(i) != null
    if A(i) <= A(parent(i)) return
        else exch.  A(i),A(parent(i))
            i = parent(i)
        end
end
```

One could present a general proof again, and one can show that insertion takes $O(\log n)$ time. Correctness does need a bit more of an explanation, though.

Extracting the maximum is also similar: take the root, put the last element there, and heapify. This also takes $O(\lg n)$ time, since it creates exactly one problem.

Building a heap initially is a bit more interesting, since one should consider the worst possible data for a given algorithm. One possibility is to Heapify(A, i, n) from $i = n$ to $i = 1$. Thus, it adds elements from the bottom up, so all the subtrees are correct at any given moment. This easily has the bound of $O(n \log n)$ — but it can be shown that it's actually linear. The trick is as follows: half of the elements are at the lowest layer, and so on. So the lower-level loops take less time, since their subheaps' heights are smaller.[9] (This is technically only true for a complete binary tree, but the timing difference is just as effective.) Thus, the time

[9] This is true not just of heaps, but also trees in general. Remembering that most of the nodes are in the bottom layer means some algorithms are faster than expected.

is

$$T(n) = \sum_{i=1}^{k} i2^{k-i} = 2^k \sum_{i=1}^{k} \frac{i}{2^i} \leq 2^k \sum_{i=1}^{\infty} \frac{i}{2^i} = 2^{k+1} = O(n),$$

where k is the depth of the heap. (The sum itself can be calculated with the geometric series.)

Another useful trick is amortized analysis. Suppose someone pays \$1 for each iteration of Heapify, and initially sprinkle \$2 on each node (i.e. a "savings account" that the payments come out of). Then, if you can prove that the budget is not exhausted, you have linear time.

This can be proved by inductively by assuming that a heap of height h has $\$h$ left (and the base case is trivial). Inductively, there are $2 + h + h$ dollars left at step $h + 1$, and the height of this subtree is $h + 1$, so $h + 1$ iterations of Heapify are needed. This leaves $\$(h + 1)$, so the inductive step is shown.

There are some variations on heaps: it could be rooted at the minimum, and the comparisons would be inverted. It could also be k-ary instead of binary, in which case each node has k children. This has about the same theoretical performance, but it sometimes works better in the real world.

One can sort using heaps: build the heap and then remove the maximum. This can be done in place: when one removes the maximum, the last place is freed, so the maximum can be exchanged with the last element of the heap. Thus, one can sort by repeatedly removing the maximum. (In some sense, the maxima are placed at the end of the array, after the now-smaller heap, so they don't get caught up in successive Heapifies). There are n Heapifies, and each takes $O(\log n)$ time, so the running time is $\Theta(n \log n)$ (since most of the Heapifies will be when the heap is larger).

It seems that sorting always seems to take $\Omega(n \log n)$ time. Maybe this is true in general — but so far, all the sorting algorithms presented have been comparison sorts, in which the only operation allowed on data is comparison. One can represent computation by a decision tree. A decision tree of size n solves all comparison sorts of n elements: if $A[1] > A[2]$ and $A[2] > A[3]$, then the array is rearranged into $\{A[3], A[2], A[1]\}$. These decisions and their results can be built up into a binary tree. Thus, if one can bound the longest path through this decision tree, this corresponds to bounding the execution time.

There is 1 leaf for each possible answer, so there are $n!$ possible leaves (since there are $n!$ permutations). And even a complete binary tree with $n!$ leaves must be $\Omega(n \log n)$ deep (since the number of leaves in each layer is exponential), so the worst-case execution time is $\Omega(n \log n)$.

Exercise 8.1. Why doesn't this argument work for selection (i.e. order statistics, finding the median, etc.)?

Solution: There are only n possible answers for the location, so the lower bound is $\Omega(\log n)$. (This is a nonsense lower bound, since you need to look at every element anyways, so the running time must be $\Omega(n)$.) ⊠

Is $\Omega(n \log n)$ really the limit in all cases? Not exactly.

Algorithm 8.2 (Counting Sort)**.** Suppose the inputs are in $\{1, \ldots, k\}$ and are all in $\mathbb{N}$. Then, the algorithm literally counts the number of elements with value i there are for each i, and then spits them back out.

Another example worth considering is radix sort.

9. Hashing: 10/25/12

First, a bit more on Counting Sort from the last lecture. It's not a comparison sort — its running time is $\Theta(n + k)$ (since it is necessary to initialize the array of counts, which takes k steps). Worse, though, for extreme k, the space is exponential in k (since it's bitwise), so it is impractical for large values.

A heap supports insert, delete, and returning the min or the max. (Delete wasn't covered in the last lecture, but given a pointer one can just remove it and re-heapify.) However, searching for a given element is less than optimal. One can search just by pulling out each maximum, which takes $O(n \log n)$ time because Heapify is called each time. It's actually faster to just go through each element of the array, which takes $\Theta(n)$ time.

So maybe there's a better solution than a heap:

- In an ordered array, binary search operates in $O(\log n)$ time, but insertion and deletion take $\Omega(n)$ time, since elements have to be moved around to preserve the sorted order
- In an ordered (linked-)list, both find and insert are slow, since binary search isn't viable.

In hashing, one maintains a direct address table $T[i]$ such that

$$T[i] = \begin{cases} x, & x \in T \text{ and } \text{key}(x) = i \\ \text{NULL}, & \text{otherwise.} \end{cases}$$

Thus, searching is extremely straightforward: just look up the entry of the array at index i. Insertion is also straightforward, since you just update the value of the array at index i. Removing elements is easy for the same reason.[10] However, the tradeoff is space: this address table is as large as in counting sort: it maintains an index for every possible value of the key. This is very bad if the key range is large.

One can instead make a smaller table. This causes collisions (i.e. two keys map into the same slot). Collisions are resolved by putting a linked list at each index. Thus, collisions are added to the same linked-list, in a process called chaining.

Another strategy, called open addressing, is to resolve collisions by checking the next slot if a given slot is full. ("Next" is a bit more interesting than just adding 1 to the index; this can have significant implications on performance.)

In terms of the amount of work needed to do (i.e. walking through the linked-list and possibly adjusting pointers), inserting, removing, and searching all take the same amount of time.

In order to analyze this algorithm, a very bad simplifying assumption is made: that each key is equally likely to be hashed to the same slot. If there are n keys and m slots, then define the load factor to be $\alpha = n/m$, and let I_j be an indicator: $I_j = 1$ iff element j hits the slot in question, so that $\Pr[I_j = 1] = 1/m$.

The expected length of the chain is the expected number of elements that were hashed into a given slot:

$$E\left[\sum I_j\right] = \sum E[I_j] = \sum_{i=1}^{n} \frac{1}{m} = \frac{n}{m} = \alpha.$$

This should not come as a surprise, since the n elements are spread evenly throughout m locations. Thus, access time (and thus searching, inserting, and removing) is $O(1+\alpha)$, and the expected length of a randomly chosen list is also $O(1+\alpha)$. Search will actually take $O(1+\alpha/2)$, but this is not very different.

Specifically, if insertion is at the end of the linked-list, the expected time to find the i^{th} element is equal to the time it takes to insert that element. So the insertion time grows with i, since there are more elements in the table to iterate along.[11] Thus, assume that they key being searched for is equally likely to be any one of the keys stored. Then, the conditional expectation for the search being for the i^{th} element is $1 + \frac{i-1}{m}$ (since that was the load factor just before element i was inserted), so the overall expectation is

$$\begin{aligned} \sum_{i=1}^{n} \Pr[i] E[T(i)] &= \frac{1}{n} \sum_{i=1}^{n} \left(1 + \frac{i-1}{m}\right) = 1 + \frac{1}{nm} \sum_{i=1}^{n} (i-1) \\ &= 1 + \frac{n(n-1)}{2mn} = 1 + \frac{\alpha}{2} - \frac{1}{2mn}. \end{aligned}$$

This last factor is not very significant.

One important aspect of a hash table is choosing a hash function. This is basically a black art. However, there are some examples:

- The division method, for which $h(k) = k \bmod m$, in effect ignoring the largest bits in a number if m is a power of 2. This is very bad if the least significant bits have something in common (e.g. far more likely to be even than odd, or tend to end in zero). However, if m is a prime not too close to a power of 2 or 10, this can work better.
- The multiplication method, where $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$. Here, m cannot be a power of 2 (otherwise, this is just choosing the largest bits, which is problematic), and A is some number between 0 and 1.

  This can be thought of as hashing into a circle $\mathbb{R}/m\mathbb{R}$: each time k is incremented, travel a distance A around the circle, and read off the sector number (or I guess equivalence class). A good choice of A is needed to ensure that all the elements of the table are accessed more or less equally.

The hash function is easy to make constant, but a good hash function needs to be fast absolutely as well as asymptotically, since it will be called frequently.

[10] As you might have noticed in CS 107, "easy" refers to running time, not necessarily implementation.

[11] Though rehashing could offset this, it's not yet part of the algorithm, for simplicity.

This still uses lots of memory for null pointers (if there aren't any keys at a given hash address). Open addressing attempts to rectify this situation. Here, the hash function is $h(k, p)$, where k is the key and p is the probe number: it starts at 1, and if there's an element at that address, then p is incremented and the hash function is called again. However, it's not obvious that the algorithm terminates, and this will be addressed[12] shortly.

Searching is fairly nice: start by getting the key at $h(k, 1)$, and if it's not what is looked for, then try $h(k, 2)$, etc. and halt when an empty cell is returned or the right value is found.

However, open addressing relies on the assumption that there are no removals! Working them in tends to be much more complicated than chaining, which is why chaining is so much more common in practice.

In the analysis of open addressing, assume again that the hash function is random and uniform. Then, the probability that at least i probes lead to occupied slots is $q_i = (n/m)^i = \alpha^i$. In an unsucessful search, there will be the expected number of probes plus one, which evaluates to

$$1 + \sum_{i=1}^{\infty} i \Pr[h(i+1) = \text{NULL}] = 1 + \sum_{i=1}^{\infty} q_i = 1 + \sum_{i=1}^{\infty} \alpha^i = \frac{1}{1-\alpha}.$$

(The probability being calculated is that it takes exactly i probes to complete the unsuccessful search.) One can show that $\sum_{i=k}^{\infty} = q_k$, which leads the probability to the expression involving α. But... as the load factor increases to 1, this rockets off to infinity, and the system becomes unworkable. Thus, there's an incentive to keep $\alpha \approx 1/2$, and it really doesn't make a huge difference if α is much smaller, and that payoff comes with a huge space gain.

For a successful search, elements inserted earlier will be easier to find again. The expected number of probes for element i is less than $\frac{m}{m-i}$. Conditioning on i, the expectation is

$$E \le \frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} = \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} = \frac{1}{\alpha}(H_m - H_{m-n}),$$

where H_n is the harmonic function. Since $\ln i \le H_i \le \ln i + 1$, then this becomes

$$E \le \frac{1}{\alpha}(\ln m + 1 - \ln(m-n)) = \frac{1}{\alpha}\left(\ln\left(\frac{m}{m-n}\right) + 1\right)\ldots$$

10. Bloom Filters and Binary Search Trees: 10/30/12

Some comments on hashing:

- Good hash functions are hard to invent. Look in textbooks, etc. before creating your own.
- The analysis discussed relied on some heavy assumptions that aren't entirely correct.
- An adversary can make any specific hash function fail by creating a huge number of collisions.
- There is a notion of a "universal hash function," in which the algorithm uses a different hash function each iteration, so the adversary has a much harder job. There is a formal proof that such families of hash functions exist, and they're even practical in several cases, since all you need to remember is the index of the hash function.
- The difference in performance between chaining and open addressing is worth remembering. However, open addressing requires considerably less memory. However, in open addressing the time to find an element isn't very pretty.

One can also consider a variant on hashing called the Bloom filter.

Standard hashing has no mistakes. If the element is in the set, it returns true, and if not, it returns false. Is it possible to reduce the space at the expense of a low error rate? Specifically, if the element is not present in the set, it might be possible to return a false positive with some low probability.

There are plenty of cases where this would be useful:

- In a dictionary, one might miss a small fraction of misspellings.
- A distributed cache of Web pages — if a nonexistent page is requested infrequently, it's OK.
- This might also be helpful if there's not space to store the objects themselves, but rather some set of bits representing whether they are present or not.

[12] No pun intended.

The Bloom filter is a long bit vector with n elements and k different (independent) hash functions (hence k is usually pretty small, even though n can be very large). Each element is hashed into a range from 1 to m by one of the hash functions, and the result is stored in a bit vector of length m (so when hashing, set each of $A[h_i(x)]$ to 1 for all of the hash functions h_i).

Then, to check if an element is present in the set, return yes if $A[h_i(x)]) = 1$ for all h_i and no otherwise. So if the element is present, this operation will always be correct, but an unlucky combination of values in the bit vector might cause a false positive.

Suppose there are n keys. Then, the probability of a specific bit in A still being 0 is more complicated than just multiplying nk, since some redundancy might occur —the hash function changing a bit from 1 to 1. Thus, the probability of a bit being zero is $p = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-\frac{kn}{m}}$, since $\left(1 - \frac{1}{m}\right)^{m} \approx e^{-1}$.

The probabillty of a false positive is thus $(1-p)^k$ (since the key that is not there is k times the probability that a given bit is 1). Thus, the goal is to make this probability small.

Since m and n might be dictated from context, the only thing we really have control over is k. When inserting, it's better for k to be small, and when checking, it is better for k to be large.

Rewrite the false positive probability as

$$(1-p)^k \approx \left(1 - e^{-kn/m}\right) = e^{k \ln\left(1-e^{-kn/m}\right)} = e^{-\frac{m}{n} \ln p \ln(1-p)},$$

and try to minimize the argument of the exponent. From considerations of symmetry, the minimum is at $p = 0.5$. Thus, the optimum is at $k = \frac{m}{n} \ln 2$, so that the probability of a false positive is $0.62^{m/n}$.

For example, suppose $m/n = 10$. If $k = 1$, then $P \approx 0.905$, so the false positive probability is 0.1. This is pretty close to the intuitive answer, actually.

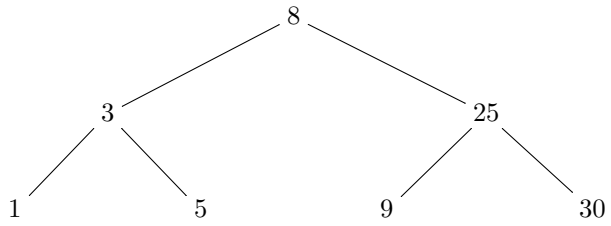
If $K = \frac{m}{n} \ln 2 \approx 7$, then the false positive probability is $0.5^k \approx 0.008$, which is much better! This is particularly noteworthy because it comes without an increase in space (though there's a tradeoff in that more hash functions have to be computed).

This is a simplification of the Bloom filter, since it doesn't even discuss how to remove or count elements in the set.

Another data structure that is worth considering is a binary search tree. Heaps support min/max and search in a nice running time, but in general there's no good way to find the predecessor or successor to a single element as in a (sorted) list.[13]

A binary search tree is a tree: a node x with two subtrees X and Y (left and right). The key property is that if $a \in X$, then $f(a) \leq f(x)$, and if $a \in Y$, then $f(a) \geq f(x)$.[14] This ordering sets it apart from the heap, and it also is not assumed to be balanced or (nearly) complete.

Example 10.1.



◂

. . . but it's just as possible for it to be greatly unbalanced.

In-order traversal takes linear time (set up the recurrence). But it can be sorted in $O(n)$ time, since the same algorithm as traversing it can be modified to inserting the element in that place.

Thus, constructing a binary searching tree takes at least $\Omega(n \log n)$ time (otherwise, you could sort arrays in less than $\Omega(n \log n)$ time). This is opposite to a heap: there, construction was easy but sorting took more time.

Searching is also very easy; it's basically optimized for binary search, since the left or right nodes hold all the values lesser than or greater than the root. This can be done in $O(\log n)$ time *if the tree is balanced* —

[13]Formally, the predecessor is the maximum value of the keys larger than a given one, and successor is defined similarly as the minimum of all keys larger than the given one.

[14]Often, it will be assumed that the entries are unique, to make analysis easier.

which is not true in general. But in general, the tree cannot be taller than n, so searching takes $O(n)$ time. (Specifically, the time is $\Theta(h)$, where h is the height of the tree.)

Insertion is much easier than deletion: search for the element, and then add it to one of the leaves (in some sense, the algorithm "falls off" the tree if it doesn't find it). This is also $\Theta(h)$, or $O(n)$, for the same reasons that searching does.

This allows sorting: insert the elements in order, then walk over the entries. The running time is $\Theta(n^2)$, since the tree can be very unbalanced (as when the input is already sorted, in which case the tree will look like a linked-list).

This sorting algorithm is directly related to Quicksort, even though it doesn't seem obvious. If this sorting algorithm is adjusted so that the input is randomly permuted before insertion, then the expected time is $O(n \log n)$, and, more interestingly, the same operations as Quicksort are performed (albeit in a different order): Quicksort looks at one element and then divides the rest into smaller and larger sets. This sorting algorithm compares an element individually with each.

11. Successor and Predecessor: 11/1/12

Similarly to searching, one can find the minimum and maximum by just following the left (resp. right) child of each successive node, starting at the root. This is $\Theta(h) = O(n)$ as well.

To find the successor, there are two cases:

- If the node in question has a right child, go to that child, then to the left until it's no longer possible. Return that node.
- Otherwise:

```
y = parent(x)
while y != NULL && x = right(y)
    x = y
    y = parent(x)
return y
```

This climbs up the tree until something is the left child of something else, at which point the successor was found.

If the target is the maximum, it has no successor (the `y == NULL` condition).

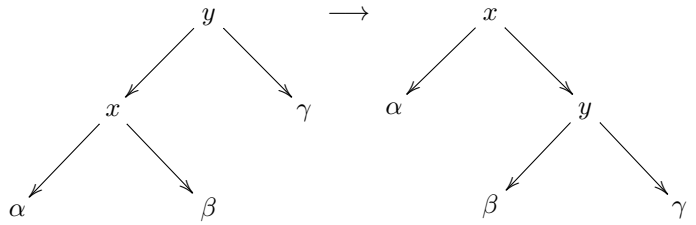
This operation is $\Theta(h)$ like the others — so if you can keep the tree balanced, everything will be very fast.

Predecessor is essentially the same; just reverse the arrows.

Deletion is a bit more interesting. There are three cases:

- If the node to be deleted has no children, replace the pointer to it from the parent with a `NULL`.
- If the node has one child, redirect the node from the node's parent to its child (as in a linked-list). This maintains the binary search tree property.
- In the third case, where there are two children, you can't just delete it, so replace the node with its successor. This preserves the binary search tree property, and the "hole" can be fixed with one of the previous cases (the successor won't have a left child).

This also depends on the height of the tree. Thus, it is important to be able to keep the tree balanced. This can be accomplished by "rotation," where the root is exchanged with one of its children. This involves some rearrangement:



(in which α, β, and γ can be subtrees). This takes constant time, since it just involves rearraging pointers.

A skip list is a structure of a linked-list with a couple extra pointers that is about as fast in expectation, and involves fewer cases in implementation (so that it is easier to implement).

Specifically, it is a sorted, doubly-linked list with "skip" pointers: in some sense, there is a bottom list, in which every element points to its successor. Every element is passed to the next list (list 1) with probability

1/2, so that the expected number of elements in list 1 is $n/2$. Then, list 2 is built on list 1 in the same process, and so on up to $2 \lg n$.

The extra pointers do take some more space, but memory is still $\Theta(n)$ in expectation. The probability that a node climbs to list $2 \lg n$ is $1/n$.

Searching is fairly simple: start at the top list, find the middle element, and decide whether to go to the left or right in the next lower level. It is the same intuition as binary search, and also takes $O(\log n)$ time.

However, deletion is more difficult, especially in the adversary scenario (delete just a few the promoted nodes, and suddenly binary search stops working).

So far, this class has dealt with some data structures and analysis. There hasn't been much insight into designing algorithms and providing tools to look at problems. One such tool is the idea of a greedy algorithm.

Example 11.1. Consider a set S of activities, such that each activity i starts at time s_i and ends at time f_i (e.g. lectures).

Items i and j are compatible if they do not intersect. (Here, the goal is to schedule the maximum number of lectures in a single lecture hall, though not the maximum lecture time). The goal is to find the maximum number of compatible activities.

The goal is not to find a specific solution, but a solution with a specific property (so there may be multiple solutions which work equally well). ◅

Look at the activity with the earliest finishing time f_k. It is always possible to take a legal solution and massage it so that there are at least as many activities scheduled and that activity k is included: if k doesn't intersect the earliest scheduled activity in the solution, then one can simply add it, and k cannot intersect more than one activity in the solution (since then one of them would finish before k), so substituting k for the earliest activity in the solution is valid.

This suggests a nice algorithm:

- Include the activity that finished first.
- Recurse on the set S' of activities that begin after k finishes.

More formally, let k have the smallest f_k and A be the optimum solution.

Case 1. $k \in A$.

> **Claim.** $A \setminus \{k\}$ is optimum for $S' = \{i \mid s_i > f_k\}$.
>
> *Proof.* Assume not; then, let B be optimal for S'. $|B| > |A| - 1$, but then adding k to B creates an optimum solution for S that is better than A. ⊠

Case 2. $k \notin A$. Then, the finshing time of the first job in A is after f_k, so it can be replaced with k and the solution is still optimal.

Most of the work in this algorithm is sorting: there is one sort and one scan through the activities, so the running time is $\Theta(n \log n)$.

12. Greedy Algorithms: 11/8/12

Consider a similar setup to the algorithm in the previous lecture, in which tasks are defined by a duration and a deadline (an excellent example is homework). The goal is to schedule tasks in a way that completes all the tasks before their respective deadlines, assuming such a solution exists.

The intuition is to complete the homework due soonest first, and this is in fact correct. Not all possible solutions have the first HW as the first task, but such a solution exists.

Claim. Assume that such a schedule exists. Then, there exists a schedule for which the first job is the one with the smallest deadline.

Proof. The trick is very similar to the previous example, using a cut-and-paste method to massage a schedule into one with the desired property.

Suppose a is the job scheduled first and b is the job with the soonest deadline. Let d_a be the deadline of a and d_b be that of b, so that $d_b \leq d_a$.

Then, putting b at the front is still legal: the homeworks done before d_b take the same amount of time and thus are still done before d_b (and therefore their respective deadlines), and the jobs after b are as before. ⊠

This gives an algorithm: sort the jobs in increasing order of deadline, and then do them in that order. As college tends to make painfully clear, a solution doesn't always exist, but this algorithm detects that; if it does not produce a solution, then no solution exists.

This algorithm takes the most time sorting, since producing the schedule afterwards is linear, so it takes $\Theta(n \log n)$ time.

In general, a greedy algorithm is an algorithm that makes a series of locally optimal decisions to produce a globally optimal solution. This also requires a partial solution to be extendable to a full solution. There may be many optimal solutions, but this method finds one of them.

Example 12.1 (Huffman Encoding)**.** The idea behind Huffman encoding is to represent more common characters with shorter codes. It is a prefix code: since it is a variable-length encoding, the code for one character cannot be the prefix for another, so that the decoder knows how to read the code.

This is done by building a binary tree for decoding, so that reading bits corresponds to walking down the tree until a letter is reached.

Assume that `a` is a very common symbol (which isn't that incorrect). One could produce a very unbalanced tree to generate the encoding, in which `a` maps to `0` and everything else has a longer code. Specifically, if one knows the symbol frequencies: if f_i is the frequency of symbol i and b_i is the length of the encoding of i, then the space used is $\sum_i f_i b_i$. Shannon's limit of information content is about 1.51, which is about as efficient as it is possible to be.

Generating the optimal encoding is a fairly simple greedy algorithm, based on the following claim:

Claim. Let x and y be the characters with the lowest frequency. Then, there is an optimum code in which they differ by only the last bit.

Proof. Consider any given encoding (not necessarily optimal) and let a and b be the deepest characters in the tree sharing a parent. In general, they may not be x and y. In particular, $f_x < f_a$ and $f_y < f_b$. Then, exchanging x and a and y and b will give a better outcome, since the amount of bits spent on x and y is less than that on a and b, so increasing the depth of x and y instead of a and b decreases the number of bits used on them overall.[15] ⊠

This leads to the following algorithm:

(1) Sort the characters by frequency.
(2) Glue the two least frequent characters x and y into a "super-character" with frequency $f_x + f_y$, and repeat until only one character remains.

This leads to an encoding tree, which is the optimal tree (though it is possible for encoding to be more efficient in non-Huffman methods; if there are a lot of two-character sequences, Huffman encoding is not optimal). ◂

Definition 12.2. A graph is a set V of nodes (vertices) and a set E of edges such that:

- Each edge connects a pair of nodes.
- It can have an associated direction (i.e. the pair is ordered), in which case the graph is called directed.
- Each edge can have a weight (i.e. a real number associated with it). This weight is completely dependent on context.

There are two common ways to represent a graph: in an adjacency matrix A, $a_{ij} = 1$ iff there is an edge between nodes i and j, and is 0 otherwise. For undirected graphs, this is somewhat of a waste of space, since the matrix is symmetric, but sometimes this is used nonetheless.

Alternatively, one can use an adjacency-list, so that every node has a linked-list of adjacent edges. This has different size concerns: if the graph is sparse, the linked-list representation is probably better, and for dense graphs, the matrix wins.

In order to determine whether a certain edge exists, it takes constant time for the matrix and linear time in the degree of the graph for the list. Thus, graph algorithms may have different running times based on which representation is used. This is somewhat offset by how easy it is to convert between the two, however.

Graphs have many different properties or characterizations: they may be directed or undirected, weighted or unweighted, sparse or dense, connected or not connected... see the book for more information.

[15] One can be more precise as to which is the minimum, but this is neither difficult nor particularly enlightening.

Definition 12.3. The graph with an edge between any two nodes is called the complete graph.

Definition 12.4. A forest is a graph without any cycles. A connected forest is called a tree.

There are many more graph definitions (e.g. strong connectedness), and many, many graph algorithms (such as finding cycles, minimal spanning trees, etc.). And many algorithms apply directly or indirectly to graphs, such as congestion management in traffic, analyzing communication networks, and (less obviously) things such as assigning interns to hospitals, scheduling jobs on a multiprocessor, searching solution spaces of some problems, etc.

In general, this involves restating a problem as a graph, solving the abstract graph problem, and then map it back. Often, this is as simple as opening a textbook, because there are a lot of solved graph problems.

Algorithm 12.5 (Depth-First Search)**.** This should be familiar to most people:

- Set all nodes to be white.
- For a given node, color it black, and then examine all of its children that haven't already been examined (i.e. are still white).

If there are n nodes and m edges, the running time is $O(m+n)$.

Edges can be classified by how they interact with a depth-first search:

- A tree edge points to an unexplored node at the time it is visited.
- A back edge points to a node that is still being explored when it is visited.
- A forward edge points to an ancestor of the node it comes from.
- A cross edge points to an already explored node which is not an ancestor of the node it comes from.

Forward and cross edges are distinguished by visitation time d; for a forward edge, the visitation time of the origin node must be less than that of the successor.

Theorem 12.6 (Parenthesis Theorem)**.** *For a node x, let $d(x)$ be the starting time for searching a given node, and $f(x)$ be the finishing time for that node. Then, for any two nodes u and v, then the time intervals $[d(u), f(u)]$ and $[d(v), f(v)]$ are either disjoint, or one is contained within the other (in which case the former is a descendant of the latter).*

Proof. Assume without loss of generality that $d(u) < d(v)$. If v was not discovered before $f(u)$, then the intervals are disjoint.

Otherwise, $f(v) < f(u)$, since v must be finished so that the recursive call of u must be returned. ⊠

13. The White-Path Lemma: 11/13/12

Consider the DFS algorithm from last lecture, and use the same notation as before.

Lemma 13.1 (White-Path)**.** *Suppose G is a graph and u and v are nodes of G. Then, v is a descendant of u iff at time $d(u)$ (i.e. the time when u is "discovered"), there is a path from u to v using only currently white (i.e. untraversed) nodes.*

Proof. Suppose v is a descendant of u. Let ww' be some edge on the path $u \to v$ in the tree. Notice that if ww' weren't white at time $d(u)$, then ww' couldn't be a tree edge. This is because if the node weren't unexplored, then the algorithm wouldn't explore it. Thus, all nodes on the path $u \to v$ are white at time $d(u)$.

Now the (harder) other direction: suppose there is a white path from u to v. Let ww' be the first edge on this path such that w is a descendant of u but w' is not (so that w' is closest to u).

In this case, $f(u) > f(w) > d(w) > d(u)$ by Theorem 12.6. But w' is also discovered after starting u and finishing w (since it was white at $d(u)$). Thus, $d(u) < d(w') < f(w) < f(u)$.

By Theorem 12.6 again, w' is also a descendant of u, which is a contradiction. ⊠

Lemma 13.2 (Simple Lemma)**.** *If G is undirected, then all edges must be tree edges or black edges.*

Proof. Consider edge uv, and without loss of generality, suppose $d(u) < d(v)$. Since uv exists, then v must be discovered and finished before finishing u. Then, either:

- If uv is discovered from u before v, then it is a tree edge.
- if uv is discovered from v before u, then it is a black edge. ⊠

Exercise 13.3. Why does the proof break down in the directed case?

Solution. In a directed graph, the DFS algorithm only considers outgoing edges, so edge uv has an orientation which means that it doesn't just matter whether the algorithm gets to u or v first. ⊠

Claim. If G is a directed graph, G is acyclic iff the DFS algorithm yields no black edges.[16]

Proof. Notice that a black edge implies a cycle, since it is an edge from a node to its ancestor.

In the reverse case, assume that there exists such a cycle. Let v be the node on the cycle with the smallest discovery time. Then, by construction, at time $d(v)$, all nodes in the cycle be white. In particular, let uv be the edge in the cycle that connects v (so that u points to v). Then, u is also white, so by the White-Path Lemma, all of the nodes in the cycle are descendants of v. In particular, u is both an ancestor and a descendant, so a black edge exists. ⊠

At this point it will be useful to introduce the notion of a partial ordering, in which some objects are comparable but not necessarily others. The easy example is inclusion of subsets: some objects are comparable and obey nice rules, but for general sets A, B it is not necessarily true that $A = B$, $A \subset B$, or $A \supset B$.

A partial ordering can be represented as an acyclic directed graph. Some relations are implicit (if $a < b$ and $b < c$, then $a < c$ but there isn't an edge between a and c, as it's not necessary to get this information). The real question is the topological sort: is it possible to create a total order compatible with the partial order? This has applications to greedy algorithms: one wants to find a schedule, given a set of dependencies in scheduling that satisfy a partial order. More than one solution might exist, which makes sense considering the previous examples for greedy algorithms. This has plenty of applications other than scheduling.

It turns out this is just a twist on DFS:

Algorithm 13.4 (Topological Sort)**.** Suppose G is a directed acyclic graph (so that it corresponds to a partial order). Then,

(1) Call DFS and compute the finishing time $f(v)$ for each node v.
(2) As each v is finished, insert it into the front of the linked list.
(3) Return the linked-list (in forward order).

Claim. The output list is a legal topological sort.

Proof. It is sufficient to prove that for all u, v such that uv is a (directed) edge, then $f(v) < f(u)$, so that the total ordering property is satisfied.

Suppose the DFS algorithm explored uv. Then, v cannot be gray. If v is white, then it is a descendant of u, so $f(v) < f(u)$, and if v is black, then it finished before u started, so $f(v) < f(u)$ again. ⊠

Another useful graph algorithm is that of the minimal-cost spanning tree. This has many applications (e.g. cable layout, generic optimization of lots of stuff, etc.).

This algorithm operates on a weighted graph, and attempts to find the tree that connects all nodes on the graph such that the sum of the weights on the tree nodes is minimal.

Definition 13.5. Suppose $G = (V, E)$ is an undirected graph with weights $w : E \to \mathbb{R}$. A minimal spanning tree (MST) is a tree $T \subseteq G$ such that the tree weight (the sum of the weights of the graph) is minimal.

For now, assume there are only single connected components, to avoid some complicated but uninteresting edge cases.

Algorithm 13.6. The solution is a greedy algorithm:

(1) Choose the edge with the smallest weight, and glue its nodes together into one "supernode."
(2) Then, choose the smallest edge going out of this node, and contract that edge as well, and so on.

Claim. If G is a connected graph and T is a minimal spanning tree of G and $A \subset T$ is a subtree. Let uv be the edge of weight connecting A to $T \setminus A$ (i.e. the graph induced by the edges in T but not A); then, there exists an MST T' such that $A \cup \{uv\} \subseteq T'$. Thus, this edge is not necessarily in every optimal solution, but in at least one of them.

[16] Cycles in undirected graphs certainly exist, but there one must tread with a bit more care.

The assumption for this greedy algorithm is that the graph is connected (so that a minimal-cost spanning tree actually exists), and this assumption is implicit in the proof. Another important ingredient in the cut-and-paste step is that (as in all greedy algorithms) there may be many optimal solutions, and this algorithm finds just one of them.

Algorithm 14.1 (Prim's Algorithm)**.** The main idea is to:

(1) Choose a node v and let $A = \{v\}$.
(2) Loop over the following:
    (a) Find the minimum-weight edge e going out of A, and
    (b) Add e (and therefore implicitly the node it points to as well) to A.

This algorithm requires one to be able to find outgoing nodes from a given node, and to compare the minimum-weights.

More than one possible implementation exists. One could keep all the edges (both outgoing and internal) of A into a heap. To get the next edge, extract the minimum-weight node from the heap and check if it's outgoing (i.e. whatever it points to isn't in A) and if it's internal, discard and repeat. To add a new node, add all of its edges to the heap.

The running time of this is based on the heap. If there are V nodes and E edges, then there are $O(E)$ insertions and $O(E)$ deletions from the heap, and since there are no parallel edges, then the running time is $O(E \log V)$. Note that this assumes the linked-list representation rather than the adjacency matrix; if otherwise, it takes $O(n^2)$ to convert between them.

But this is wasteful; only $V - 1$ edges were used, and the rest were wasted. Thus, one could keep nodes in the heap, rather than edges, storing the distance of the node from A from a single edge (using some large number for infinity).

Initially, let the distance from the root be 0 (where the node is randomly chosen) and the key (i.e. distance) of every other node be infinity. Then, starting from the root, update the keys of every node with the minimum distance to it, and repeatedly add the minimum node to A. Here, the running time is $O(E)$ to decrease the keys, and $O(V)$ for extracting the minimum.

	extract-min	decrease-key	Total
array	$O(V)$	$O(1)$	$O(V^2)$
heap	$O(\log V)$	$O(\log V)$	$O(E \log V)$
Fibonacci heap	$O(\log V)$	$O(1)$	$O(V \log V - E)$

Table 1. Running-times for different implementations of this algorithm. (Note that the Fibonacci heap hasn't been discussed in this class, but it's a very useful way of approaching this problem.)

There's a completely different way to go about this:

Algorithm 14.2 (Kruskal)**.** Loop over the following:

(1) Scan the edges in increasing order of weight.
(2) Add the current edge if it doesn't close a cycle among the edges already in the tree.

This algorithm, unlike the previous one, makes no claims about connectedness of its list of edges until the very end.

Notice that the minimum-weight edge (or one of them) is always present in an MST. More strongly, for any minimum-weight edge, there is an MST that contains it.

Why does this work? Consider the instant where the first wrong edge is added (i.e. an edge xy is added that isn't part of any optimum tree). The optimal solution has a path from x to y (since it's a spanning tree) that doesn't include xy. If this path contains edges of weights less than that of xy, then xy isn't considered by the algorithm, so they must be greater. But this is a contradiction, since xy is non-optimal.

Consider the implementation. In order to implement Kruskal's algorithm, one need to know if two nodes u and v are in the same connected component. If one treats the connected components as sets, then one needs the ability to detect if something is in a set (specifically, find the set that contains a given node) and construct

the union of two sets. In particular, one needs to make $O(V)$ sets and find $O(E)$ of them. Additionally, there are $O(V)$ unions and the initial sorting algorithm, the latter of which can be worried about later.

So it turns out union-find takes running time inverse to the Ackermann function (i.e. $O(\alpha(E, V))$, where α is the inverse Ackermann function), which is *much* faster than logarithmic time. This is accomplished with a union-find data structure.

One might consider maintaining every set as a linked-list, and every element points to the head of the list. Sets are merged by manipulating pointers, and the ID of the set is that of the list head.

Then, finding the set of a given node is $O(1)$, and union is constant per element of the list.Thus, there are $O(V)$ changes per element for all unions, so the total work put into the unions is $O(V^2)$.

This implementation is pretty bad, but there's a quick fix: merging smaller lists into larger ones. This makes the surprisingly large difference that there are $O(\log V)$ charges per element for all unions, making the overall work for the unions $O(V \log V)$ giving a total running time of $O(E + V \log V)$ plus sorting. (Using the union-find structure, one can get a running time of $O(E + V\alpha(E, V))$ plus sorting.)

15. Dynamic Programming: 11/27/12

The goal is to prove this very non-obvious claim:

Claim. If all the weights on a weighted graph are different, then the minimal spanning tree is unique.

Proof. Assume that there exists an MST T that is different from the tree given by Kruskal's algorithm, and follow Kruskal's algorithm until the first disagreement between the two. Here, Kruskal adds an edge $e \notin T$.

If e is added to T, then it forms a cycle. One can look at the cycle and remove an edge to get another tree T' again. But one of the edges in the cycle is heavier than e, or Kruskal would have added it before e. Thus, if T' is made by deleting the heaviest edge, it is distinct from T and better, so T is not optimum, which is a contradiction. ⊠

And now for something completely different: dynamic programming.

The major issue with greedy approaches is that sometimes it isn't possible to partially commit to a solution up front. There may be many choices, but it's not clear that there is a good first choice. As with greedy algorithms, this is more of a concept or technique than a specific algorithm, in which one tries to solve lots of smaller sub-problems in order to combine them into a larger solution. Unlike the divide-and-conquer recursive approach, this is based on combining solutions.

Example 15.1. Consider a line with nodes $v_1, v_2 \ldots, v_m$, such that each node v_i has an associated positive weight w_i. The goal is to choose a subset of nodes such that the total weight of the chosen nodes is maximized but that no two adjacent nodes are both present in the set.

For simplicity, return the maximum value, rather than the nodes that yield it.

For example, in the set $\{5, 2, 8, 1, 1, 6, 3, 9\}$, return $5 + 8 + 6 + 9 = 28$. ◂

The greedy approach is wrong here because the solution requires looking ahead rather than just considering the first k nodes.

The main trick in dynamic programming is appropriately choosing the subproblems, so let W_i be the maximum weight if one only considers nodes 1 to i. $W_0 = 0$, $W_1 = w_1$, etc. In particular, $W_{i+1} = \max\{W_i, w_{i+1} + W_{i-1}\}$. Thus, it is easy to compute W_i for any given i, and the running time is $\Theta(n)$.

This is not greedy because one node might be ignored in one step (as W_i) and then taken in the next (as (W_{i-1}).

A classic problem that can also be approached with dynamic programming is the longest common subsequence problem. Consider two sequences (a bunch of objects with an ordering) x and y, and let $|x| = m$ and $|y| = n$. A subsequence in this problem is a sequences with the same ordering but with some elements removed (as in analysis, though here the sequences are finite). The goal is to find the longest sequence that is a subsequence of both x and y.

A greedy approach again doesn't work, because you can't just look at things in sequence. A brute-force approach considering all substrings of x and Y has running time of $O(2^{m+n}(m + n))$. It might be better to take every substring in x and check it against y, which has a better running time of $O(2^m n)$.

Here's a pictoral explanation of the main dynamic step: if AB is compared with BDC, then the longest common substring is B. But if C is added to the first string, there is a C later in the second one, so it can be added to the subsequence.

Formally, define $C(i,j)$ to be the longest common subsequence among $x_1,\dots,x_i$ and $y_1,\dots,y_j$ (i.e. taking the first i entries in x and the first j in y). The answer to the question in general is $C(m,n)$, and that $C(1,1)$ is very easy to compute (depending on whether the first two characters are equal). Then,

Theorem 15.2.

$$C(i,j) = \begin{cases} C(i-1,j-1)+1, & x_i = y_j, \\ \max\{C(i,j-1), C(i-1,j)\}, & \textit{otherwise.} \end{cases}$$

Proof. Divide the proof into the two cases above:

Case 1. $x_i = y_j$.

Define $z = z_1,\dots,z_k$ to be the longest common subsequence of $x_1,\dots,x_i$ and $y_1,\dots,y_j$. If $z_k \neq x_i$, then z isn't an LCS.

Claim. $z_1,\dots,z_{k-1}$ is the LCS of $x_1,\dots,x_{i-1}$ and $y_1,\dots,y_{j-1}$.

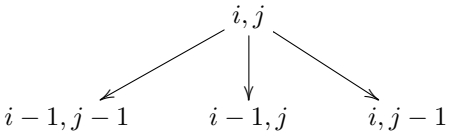
Proof. If there exists a subsequence longer than $z_1,\dots,z_{k-1}$, then append z_k to it, which creates a contradiction. ⊠

Case 2. $x_i \neq y_j$.

There are several sub-cases which are mostly symmetric:

- If $z_k = x_i$, then $z_k \neq y_j$, so $z_1,\dots,z_k$ is an LCS of $x_1,\dots,x_i$ and $y_1,\dots,y_{j-1}$.
- If $z_k = y_j$, then the same thing happens, just switching x and y.
- Otherwise, $z_k \neq x_i$ and $z_k \neq y_j$, so the LCS is the same as one of the previous ones. ⊠

This theorem yields an obvious algorithm for computing $C(m,n)$ based on the recursive formula outlined in the theorem. One can set up a tree of the recursive calls. Unfortunately, the depth of the tree is $O(m+n)$, so the bound is $O(3^{m+n})$ (or $O(2^{m+n})$ if one is careful), which is too large. The tree for one call and its subtrees is as follows:



Notice that the number of subproblems increases more than one would like, which is a bad sign in dynamic programming. However, there are plenty of repeated subproblems, so it might be possible to optimize this strategy:

- Memoization: every time a $C(i,j)$ is computed, store it in a matrix and reuse it when the problem is asked again. This is nice, but also a bit harder to explain.
- The dynamic approach, which is just to compute the table from the bottom-up rather than from the top down. The table of $C(i,j)$ can be computed from the top-left corner and going row-by-row. This has the advantage that the entries that are called by the theorem are already computed.

The second approach takes running time of $\Theta(mn)$, which is much nicer. More interestingly, the algorithm is very easy to implement, which is also nice. And you can get the subsequence by adding a flag that represents the trace-back.

16. More Dynamic Programming Examples: 11/29/12

This example is not as useful on its own, but it illustrates a different approach to dynamic programming that can be useful. Consider a set of matrices $A_1,\dots,A_n$ such that A_i is a $p_{i-1} \times p_i$ matrix (so that the product $\prod_{i=1}^{n} A_i$ is defined). The time it takes to multiply matrices A_1 and A_2 is $O(p_0 p_1 p_2)$, and if there are three or more matrices, then there are several options. For example, if A_1 is a 5×100 matrix, A_2 is 100×2, and A_3 is 2×50, then:

- Computing $A_1(A_2A_3)$ requires $(100)(2)(50) + (5)(100)(50) = 35000$ operations.
- Computing $(A_1A_2)A_3$ requires only $(5)(100)(2) + (5)(2)(50) = 1500$ operations.

So for the general case, what is the optimum order for multiplication? Remember that matrix multiplication is associative but not commutative.

A greedy algorithm doesn't work, but a dynamic programming approach requires a subproblem. Consider the time it takes to multiply matrices A_1 to A_i, and consider the last optimum multiplication

$(A_1 \cdots A_k)(A_{k+1} \cdots A_n)$. That such a k exists is the main trick. Then, both $\prod_{i=1}^{k} A_i$ and $\prod_{i=k+1}^{n} A_i$ were also computed optimally.

Thus, let $m(i,j)$ be the optimum time to multiply $A_i \cdots A_j$, which is the subproblem. Then,

$$m(i,j) = \begin{cases} 0, & i = j \\ \min\limits_{i \le k \le j} \{m(i,k) + m(k+1,j) + p_{i-1}p_k p_j\}, & i < j, \end{cases}$$

and the answer to the whole problem is $m(1,n)$. The recurrence relation is

$$T(n) \ge 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \ge 2 \sum_{k=1}^{n-1} T(k) + n$$

This is exponential, unfortunately: $T(n) \ge 2^{n-1}$. But the power of dynamic programming is to save the results, either by memoization or by proceeding in a way that avoids recalculating information. Since there are only $O(n^2)$ subproblems, this is much nicer. Thus, build the table up in order of increasing row index i, so that the necessary information is already calculated. For each $m(i,j)$, it is $O(n)$, so the total time is $O(n^3)$. $\Omega(n^3)$ is a bit harder to show, since the time depends on $i - j$. At least $n^2/16$ take at least $n/4$ time to compute (the top fourth of the j and the bottom fourth of the i), so this is $\Omega(n^3)$ once the constants fall out.

Another problem that can be answered with dynamic programming is the subset sum problem. Suppose $x_1, \ldots, x_n, B \in \mathbb{N}$; then, the goal is to determine whether there is a subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} x_i = B$.

There is actually a nice brute-force approach:[17] consider the set S as a bit vector, and check each possible bit vector, which has running time $O(n2^n)$. A greedy algorithm wouldn't work, because there's no guarantee that an optimal solution includes the least element.

Unfortunately, this problem is NP-hard. But consider the subproblem $M(k,b)$ is "true" if $\sum_{i \in S'} x_i = b$, for $S' \subseteq \{1, \ldots, k\}$. Then, the total problem is $M(n,B)$. Also, notice that $M(1,b)$ is true if $x_1 = b$ and is false otherwise.

The recurrence is $M(k_1,b) = M(k,b)$ or $M(k, b - x_{k+1})$, depending on whether x_{k+1} is used. Since M is an $n \times B$ matrix, and each element takes constant time, the running time is $O(nB)$.

But wasn't it NP-hard? This still works, because the size of the problem is $O(n \log B)$ (in terms of the number of bits required), so this running time is still exponential. However, this is better than the brute-force approach, particularly if B is small relative to n.

The next problem to consider is the knapsack problem, in which there are n items, such that each item i has a cost $v(i)$ and a weight $w(i)$. The cost and weight are unrelated. The goal can be stated in terms of a thief who is robbing a store — he can hold at most W weight, but would like to maximize the value of the items he puts into his knapsack given this restriction. This is partcularly apt for solving real problems in which utility is maximized while some sort of cost ceiling is imposted.

If one is allowed to take parts of an item (the fractional case), the solution is easy! In this case, there is a greedy solution operating on the best value-per-weight element. The proof is by "cut-and-paste:" replace an amount of a cheaper substance by that amount of a more expensive one. Obviously, though, the fractional case is not always useful. What would you do with 1.3 dishwashers?

The greedy algorithm doesn't work in the integer case: if item 1 has weight 20 and cost \$30, item 2 has weight 50 and cost \$60, and item 3 has weight 50 and cost 50. If the total weight is 100, then the greedy algorithm chooses 1 and 2, which has value \$90. The optimum solution is 2 and 3, with value \$110. However, the fractional case is even better: 1, 2, and 3/5 of 3 gives \$120; in general, the fractional case is at least as good because there are more options to choose from. Thus, the fractional solution can be used as an upper bound on the integer solution. Additionally, the optimum weight may be strictly less than W.

Notice that if the highest-index element (where the goods are ordered) is removed form the knapsack, the remaining items are a solution for a smaller knapsack; specifically, if the optimum solution is $x_1, \ldots, x_k$, then $x_1, \ldots, x_{k-1}$ is optimum for $W - w(x_k)$. If $C(i,w)$ is the optimal solution using items 1 to i in a knapsack that can hold weight w. Then,

$$C(i,w) = \begin{cases} 0, & i = 0 \text{ or } w = 0 \\ \max\{v_i + C(i-1, w-w_i), C(i-1,w)\}, & \text{otherwise,} \end{cases}$$

[17] In practice, brute-force is a great way to start, often paired with a data structure or sorting. Then, one can proceed to a greedy or dynamic solution.

depending on whether the i^{th} element is taken. Then, the table size is nW and it takes constant time per element, so the total is $O(nW)$. Again, however, since the problem size is $O(n \log W)$, then this is exponential, yet still useful in some cases. In fact, this problem is also NP-hard.

To summarize dynamic programming:

- A good first step is to analyze the optimum substructure.
- The crucial step is to define the subproblems:
  - Make sure there aren't too many subproblems.
  - Solving the subproblems must lead to a solution to the original problem.
  - There should be a way to solve larger problems in terms of smaller ones, leading to a recurrence.
- Then, organize the subproblems into a table, so that one doesn't need to calculate the same thing many times.

17. Single-Source Shortest Paths: 12/4/12

Consider a directed graph $G = (V, E)$ with n nodes and m edges. The edge uv has a weight $w(uv)$, thinking of the weight as a function $w : E \to \mathbb{R}$.[18] This time, though, there is also a distinguished node s, called the source.

What is the shortest path from s to all nodes that are reachable from s? And in particular, how much space is necessary? It takes $O(n^2)$ space to store the paths, but $n - 1$ distances if that is all that is necessary. However, the paths form a tree, which makes encoding easier.

The main observation is as follows, and should look somewhat similar to past optimum solutions. Suppose the shortest path from s to v goes through some node u; then, the section of that path from s to u must be the shortest path from s to u. If this weren't the case, then you could swap it with the shortest path and get a shorter path from s to v. This is still true even if uv is a single edge rather than a longer path.

Thus, for every node v, one needs to recall a node u such that the shortest path from s to v contains the edge uv. This results in a tree, since there cannot be cycles, which is nice, because negative-weight cycles do odd things to distance. However, it isn't always true that the shortest such path is unique.

So really, we already have an algorithm. If the weights are all positive, Dijkstra's algorithm, a greedy algorithm, can be used (as will be shown in the next lecture), but in the general case, dynamic programming will be needed.

Algorithm 17.1 (Bellman-Ford)**.** Since the problem calls for dynamic programming, consider the subproblem $d^k(v)$, which is the distance from s to v over all paths with at most k edges. Then, to reach v in using most $k + 1$ edges, one must reach a neighbor u of v using k edges, and then jump from u to v, or just reach v in k hops. This yields the following recurrence:

$$d^k(v) = \min\{d^k(v), \min\{d^k(u) + w(uv) \mid uv \in E\}\}.$$

Without thinking too much, this is $O(m)$ per phase (i.e. each value of k). But it's possible to do better: since each edge into v is looked at only once over the course of the whole algorithm, then the total running time is $O(m)$, the number of edges.

Start with $d^k(v) = \infty$, and think of the algorithm as receiving updated distances from its neighbors. At every iteration, each node "asks" its neighbors for their values, processes, and then updates its own value as a result. One can terminate the algorithm after phase k if $d^k(v) = d^{k-1}(v)$ for all $v \in V$, since there will be no further changes after that. Even if this doesn't happen, the algorithm still only needs to run at most $n - 1$ times as long as there are no negative cycles, since $d^k(v)$ never increases with k, and decreases with each update. If $d^k(v)$ is updated when $k > n - 1$, then $d^k(v) < d^{n-1}(v)$, so the path cannot be simple, so there is a negative-cost cycle.

The total running time of the algorithm is $O(mn)$.

This sort of algorithm is very useful in networking protocols and such.

If there is a negative-weight cycle, the algorithm doesn't terminate at all. This is intuitively true, but it deserves a formal proof:

[18] Note that negative weights are allowed, and sometimes one might want an infinite value. The latter can be represented by 1 greater than the largest value. This is much easier algorithm-wise than working with actual infinities.

Proof. Suppose the nodes along the negative-cost cycle are numbered $v_1, \dots, v_r$. If the algorithm terminates, then for all edges $v_i v_{i+1}$ along the cycle, $d(v_{i+1}) \le d(v_i) + w(v_i v_{i+1})$ (or otherwise, it would be updated). Then (using slightly sloppy notation),

$$\sum_i d(v_{i+1}) \le \sum_i d(v_i) + \sum_{i=1}^{r} w(v_i v_{i+1}),$$

so the cycle has positive weight. ⊠

Thus, it is possible to detect the existence of negative cycles by running over $n-1$ iterations and detecting whether the algorithm terminates.

Another concern is whether all cycles are reachable from s. The algorithm can be modified to deal with them, such as adding a node with edges of cost 0 to every other node, which allows for every path to be reachable, creates no new negative cycles, and thus can be used to detect negative-cost cycles throughout the graph. This allows one to avoid wasting time on such infinite cycles.

18. All-Pairs Shortest Paths: 12/6/12

An extension of the algorithm from the previous lecture is to try to find the shortest paths from all possible starting nodes to all possible ending nodes. Using Bellman-Ford on each source as above, there are n sources, each of which takes $O(mn)$ for its call, so the total is $O(n^2 m)$. This is pretty bad in the case of a dense graph ($m = \Theta(n^2)$), since the total is $O(n^4)$.

Another possibility is to make a matrix W of weights (such that the weight between two unconnected edges is infinity, and there are zeros on the diagonal). Then, define L_{ij}^k to be the distance from node i to node j in at most k hops. Then,

$$\begin{aligned} L_{ij}^{k+1} &= \min(L_{ij}^k, \min_{1 \le v \le n} (L_{iv}^k + w_{vj})) \\ &= \min_{1 \le v \le n} (L_{iv}^k + w_{vj}), \text{ since } w_{jj} = 0, \end{aligned}$$

where v are only the neighbors of j, which is important to remember when implementing.

This is still $O(n^4)$, since for every k it is necessary to compute an $n \times n$ matrix. However, this resembles matrix multiplication, except calculating the minimum rather than adding and using addition instead of multiplication.[19]

Let k be the smallest natural number such that $p = 2^k > n$. Then, W^p gives all pairs of shortest paths (since after that, nothing is updated). Again, this is $O(n^3 p) = O(n^4)$. But p can be computed by repeated squaring, so $W^k W^k = W^{2k}$, which has better running time of $O(n^3 \log n)$.

There is, however, another approach:

Algorithm 18.1 (Floyd-Marshall)**.** Consider the different subproblem $L_{ij}^k$to be the shortest path distance from i to j using only the intermediate nodes 1 through k. When one adds another intermediate node, then either it helps (in that the path uses it), or it doesn't. Formally,

$$L_{ij}^k = \begin{cases} w_{ij}, & k = 0 \\ \min(L_{ij}^{k-1}, L_{ik}^{k-1} + L_{kj}^{k-1}), & k > 0. \end{cases}$$

This has time $\Theta(n^3)$.

Returning to the single-source shortest path, it's possible to do much better than Bellman-Ford if it is known that there are no negative-weight edges. This is very common in applications (weights of a network, distances on a map, etc.).

Algorithm 18.2 (Dijkstra)**.** Start with $d(s) = 0$ for all $v \ne s$ and $d(v) = \infty$. Then, put everything into a heap, with the key value equal to the distance. Then, while the heap is not empty:

- Let u be the minimum element, extracted from the heap.
- For every v adjacent to u, let $d(v) = \min(d(v), d(u) + w(uv))$.

[19] Technically, this structure is a semiring; there is both addition and multiplication, but there are no additive inverses. An excellent example is $\mathbb{N} \cup \{0\}$ with $+$ and $\times$. In a sense, you can add and multiply, but there is neither subtraction nor division. If one adds additive inverses, a ring is obtained, and if division is also possible, then the set is called a field.

This algorithm adds the node with the shortest perceived distance. In particular, it looks like Prim's algorithm for a minimal-cost spanning tree. The running time is $O(m \log n)$, though if you use a Fibonacci heap, this is $O(m + n \log n)$, which is sometimes better. In either case, this is much better than Bellman-Ford!

Proof of correctness. For this proof, let $d(x)$ be the value computed for edge x, and $d(x, y)$ be the actual distance.

Let u be the first extracted node such that $d(u)$ is not equal to its distance (i.e. u is the first mistake). All distances are nonnegative and $d(v) \geq d(s, v)$ in all cases, so $d(u) > d(s, u)$.

Consider the shortest path from s to u, and in particular the edge xy where x was extracted already (so that it has the correct distance) and y is not yet extracted. Then, $d(y) \leq d(x) + w(xy)$, so

$$\begin{aligned} d(u) > d(s, u) &= d(s, x) + w(xy) + d(y, u) \\ &= d(x + w(xy) + d(y, u) \\ &\geq d(y + d(y, u) \\ &\geq d(y). \end{aligned}$$

Thus, $d(u)$ is currently not the minimum, so u is not extracted here. ⊠