CS 2413 — Informal Recap

Greg Aloupis
greg.aloupis@nyu.edu

This isn't a substitute for a textbook or the class notes. It is a set of explanations that might be of assistance while you are reading the class notes or the book. I'm trying to highlight what the more important concepts are.

This is an ongoing draft. It was written while I taught algorithms at Tufts. Please feel free to report bugs and unclear descriptions.

Contents:

Big-O
Recurrences.
Comparison-based sorting (mainly heap sort)
Decision trees and the lower bound for comparison sorting
Counting sort and radix sort
Indicator random variables
Selection (deterministic)
Selection (randomized)
Quicksort (randomized analysis)
Alternate Quicksort analysis
Binary search trees
Worst, best, and expected cost of constructing random BST
Red-black trees
Applications of dynamic BSTs (e.g. range counting and interval trees)
Dynamic programming
Hashing
Amortization
BFS and DFS
Topological sort, and strongly connected components
Minimum spanning trees (intro)
Kruskal's MST algorithm
Prim's MST algorithm
Single source shortest paths intro
The Bellman-Ford SSSP algorithm
Dijkstra's SSSP algorithm
NP-completeness

Sections to add: Huffman codes, Linear time algorithm for 2D LP, Intro to network flow

Big-O

You should understand what $f(n) = O(g(n))$ means mathematically. Same for Ω and Θ. When analyzing functions asymptotically, we basically don't care about certain constants: mainly (but not only) additive constants and leading multiplicative constants. For example if you're given a quadratic function, at this level we don't care if it is $5n^2$ or $37n^2$, and we don't care if there are less significant terms added on (for instance something linear, like $37n^2 + 83n$). Asymptotically, all these functions are indistinguishable. This justifies having the constant in the mathematical definition of big-O. For example, $132n = O(n^2)$. Remember that the big-O can be thought of as a $\leq$. In fact, by definition we have $132n \leq c \cdot n^2$. The c is there, because we don't care which quadratic function we're comparing $132n$ to. As long as it is smaller than *some* quadratic function, we're happy. The intuition is that $132n$ is not more significant than the class of quadratic functions.

So, you should know the definition and what it means in terms of comparing functions. You should know what it takes to compare two functions asymptotically. The main recipe is to set up the mathematical definition, and then find some combination of constants that makes it work. Always remember that the definition is meant to hold for a range of *large enough n*. It is your job to decide what range this will be (i.e., pick some n_0), and establish that there is a constant (c, as used above) that works for the entire range $n > n_0$. Getting the constant c to be small is a secondary goal. For lower bounds (using Ω), the secondary goal is instead to get the constant as large as possible. Remember that you have to use a different constant for the upper and lower bounds, if you're trying to get Θ. Ideally they will actually end up being the same number. Sometimes we can't achieve that.

It is critical that you understand how to compare functions asymptotically, even if informally. We will be doing this for the rest of the course.

Finally, I will mention a convenient way of showing that $f(n) = O(g(n))$.

Exaggerate and simplify. Remember, $g(n)$ is supposed to be a simpler function than $f(n)$. Also, we are upper-bounding $f(n)$, so it's ok to exaggerate. For example:
$f(n) = 223n^2 + 10^6 \log^3 n - 14$. We want to show that this is $O(n^2)$. We can exaggerate $f(n)$ by adding 14, and by replacing each $\log^3 n$ term with n^2, thus obtaining a function $f'(n) \geq f(n)$, for large enough values of n. This range is determined by comparing the replaced terms with the new ones. For example, for small n if we replace $\log^3 n$ with n^2 we are not exaggerating. In any case, $f'(n) = (223 + 10^6)n^2$, and this is $O(n^2)$ by definition. Thus, for large enough n we have: $f(n) \leq f'(n) = O(n^2)$.
For a matching lower bound, we could subtract all the $\log^3 n$ terms from $f(n)$, and replace the -14 with $-14n^2$. In both cases we make $f(n)$ smaller, thus obtaining $f(n) \geq f''(n) = 223n^2 - 14n^2 = \Omega(n^2)$. Therefore $f(n) = \Omega(n^2)$.

Recurrences

Recurrences will be popping up frequently in the course. They are often critical for understanding how good or how bad algorithms are. All we care about is getting asymptotic results, so in some sense this topic is easier than what you may have seen in a discrete math course, where more attention is given to finding all constants.

You need to be able to realize when an algorithm is recursive, and then how to write down the time complexity, $T(n)$, in a recursive way. Once you have a recursive relation, you should be aware that there are three main ways of solving it.

A typical recurrence will be in the form $T(n) = a_1 \cdot T(\frac{n}{b_1}) + a_2 \cdot T(\frac{n}{b_2}) + \ldots + f(n)$. In other words, there will be a non-recursive part, $f(n)$, and a collection of recursive calls. These might not all have the same sub-problem sizes, hence the various b_i. In algorithms, any a_i value will be a positive integer. We don't half-recurse. Given this fact, any b value must be greater than 1, otherwise the problems get bigger and bigger, and you'll have an infinite amount of work to do.

Recursion trees are typically described as being a tool just for providing intuition, although as they can give a pretty precise and convincing answer. When drawing a recursion tree, you always start with the non-recursive $f(n)$ at the root, and you assign a set of children; one per recursive call. Each such child will have the form $T(\frac{n}{b_i})$, for some value b_i. Once you've done this, it's time to replace each such term with a non-recursive amount of work. This work will be $f(\frac{n}{b_i})$. What you're doing is simply rewriting your initial recurrence relation for $T(n)$, using a different parameter. In other words, $\frac{n}{b}$ is your new n. Thus each current child becomes the root of a new recursion tree. It will have just as many children as its parent, and so on. However, all of this is only true while we have not encountered any base cases. The easiest kind of trees will be fully balanced, resulting from "nice" recurrences where there is only one value b. In other words there is only one sub-problem size to deal with. This makes every path from the root end at a base case after the same number of levels. That will not always be true.

The point of drawing the tree is to somehow count the total amount of non-recursive work. Typically this is done by counting the amount of work in each level, and finding out how many levels there are. By noticing a pattern in the amount of work involved for a few levels at the top, one is hopefully able to find an expression for the amount of work, as a function of the level number.[1] For a full (symmetric) tree, the total amount of work will be a summation over all levels, although to be a bit more formal you also add the number of base cases (i.e., leaves in the tree). If the recursion tree is not symmetric, then there will be base cases on various levels. A typical way to deal with this is to obtain an upper bound and lower bound separately. For an upper bound, we assume that all levels are full. In other words, we ignore the fact that some base cases have occurred, and let our expression for the amount of work per level be summed over all levels (whereas in reality the expression degenerates below the lowest full level). For a lower bound, we only compute the sum of work-per-level over all full levels, i.e., the ones that don't have base cases. In other words, both the upper and lower bounds are calculated on full trees, that are respectively an exaggeration and an underestimate of the true tree.

[1] To make this approach formal, one would have to prove that the expression will hold over all levels.

The second method for solving recurrences is the **substitution method**. There is a standard recipe for this. First you have to *guess* a solution. This is done either by intuition, or by getting a hint from a recursion tree, or, of course, you could actually guess. When you guess a solution, if it is expressed asymptotically, you must instead express it precisely, using the constant in the mathematical definition of O or Ω. For instance, if you've guessed $T(n) = O(n^2)$, you must express this as $T(n) \leq c \cdot n^2$. This is critical. It is also critical to use specific constants for the non-recursive part of $T(n)$ (meaning, $f(n)$). For example, if the non-recursive part is $\Theta(n^3)$, you have to state that this really means $d \cdot n^3$, for some positive constant d.

The second step is to use induction: assume that your guess is valid for all $k < n$. That means that if you write the recurrence with $T(k)$ instead of $T(n)$, you have the right to assume that your guess holds for $T(k)$. As an example, if you have $T(n) = T(n-5) + n^2$, and you've guessed that $T(n) = O(n^3)$, which really means you've guessed $T(n) \leq c \cdot n^3$, then for any $k < n$ you can assume that $T(k) \leq c \cdot k^3$. The only reason you're doing this is so that you don't have to deal with the recursive parts of the relation. You get rid of them, because each one has a parameter that is smaller than n, fitting the description of k. In the previous example, $n{-}5$ is smaller than n, so we can assume that $T(n{-}5) \leq c \cdot (n{-}5)^3$.

By now you should be able to express $T(n)$ in a way that has no big-O notation, and no recursive parts. If your guess involves a $\leq$ or a $\geq$ (for instance because you're just focusing on getting an upper or lower bound), that will work its way into the original recurrence when you make the substitution. What I mean is even if you had $T(n) = \cdots$, you may get an inequality after substituting. The rest is just arithmetic and algebra, not to imply that it is always trivial. The objective is to establish that your guess is true, for n. So far you've assumed that it's true for all $k < n$. It is crucial that you get the non-asymptotic guess *exactly* right. So, whatever you end up with for $T(n)$ must be separated into two parts: the first part is exactly what your guess was; the second part is all the leftovers. If you are trying to prove an upper bound, then these leftovers must be zero or less. If you're trying to prove a lower bound, they must be zero or more. Abstractly, what's going on here (using the upper bound as an example) is that you have made a guess in the form $T < X$, and you end up getting something like $T < X + Y$. So you have to be able to claim that Y is non-positive in order to actually have $T < X$. The great thing is, you get to affect these leftovers by your initial choice of constants (like d for the non-asymptotic part of $T(n)$, and c when you made your guess). Your goal is to assign some combination of values that makes the leftovers lean in your favor. These constants are your friends. To recap, the recipe is:

- guess a non-asymptotic solution for $T(n)$ (i.e., use a constant, not big-O)
- express the non-recursive part of $T(n)$, non-asymptotically (i.e. use a constant)
- assume your guess is true for $k < n$
- for any recursive call in $T(n)$, substitute in your guessed solution.
- by now you have $T(n)$ without any recursive or asymptotic parts, so extract the format of your guess, and collect all other leftover terms.
- make the leftover terms non-positive or non-negative, depending on the bound you're aiming for. To do this, assign any value you need to the constants you've defined.

The **master method** deals only with recurrences of the form $T(n) = aT(\frac{n}{b}) + f(n)$. The recursion tree is complete (symmetric). In that tree, every node branches out a times, so the number of nodes in level j is a^j. Also, every subproblem is a factor b smaller than its predecessor, which means that there are $\log_b n$ levels, and also means that the amount of work done at any node in level j is $f(\frac{n}{b^j})$. Thus the amount of work in level j is $a^j \cdot f(\frac{n}{b^j})$. At the leaf level, $j = \log_b n$, so the number of nodes is $a^{\log_b n}$. This is equal to $n^{\log_b a}$.

The first thing to do when applying the master method is to asymptotically compare $n^{\log_b a}$ to $f(n)$. In other words, when looking at the recursion tree, we are roughly comparing the work done at the leaf level to the work done at the root. There are three useful outcomes.

Case 1. The leaf level dominates the root level, by a polynomial factor.
This means something stronger than simply saying $n^{\log_b a} = \Omega(f(n))$. We are saying that $n^{\log_b a} = \Omega(f(n) \cdot n^\epsilon)$, for some positive ϵ. The n^ϵ is the polynomial factor that makes $n^{\log_b a}$ "much" larger than $f(n)$. If instead of n^ϵ we had a non-polynomial multiplicative factor (like constant, or logarithmic), the condition would not be satisfied. We can rewrite the condition as $f(n) = O(n^{(\log_b a)-\epsilon})$.
If case 1 is satisfied, then we have $T(n) = \Theta(n^{\log_b a})$. Again, this means that the amount of work at the leaf level dominates.

Case 2 is loosely interpreted as having about the same amount of work in all levels. If this were true, we would have $f(n) = \Theta(n^{\log_b a})$. Because there are $\Theta(\log n)$ levels, the total work becomes $T(n) = \Theta(f(n) \log n) = \Theta(n^{\log_b a} \log n)$. That is the extent of the coverage of case 2 in the textbook.
However, case 2 actually allows the root-level work to be a little larger than the leaf-level work. How much larger? Any multiplicative factor of $\log^k n$, for non-negative k. Formally, we allow $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$, where $k \geq 0$. (The case $k = 0$ is the basic case 2 described above).
Given the intuition that we have (somewhat) the same amount of work in all levels, so, as in case 2, we still multiply the root's work by a log factor and get $T(n) = \Theta(f(n) \cdot \log n)$, or in other words, $T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$.

Case 3 requires that the root dominates the leaf level by a polynomial factor.
Thus $f(n) = \Omega(n^{\log_b a} \cdot n^\epsilon) = \Omega(n^{(\log_b a)+\epsilon})$. This is the similar to case 1. However there is an extra condition, requiring that the work per level decreases geometrically. To establish this, we just need to compare the top two levels. That is, compare $f(n)$ with $af(\frac{n}{b})$, and show that the latter is at most a constant fraction of the former. That means showing $af(\frac{n}{b}) \leq qf(n)$, for some $0 < q < 1$. For every recurrence dealt with in this course, this will be true (and it's not really hard to prove).
The answer for case 3 is $T(n) = \Theta(f(n))$. Again, this means that root-level work dominates.

Comparison-based Sorting (mainly heapsort)

Even though sorting is a fundamental concept in algorithms, this course mainly uses it to introduce big-O, recurrences, and divide-and-conquer, with the exceptions of Heapsort and Quicksort. This section focuses on Heapsort after a brief note about Mergesort: I think it's safe to say that mergesort doesn't really need much of an explanation. One thing you might have not thought about is whether it can be implemented *in place*. For our purposes this means using only the space that holds the data, plus a constant extra amount. For mergesort it turns out that it is really difficult to avoid copying data, unless you're willing to sacrifice time complexity. Specifically, it is difficult to merge two sorted arrays without using extra space. (Linked lists are easier to handle).

Heapsort is another standard $O(n \log n)$-time sorting algorithm. It has one advantage over mergesort, in that it works entirely in place. Conceptually, a binary max heap (or just a heap from now on) is a binary tree where every node must be no greater than its parent. Assuming distinct elements for simplicity, the root stores the largest element, and any path from the root down to any leaf has elements decreasing in value on the way down. The other structural property of a standard binary heap is that it is balanced: if it has $k+1$ levels, then either they are all full, or the top k levels are full and the bottom level is filled in from left to right. Those two rules define a standard heap[2]. Notice that the subtree below any node in a heap is also a heap. Also, by the second rule, a heap with n elements has $O(\log n)$ levels. A heap is typically stored in an array; one index per node. We don't actually need pointers to form a tree, even though the heap is visualized as one. There is a correspondence between node positions and array indices. That is given by indexing nodes, top-down level by level, and left-to-right within any level. So the root has index 1 and the rightmost leaf in the lowest level has index n. Then, given any node with index j, if it has a left child, that will be found at index $2j$. A right child would be at index $2j+1$. The parent of j will be at index $\lfloor \frac{j}{2} \rfloor$. An easy way to remember this is to think of indices 1, 2, 3. So, it is just as easy to move around (between adjacent nodes) in a heap within an array as it is to do so conceptually when looking at the tree shape.

There are two primary ways of building heaps. One is incremental, meaning we add the k-th element after building a heap on the first $k-1$ elements. The k-th element is placed at the first available leaf position (rightmost in the lowest non-full row). In fact this is automatically true if we are just looking at the k-th index in the array, after having heapified the first $k-1$ indices. After bringing element k into the picture, we locally make sure that the *parent* $>$ *child* rule is maintained. So, having inserted the new element at the bottom, it will move up on the unique path to the root, as long as it encounters elements that are smaller. It is quite easy to see that this process restores the primary heap property. The path to the root could be logarithmic in size, in the worst case. If that happens every time, then this heap-building algorithm costs $O(n \log n)$ time overall. Convince yourself that there is an input that will actually cost $\Theta(n \log n)$ time to heapify. Again, remember that what really happens is we perform comparisons in an array, and make swaps as necessary.

The second heap-building algorithm processes the input array starting from the end, rather than from the beginning. Conceptually, we start with the correct tree structure, but

[2] There are other types of fancier heaps that achieve some pretty amazing results.

we have no idea what the values are. We begin at the rightmost leaf in the lowest level, and move right-to-left, then up to the next level starting at its right end, etc. Every time we move to a new node, we stop to make sure that the subtree rooted at that node is a heap. The last stop is at the global root, so when we are done with that, by construction we will have made a heap. The reason we move right-to-left, and in ascending level order, is to make sure that don't have to do a lot of work at every stop. Notice that all leaves are already heaps (each of size 1). So we can actually start this process at the rightmost non-leaf. In general, when we stop at a node, x, we can assume that its children are roots of heaps. So what could prevent the tree rooted at x from being a heap? The only thing that could violate the rules is x being smaller than at least one of its children. In this case we just swap it with the largest child. This makes the new root equal to the largest element in this subtree, so the new problem, if any, will again be rooted at x (one level down). So x will trickle down while it is smaller than at least one of the two nodes directly below it. The cost of making sure that the subtree rooted at x is a heap is $O(\text{height}(x))$, where height is the number of levels measured from the leaf level going up up to x. Height is always $O(\log n)$, so the cost of this method is $O(n \log n)$. However, some finer counting shows that the work is in fact linear, in the worst case. The key is that height is very small for lots of nodes. Recall that half of the nodes are leaves, and they have height 1 (or zero, depending on who's defining it). Every two leaves have one parent, with the possible exception of one single-child leaf. So the number of leaf parents is half the number of leaves. Similarly, every time we increment height by one, the number of nodes at that height is halved. So the amount of work can be upper-bounded by $\sum_{H=1}^{\log n}(H \cdot N_H)$, where H is height and N_H is the number of nodes with height H, which means we have $\sum_{H=1}^{\log n} H \cdot \frac{n}{2^H}$. The n can be factored out, and the sum becomes a constant (see class notes). So overall this is $O(n)$.

Once we've built a heap, we can use it to sort, easily. The largest element is at the root, and thus at the first index of the input array. We swap it with the last element in the array, so the largest element goes exactly where it belongs in sorted order. Thus we say that the largest element has been "extracted". Then we decrement a counter that marks the active prefix of our array. In other words, we want to leave all extracted elements where they are, and decrement the position of what is considered the "last element". Notice that the formerly last element, which we placed at the root, will be relatively small, given that we started with a heap. So it will most likely violate our first heap property. Thus it will have to trickle down, always making local comparisons and swaps, until the primary property is restored. This costs logarithmic time in the worst case. We have n extractions to perform, so the time will be $O(n \log n)$ overall. Actually, the true cost is the sum of *depths* of all nodes, i.e., counting the level of every node starting from the top. That is because the more we extract, the less trickling down there will be. Think of what happens after we have extracted all but few elements. The trickling down will only cost a constant. It's tempting to think that we will be able to use a summation trick like above, to reduce the time complexity. However, counting the sum of depths is much different than the sum of heights. Intuitively, there are many nodes with high depth, unlike height. In fact, no such trick will work, because heap sort consists of two phases: building a heap and iterative extracting. The former takes linear time as we demonstrated already. So the latter must take $\Omega(n \log n)$ time, because anything faster would imply that heapsort contradicts the known lower bound for the problem of comparison-sorting. That lower bound is described next, after an intro to decision trees.

Decision trees and the lower bound for comparison sorting

Decision trees are algorithms.
In this section I will specifically describe binary decision trees, where decisions are defined as comparing two elements and deciding which is larger. These trees represent algorithms that are based on doing just that: comparing elements. Examples are: mergesort, quicksort, heapsort, insertion sort, selection. All of these algorithms involve moving elements around, but each time we move an element, we do so because of the outcome of a comparison. So the number of comparisons involved in each of these algorithms is the key to understanding their time complexity. Critically, none of these algorithms perform other arithmetic operations, nor do they assume anything about the type or distribution of the data.

A (binary) decision tree can be shaped like any arbitrary binary tree. Each non-leaf node represents a decision (i.e., a comparison of two elements), for which there are two possible outcomes (hence the "binary"). When comparing b and c, we either have $b \leq c$, or $b > c$. (It doesn't matter which side is a strict inequality, as long as we're consistent). So the two branches below any non-leaf node represent paths to two different possible realities that differ in only one way: the outcome of the decision in that node right above. Each of the two branches will lead to another node. That might be another non-leaf, meaning that another decision is to be made. The new decision could be the same on both sides, but doesn't need to be. Algorithmically, the new decision might depend on the outcome above. This continues all the way down, until we reach a leaf node, which represents output. Overall, a decision tree is an algorithm, that compares elements iteratively (top-down), and produces its output according to the outcomes of the decisions. The unique path from the root to any leaf represents the execution of the algorithm for a particular input that is consistent with the outcomes of all the decisions made along the way on that path. The path is essentially discovering certain critical properties of the input, at least enough to be able to provide the desired output, without a hint of doubt. The length of any path represents the execution time, because it counts the number of comparisons used to get to the output. It is assumed that other work involved in the algorithm is overshadowed by the number of comparisons, as is the case for the algorithms listed in the first paragraph.

You can think of decision trees in terms of playing the game of 20 questions. Your friend is thinking of some person, and you have 20 question to narrow down who it is. The questions can only be answered by yes or no. Depending on the information you collect along the way, you might change your strategy, but in theory you could precompute your entire strategy for all possible combinations of answers that you might receive (good luck). That's what a decision tree is though. It encodes all possible sets of questions that you might ask, in the order that you will ask them. In some cases you might figure out who your friend is thinking of, using fewer than 20 questions. That would correspond to having a leaf at a relatively close distance to the root. Depending on the questions you ask, it might take you much longer to narrow down to one person. So a decision tree represents how good your algorithm is. What we care about is the longest path from the root down to a leaf. That's the worst case scenario.

As an aside, in general, decision trees don't need to be binary. If there are more than two outcomes for an operation, then that can be represented with nodes that have more than two children. For example, a decision could distinguish $<$, $=$, $>$. Also, the operations don't need to be restricted to comparisons. For example, there are *algebraic* decision trees, where decisions are made based on the sign of the solution to a polynomial. For instance, for some b, c, x, is $bx + c \leq 0$ or not? The higher the power of the polynomial, the more powerful the decision tree.

Here is a simple trivial problem: Given an array of n integers, provide the first index that stores the value 13, if one exists. The reasonable thing to do is scan through until we find a 13, or until we realize that there is no 13. Suppose that we have a binary decision tree that can distinguish $a{=}b$ vs $a{\neq}b$ at each node. Then the primary shape of this tree would just be one long path of non-leaf nodes. Each such node would have a leaf child (say, to its left) corresponding to an output that announces the current depth (corresponding to the current index scanned). The other child would be a new decision, this time comparing a new index to 13. However the lowest non-leaf node would have a second leaf child, corresponding to the output "not found". For the classic binary decision tree model where the nodes decide between $\leq$ and $>$, we would actually need two nodes to determine equality of two given integers. For instance, to compare the element at index i to the value 13, we would ask if that element is greater than 13, and if not, then we would ask if it is greater than 12. So the first node involving the parameter i would branch off to either the secondary node, or to a decision involving the next index, $i{+}1$. The secondary node would branch off to a positive output (declaring that index i holds 13), but also to a node dealing with the next index as well. In other words there are two paths corresponding to searching further (incrementing i). So the shape of the tree would be different. However it would also have linear maximum depth, which makes sense because this problem takes linear time to solve, whether you ask one or two questions per index. Note that we could determine equality by asking if $a \leq b$ and then if $b \leq a$, even for non-integer input.

Sorting lower bound:

Decision trees aren't just a different way of describing algorithms. They can provide powerful results about the difficulty of certain problems. Recall the game of 20 questions. You lose if you can't identify who your friend is thinking of, using at most 20 questions. Now, you might lose because you didn't come up with a good strategy (algorithm), and happened to be given a difficult person to discover. But in fact, no matter how good your strategy is, if your friend may use a huge pool of candidates, then your strategy will not be able to guarantee a correct answer in at most 20 questions every time. You can only guarantee to narrow down the set of candidates by a factor of a half with every question, so with 20 questions there's only so much that you can do. This is the more interesting use of decision trees.

Suppose that we have n real numbers that we want to sort. For simplicity let's assume that the numbers are distinct. Let our numbers be $a_1, \ldots, a_n$. The output will be a re-ordering of the numbers, so that they are in sorted order. In other words the output is a permutation of the numbers. For every possible input (of which there are $n!$) there is a distinct output permutation required to obtain a sorted set. A sorting algorithm must be able to recognize which permutation is required. So the algorithm must gain enough information, via comparisons, to be able to distinguish among all permutations. It might have to do even more work, but all we care about right now is that there is a bare minimum that must be done. For example, if we compare a_1 with a_2, and determine that $a_1 > a_2$, then we know that any permutation that maps a_2 to a position before a_1 is not the correct output. A sorting algorithm has to make enough comparisons among items in the input, to be able to eliminate all but one output permutation. Clearly if we compare all pairs we will have enough information to do this. But that just suggests that it should be easy to sort in quadratic time. What we're interested in here is a lower bound. How many comparisons are *required* to get enough information to distinguish among all permutations? It is important to note that we care about the worst case here. In the best case, we could just make $n-1$ comparisons, get lucky, and know the sorted order. For instance this would happen if we were given the numbers in sorted order and just compared elements at adjacent indices. But this strategy won't determine the correct permutation for other inputs; it hasn't done enough work. The problem we are concerned with here is to figure out the number of comparisons that *any* sorting algorithm will have to make, for some particular input that happens to make that particular algorithm work the hardest. Every sorting algorithm has its nemesis input. We're not actually characterizing what that nemesis is, we're finding out how much work every algorithm will be forced to do if it faces its nemesis, whatever that may be.

So, suppose that we have a decision tree that compares $<$ vs $>$ for some pair of indices at every internal node. The leaves represent output, and for the sorting problem we said output is a permutation. Every permutation must be represented by at least one leaf. If a permutation were missing, then how would our algorithm ever handle the input that requires that permutation? So the conclusion is that any decision tree corresponding to a comparison-based sorting algorithm must have at least $n!$ leaves.

The next question is, how shallow can a tree be, if it has at least $n!$ leaves? What we want to measure here is the height of the tree, in other words the longest path from root to leaf, because that corresponds to the worst case performance of the corresponding algorithm. The best way to minimize height is to construct a balanced tree, and such a tree will have logarithmic depth with respect to its size. If it is unbalanced, some path will have to be even longer. So even if our decision tree is perfectly balanced, it must have height no less than $\log(n!)$, by a a standard binary tree property. If this value is not an integer, then the height must be $\lceil \log(n!) \rceil$.

To recap, every decision node (i.e., comparison) gives us a little bit of information, that allows us to figure out what permutation we will need to apply the given input. In the best case we can eliminate half of the permutations that we're still considering. The problem is that there are $n!$ permutations that we start out with. Essentially, a perfect sorting algorithm would be able to do binary search on the set of all permutations. In reality there is no sorting algorithm that does this perfectly, but some come really close.

The value $\log(n!)$ is actually $\Omega(n \log n)$. In fact it's $\Theta(n \log n)$ but we are only proving a lower bound here. The classic way of showing this is to use Stirling's approximation. In my class notes (last page) I show another easy way as well. In any case, the interesting part of this lesson is how we got to $\log(n!)$ in the first place. This is a more precise answer than quoting $\Omega(n \log n)$ as a lower bound for sorting, especially for small values of n. We only use $\Omega(n \log n)$ because it has a familiar form.

The conclusion is that every comparison-based sorting algorithm that could ever be invented has no hope of sorting with fewer than $\log(n!)$ comparisons (and thus can't beat a time complexity of $n \log n$), in the worst-case.

Counting Sort and Radix Sort

From the previous section, we know that there is a $\Omega(n \log n)$ lower bound for the worst-case time complexity of any comparison-based sorting algorithm. This does not apply to *Counting sort* and *Radix Sort*, because they come with extra assumptions. Specifically, counting sort assumes that the data consists of n integers within a range k (between smallest and largest). All it does is use an array with one cell reserved for every possible input in the range, whether it is in the input or not. By scanning through the data, we can easily update how many times we've seen each value. Then by scanning through the counter array, we can easily list all input values in sorted order. All together this takes $O(n+k)$ time. What this simple procedure does not do is preserve the order of duplicates. In other words, this is not *stable*. This might matter if the order hides some secondary information (that happens to be in sorted order). Also, stability is critical when sorting is used as part of other algorithms, even other sorting algorithms such as Radix sort which follows. To get stable counting sort, we can convert our original counter array to a new helper array which stores the number of elements that are no greater than a particular value, for all possible values. This is illustrated in detail in the class notes. In a nutshell, the helper counter tells us how much space to leave in an array for each value, so that when we scan in the original input array we can easily map elements to the output array.

Given the tool of stable counting sort, we can set up Radix sort, which works under different assumptions. Here we assume that our n input numbers consist of ℓ digits. For instance, 5230909 has 7 digits. We can assume that all input has the same length, or make a conversion. The base, or *radix*, r is also a parameter. This is typically 2, 10, 16, for binary, decimal, hex, etc, but it could be anything. Radix sort first considers the least significant digit (i.e., the rightmost) in each of the n numbers. It uses stable counting sort to sort them, and then moves on to the next significant digit, does stable counting sort on those n values, etc. At iteration i, we will have sorted the suffixes (of length i) of all n numbers. So at the last iteration we will have sorted all n numbers. The proof is easy, by induction. At every iteration we deal with more important (significant) digits, which are the main criterion for sorting. Ties are broken by what happened in previous iterations, and that's why stability matters. In each iteration we deal with n digits, each of which is in the range from 0 to r, so counting sort takes $\Theta(n+r)$ time. Over all ℓ iterations, the total time complexity of Radix sort is $O(\ell \cdot (n+r))$. So the time complexity depends on the representation of the input. Notice that the length ℓ and the radix r are related. The smaller the radix, the longer the representation. Under certain conditions, the time complexity can beat other sorting algorithms.

Indicator random variables

An *indicator function* can be used as part of a mathematical expression that involves different scenarios. In some sense it is like an IF statement. For example, we could define the function $I(x)$ to be 1 if x is even and 0 if x is odd (assuming x is an integer). If we have some procedure that handles odd and even numbers differently, we can express this all in one. For example: $f(n) = I(n) \cdot n^2 + (1 - I(n)) \cdot 5n$.
This says that if n is even then we square it, but if it is odd then we multiply it by 5. Typically, indicator functions only output 0 or 1. Here, we are concerned with such functions in the context of randomized algorithms.

A *random variable* can be thought of as a function. As an example, a random variable X could be the sum of the numbers we see when we roll 3 dice. When we actually roll 3 dice, we know what X is, but that's not very interesting. It is more challenging to consider what we *expect* X to be, before rolling the dice. This expected value is a weighted average, and is written as follows: $E[X] = \sum y \cdot P(y)$. Here, the sum is over all possible outcomes y that X could have, each multiplied by the probability of it occuring. We have a summation instead of an integral because we are dealing with a discrete set of possible outcomes for X. If this expression is not clear to you, try out a few simple scenarios. What happens if there is only one possible outcome? What if there are 2 or 3 but they all occur with the same probability? Think of X as the amount of money you expect to get when you play the lottery that involves guessing a number. If you guess correctly, you win 2 million dollars, otherwise nothing. But the former happens once every one million times, and the latter happens with probability almost 1 (actually 1 minus a millionth). So you expect to get $2000000 \cdot 1/1000000 + 0 \cdot (1 - 1/1000000)$. Thus you expect to win 2 dollars. Sounds good, except whoever designed this lottery would know this, and would thus make sure to charge strictly more than 2 dollars for a ticket. On average the lottery company will win and you will lose.

Notice that the lottery example is a case where we could use an indicator function. There are only two possible scenarios, and precisely one of the two will occur (with a hugely different associated function). In fact, for an indicator function (or indicator random variable), X, that evaluates to either 1 or 0, we have $E[X] = \sum y \cdot P(y) = 0 \cdot P(y{=}0) + 1 \cdot P(y{=}1)$. Here, the possibly outcomes (y) are 0 and 1. So, by definition we have $E[X] = P(y{=}1)$. In other words, by definition, the expected value of an IRV is the probability that it evaluates to 1.

Sometimes it's quite difficult to evaluate the expected value of a random variable directly. This will be true if there are many possible outcomes, and it's difficult to group them together. Consider flipping a coin 10 times, and counting how many occurrences of HT we find. For example, in HTTHHTHTHT, we have HT appearing 4 times. But on average how many times will it appear? We could list all possible sequences of H and T of length 10, and then count for each sequence. But there are 2^k such lists, for length k. It doesn't seem to be much help to consider all the possible outcomes, even though there are less than 9. That is, HT will either not appear at all, or once, or twice, etc, but it clearly can't appear more than 9 times. In fact it can't appear more than 5 times. Still, figuring out the probability of HT occurring exactly 0, 1, 2, 3, 4 or 5 times seems just as annoying as the brute force exponential method we just mentioned. Here's where indicator random variables help us.

When we have to find the expected value of some variable X, and this involves counting individual events, each of which may or may not happen, *and* it's not straightforward to do it directly, one thing that should pop into mind is IRV. The goal will be to set up k new indicator random variables such that $X = X_1 + X_2 + \ldots + X_k$. Again this is why IRVs are like IF statements. Here, every X_j should be 1 *if* the corresponding event should be counted. Otherwise, it should be zero. We are interested in $E[X]$, so by definition we want to know $E[X_1 + X_2 + \ldots + X_k]$. It is always legal to convert this to $E[X_1] + E[X_2] + \ldots + E[X_k]$. In other words, we have $E_{j=1}^{k}[\sum X_j] = \sum_{j=1}^{k} E[X_j]$. This is by *linearity of expectation*, which you are allowed to take for granted in this course although it is technically straightforward to prove. Even if the various IRVs are dependent, linearity of expectation holds. It is often these dependencies that make a brute force calculation really difficult or time consuming in the first place. So far, the only difficulty is to define those IRVs. The second difficulty is to be able to quickly evaluate $E[X_j]$, for any j. As long as you can do that, you can evaluate the summation and you're done.

For the example with flipping coins, we have an X that involves *counting*: How many times will we see HT? If you were to count for a particular string, you would literally look at every index to spot an H, and for every such H you would check the next index to spot a T. So it makes sense to define $X_j = 1$ *if* we see an H at position j, and a T at position j+1. Otherwise $X_j = 0$. Then, $X = X_1 + X_2 + \cdots + X_9$. So we have just accomplished the first "difficult" task: setting up some IRVs that add up to X. Then we follow standard procedure by taking the expected value and applying linearity of expectation, to get $E[X] = \sum E[X_j]$. The second "difficult" task is to evaluate each $E[X_j]$. It turns out that for this problem, this is easy. As explained previously, the expected value of any IRV is equal to the probability that it evaluates to 1. So $E[X_j] = P(X_j{=}1)$. That is the probability that we find an H at index j and a T at index j+1. This probability is affected only by those two coin flips. There are 4 outcomes for flipping two coins: HH, TT, TH, and HT. Only one of four makes X_j=1. So $E[X_j] = 1/4$, for $1 \leq j \leq 9$. Thus $\sum E[X_j] = 9/4$.

Apart from this being a neat way of solving many interesting problems, the point of understanding IRVs in this course is to be able to later follow the analysis of randomized Selection, and Quicksort.

Selection (deterministic)

The selection problem involves a set of elements that can be ranked (i.e., sorted), and finding a particular element that has rank r, without actually finding all ranks (which would involve sorting). The classic example is to find the median of a set, i.e., $r = \lceil n/2 \rceil$. Duplicates can be handled as well, but we will just assume distinct values. The objective is to do this in linear time, just by using comparisons. This is achieved by the classic algorithm by Blum, Floyd, Pratt, Rivest, and Tarjan.

I won't describe the entire algorithm in detail. In a nutshell, it is a recursive algorithm that involves first "guessing" that a particular element x has the desired rank r. It is easy to verify if x has rank r: simply compare it to all other elements in the set and count how many are smaller. This takes linear time. This forms a *partition* of the set, and x is sometimes called a *pivot*. If x has rank p, and $p \neq r$, then we have to keep trying. To do this, we can place x at index p in the input array, and place all smaller elements at smaller indices, and all larger elements at larger indices. Note that we have no idea, nor do we care, where the other elements go [3], apart from having them partitioned with respect to x. Now, having just partitioned, we only need to reconsider one side of x in the array, depending on whether r is greater or less than p. For example, if $r < p$ then the element that has rank r must have a value smaller than x. This means that we have eliminated x and any element larger than x from the candidate list. Thus we can recurse on the subarray to the left of x. If $r > p$ we recurse on the right. To make the actual recursive call, we may have to recalculate the rank that we are looking for, because this rank is relative to the array we are recursing on. Specifically, this happens only if $r > p$. See class notes for an illustration. What remains to be shown is how to pick some x, without really guessing, with the goal of obtaining linear time complexity. Note that if we consistently (recursively) make guesses that give us elements with very high or very low ranks, we will get a quadratic-time algorithm. That is because we will keep recursing on problem sizes that are barely reduced. On the other hand, if our guess is always relatively near the middle of the subset we're dealing with, we will get linear time, via geometric series.

The paragraph above actually describes the randomized Selection algorithm, because it involves guessing x. The difficult part of the randomized version is the analysis (if you want to do it rigorously), for which you need to understand indicator random variables. The deterministic algorithm has a beautiful method of picking an x that guarantees a low time complexity overall. So far we know that after choosing an x, we will use it as a pivot, and then recurse. The time complexity for finding the element with rank r among n elements is
$T(n) = [\text{choose an } x \text{ without guessing}] + T(\text{new problem size if } r{\neq}p) + \Theta(n)$.
We need to pick x efficiently, and we need it to guarantee a relatively small problem size if $r \neq p$. In other words, we need p to not be extreme. The current $\Theta(n)$ term involves using x as a pivot to partition, and thus determine if $r = p$.

The solution is to find x recursively as well, after doing a linear amount of preparation work. Specifically, as mentioned, we want x to be larger than "many" elements, and also

[3] Actually, sometimes we do care. There is such a thing as *stable* partitioning, where the initial order of elements that end up in the same group is preserved. This is useful in the context of other algorithms, such as Radix sort, and Quicksort. Stable partitioning is not difficult.

smaller than “many” elements. One way to do that is to pick x as the median, but that’s essentially what we’re trying to do in the first place, so it’s not an option. So we could instead try to have x be the median of a significantly smaller subset of our input, in other words we could try to compute a sample median. Then at least our recursive call would be on a smaller problem size. Suppose we arbitrarily pick $n/5$ elements and compute x as their median. This would guarantee that x will have an overall rank of at least $n/10$ and at most $9n/10$, in the original set. Our worst case recurrence would be:
$T(n) \leq T(\frac{n}{5}) + T(\frac{9n}{10}) + \Theta(n)$, which unfortunately is superlinear (try it out). In other words, after picking x in time $T(\frac{n}{5})$, we might have $r \neq p$ and thus have to recurse on a subarray of size $\frac{9n}{10}$. Our goal of recursing on a “small” subarray is achieved, but it costs us too much time to accomplish this. Changing the number 5 to something else won’t help either.

Instead of picking $n/5$ elements arbitrarily, we can pick $n/5$ particular ones that will provide some information about the rest of the elements. That can be done by first forming $n/5$ groups of 5, and within each group determining the median element, all by brute force. The total time for this is $\Theta(n)$. Now we know that within each group there are 2 elements larger than the group median, and 2 elements that are smaller. We will let our x be the median of these $n/5$ group medians. What does this mean? We can make a copy of those $n/5$ elements and just solve the problem recursively, in $T(\frac{n}{5})$ time. This is just the same cost as before, when we had picked those $n/5$ elements arbitrarily. What changes now is the size of the recursive problem that we will have if x does not have our desired rank r. Specifically, if the rank of x is too high, that means that half of the group medians also have a rank that is too high, because they are all larger than x. We also know that for each of those group medians that we are too high, there are 2 elements (from within their respective groups) that have even higher values. So we have determined that there exist roughly $n/10$ group medians that can’t have rank r, and another $2n/10$ elements that also can’t have rank r. All together, roughly $3n/10$ elements are no longer candidates. So when we actually use x as a guess and as a pivot, in the worst case we will recurse on $7n/10$ elements. The case where the rank of x is too low is symmetric. Here I am ignoring some small boundary conditions that affect the number $7n/10$ slightly (or equivalently, that affect the range of n for which the claim holds). Our recurrence becomes
$T(n) \leq T(\frac{n}{5}) + T(\frac{7n}{10}) + \Theta(n)$. As a recap, the first term comes from choosing x, the second term exists only if $p \neq r$, and the last term is for partitioning but also doing all the brute force work with the groups before we recursively find x. By simple substitution we get $T(n) = O(n)$. This means that our new, smarter, recursive choice of x reduced the worst case size of the second recursive call (the one associated with the work that must be done if $r \neq p$), enough to get $T(n)$ down to linear.

Selection (randomized)

The randomized algorithm for selection is simpler than the deterministic one. See the first two paragraphs of the preceding section. We don't bother with the elaborate procedure to choose a pivot. Instead we use a randomly chosen element from the current subset of elements. If we know that the array of data is in random order, then choosing the element at any index is fine. So we can always just pick whatever is at the smallest index. We can always randomly permute the data once before running this algorithm, to make sure that every pivot will have a random rank. Without the assumption of random order, we might get sub-optimal behavior.

In any case, the simplification made by randomly picking a pivot comes with an added risk that the algorithm might become very slow. For instance, if we consistently pick a pivot that has an extreme rank (relative to the current subset), and the pivot never has the rank we're looking for, then we will consistently have to recurse on a subset with only the pivot removed. This will lead to quadratic time overall. The recurrence would be $T(n) = T(n{-}1) + \Theta(n)$. Here the $\Theta(n)$ represents the time it takes to choose the pivot (which is just a constant if we pick the first index), plus the time to partition. So, technically, randomized selection has worst-case quadratic time complexity. But we can prove that we expect it to have linear time complexity, even with certain pessimistic assumptions. It turns out that you would have to be really unlucky to get something slower than linear. Note that our analysis doesn't provide a probability of running in linear time. To do that, we could set a leading constant that we're happy with (say, $20n$), and show what the probability is of randomized selection running in less than that time. All we are saying with our analysis is that if we run randomized selection on sets of n elements, over and over, on average it will run in linear time. In fact we also get the leading constant that we expect to see on average.

The analysis begins with an important step: defining the time complexity of randomized selection. This time complexity depends on the rank that our random pivot will have. Specifically, the pivot will have some rank $k{+}1$, meaning that k elements are smaller than the pivot. We will always assume that the pivot does not have the rank we are looking for. That is our first pessimistic assumption. In other words, even if this happens consistently, we expect the algorithm to run in linear time. Given this assumption, when we choose our pivot, we will partition the current subset that we are dealing with, and recurse. As a reminder, this means comparing all remaining candidate elements to the pivot, and deciding which side has the element with the rank we are looking for. So we will recurse on all remaining elements that are smaller than the pivot, or on all elements that are greater. Now it's time to apply our second pessimistic assumption. Among those two groups, we will assume that we always have to recurse on the larger group. Even with this assumption, we expect to get linear time. Take a moment to realize that removing either of these two assumptions will only improve the time complexity. So, with the first pessimistic assumption, the time complexity is
$T(n) = \Theta(n) +$ ["either $T(k)$ or $T(n-(k{+}1))$ depending on the rank"]
but with the second assumption we just have $T(n) \leq \Theta(n) + max\{T(k), T(n{-}k{-}1)\}$. However, we have no idea what the rank $k{+}1$ will be. It could be anything from 1 to n.

We can use indicator random variables to express the fact that there are multiple possible

values of k but only one will actually occur. We define X_k to be 1 if our pivot has rank $k{+}1$, otherwise it is zero. In other words, we're letting the subscript of our IRV be the same as the parameter on the left in the *max* term of our recurrence. So for $0 \leq k \leq n{-}1$, we know that precisely one X_k will be equal to 1, and all others will be zero. Thus the time complexity is $T(n) \leq \Theta(n) + \sum_{k=0}^{n-1}[X_k \cdot max\{T(k), T(n{-}k{-}1)\}]$.
In that entire summation, only one term will be non-zero. So the expression is equivalent to what we had before.

What we've been interested in all along is $E[T(n)]$. We now know that this is no more than $E[\Theta(n) + \sum_{k=0}^{n-1} X_k \cdot max\{T(k), T(n{-}k{-}1)\}]$. Note that I've removed the square brackets that were previously used to show that the sum is over all terms that followed. Now the square brackets are just for expectation. What we have here is the expectation of a bunch of terms summed together. There are n terms in the summation itself, plus the $\Theta(n)$ term. By linearity of expectation we know that this is equal to the sum of the expectations of each individual term. So we have $E[\Theta(n)] + \sum_{k=0}^{n-1} E[X_k \cdot max\{T(k), T(n{-}k{-}1)\}]$. The first term has nothing random about it. Recall that it corresponds to partitioning. We can partition in dn time regardless of what pivot is chosen, so $E[\Theta(n)] = \Theta(n) = dn$.

Let's look at one term in the summation, for fixed k: $E[X_k \cdot max\{T(k), T(n{-}k{-}1)\}]$. This is the expected value of the product of *two* random variables. The first is the indicator random variable, X_k, which is determined by the random choice of pivot. The second is the recursive call. The size of the recursive call depends on k, but once we assume that we're going to recurse on this size, what happens recursively has nothing to do with why we recursed. In other words, we have a brand new subproblem, on some subarray, and what happens in that problem depends only on a brand new randomly chosen pivot. So the two random variables in our expression are *independent.* For any two independent random variables Y, Z, it is a fact that $E[Y \cdot Z] = E[Y] \cdot E[Z]$. You may take this for granted. So our expression is equivalent to $E[X_k] \cdot E[max\{T(k), T(n{-}k{-}1)\}]$. This happens for all terms in the summation that we had, so we now get $E[T(n)] \leq dn + \sum_{k=0}^{n-1} E[X_k] \cdot E[max\{T(k), T(n{-}k{-}1)\}]$.

For any IRV, we have $E[X_k] = P(X_k = 1)$, and in our problem, for any k this is $1/n$. In other words, all ranks are equally likely for our randomly chosen pivot. This is why it matters that we can assume that our data is randomly distributed. The result is that our expression simplifies to: $E[T(n)] \leq dn + \frac{1}{n}\sum_{k=0}^{n-1} E[max\{T(k), T(n{-}k{-}1)\}]$.

Next, notice that for both $k = 1$ and $k = n$, we have a term $E[max\{T(n-1), T(0)\}]$. This is just $E[T(n-1)]$. For $k = 2$ and $k = n{-}1$ we have a similar equivalence. They both contribute a term $E[T(n-2)]$. This double-counting pattern continues in a nested fashion. Note that if n is even then we have pure double-counting. If n is odd, then one term in the middle has no "partner". In that case we'll double-count it anyway and get a slight overestimate. So in either case, we can rewrite our expression:
$E[T(n)] \leq dn + \frac{2}{n}\sum_{k=\lfloor \frac{n}{2} \rfloor}^{n-1} E[T(k)]$.
Note that in the new range of the summation, $T(k) \geq T(n{-}k{-}1)$, so this is equivalent to the *max* term that we just got rid of. We could have also simplified to this instead: $E[T(n)] \leq dn + \frac{2}{n}\sum_{k=1}^{\lceil \frac{n}{2} \rceil} E[T(n{-}k{-}1)]$. We'll stick with the former.

We're trying to prove that $E[T(n)] = O(n)$, meaning $E[T(n)] \leq cn$, for some constant c. We will use the substitution method on the recurrence that we have just obtained. So, we assume by induction that $E[T(k)] \leq ck$, for all $k < n$. Then we can substitute into our expression, to get $E[T(n)] \leq dn + \frac{2}{n}\sum_{k=\lfloor\frac{n}{2}\rfloor}^{n-1} ck = dn + \frac{2c}{n}\sum_{k=\lfloor\frac{n}{2}\rfloor}^{n-1} k$.

We're going for an upper bound here, so we're free to exaggerate things. Note that $\sum_{k=\lfloor\frac{n}{2}\rfloor}^{n-1} k \leq \sum_{k=1}^{n-1} k$, which is an expression we often encounter. However, if we make this exaggeration, we won't be able to conclude that $E[T(n)] \leq cn$. Try it out. Instead, we can use the fact that $\sum_{k=\lfloor\frac{n}{2}\rfloor}^{n-1} k \leq \frac{3}{8}n^2$. This is actually not hard to figure out analytically. Roughly, when the sum is from 1 to n, we get approximately $\frac{1}{2}n^2$. If we subtract the sum from 1 to $n/2$, then we are subtracting roughly $\frac{1}{2}(\frac{n}{2})^2$, which is $\frac{1}{8}n^2$. It also helps to think of the diagram that I provide in the class notes. Anyway, with this observation, we get rid of the pesky summation and have $E[T(n)] \leq dn + \frac{2c}{n} \cdot \frac{3}{8}n^2 = dn + \frac{3c}{4}n$. It is now easy to set c large enough compared to d so that this is all less than cn. Specifically if $c \geq 4d$, we're all set. Recall that dn corresponds to the time that we spend partitioning before we recurse, when the problem size is n. So this is a small constant, and accordingly we get a fairly small constant for the expected runtime of the entire algorithm. Note that this is true even though we made some pessimistic assumptions. The expected average cost is even lower (although still linear, because of the trivial lower bound).

What should you take away from all of this?
Well, for one, now you have some proof that you should expect randomized selection to perform well on average, even though it can easily have a bad time complexity in the worst case. In fact, not only does it perform well, on average it matches the best case and the deterministic algorithm!
This is an example of how a randomized algorithm can compete against a deterministic one. The code for randomized selection is definitely simpler, and in fact we expect it to run faster.
This is a nice relatively advanced example of how awesome IRVs are for formal proofs.
There's a pretty good chance that you'll be asked to explain something about medians in job interviews, and being able to at least talk about the basics of this and the deterministic algorithm can put you ahead of the competition.
Finally, take a moment to think about how relatively short and elegant both results on selection are.

Quicksort (randomized analysis)

I'll assume that you know what Quicksort does. In fact the algorithm is quite similar to randomized selection. They both involve choosing a random pivot, and perform a partition on that pivot. That means grouping all elements smaller (or respectively larger) than the pivot. The two groups are placed in the array, to the left and right of the pivot, which resides at the index that it would have had if the array were sorted.

In selection, after partitioning, we recurse on one side (or not at all if our choice of pivot was lucky). In Quicksort we always recurse on both sides. In fact as the algorithm runs, every single element will be chosen as a pivot, precisely once. The *luck* factor in Quicksort has to do with the relative size of the two groups that are formed when we partition. If they are anywhere near the same size (i.e., if one is a constant multiple of the other), then the partition is essentially balanced, and we are "lucky". Consistently obtaining balanced partitions (even with this relaxed definition) leads to a time complexity of $O(n \log n)$. To understand this, see the class notes where we get a skewed recursion tree, in which the work at all levels is linear. In fact even if we get balanced partitions only some constant fraction of the time, we still get the same asymptotic time complexity. An example of that is given in the class notes where I give an informal definition of lucky and unlucky recursive calls. Like with selection, the worst case time complexity of Quicksort is quadratic. In the worst case, the pivot will cause an extremely unbalanced partition, and we will have to recurse on a problem size that is reduced by just a constant. Whatever that constant is, we will get quadratic time if this keeps happening.

The randomized analysis of Quicksort starts off similar to randomized selection. Once again, the time complexity depends on the rank of the pivot, so all we can do is determine the average time that the algorithm will take, over many sorting jobs of size n. We have a non-recursive partition component in the algorithm, which takes $\Theta(n)$ time. After that we recurse twice. For a pivot that has rank $k{+}1$, the two problem sizes are k and $n{-}k{-}1$. So, either we have to recurse on $T(0)$ and $T(n{-}1)$, or we have to recurse on $T(1)$ and $T(n{-}2)$, etc.

So we define n IRVs, exactly as for selection. $X_k = 1$ if our pivot partitions the current array range such that the left subarray ends up with size k. The recurrence involving n different possible scenarios can be written concisely as
$T(n) = \Theta(n) + \sum_{k=0}^{n-1}[X_k \cdot (T(k) + T(n{-}k{-}1))]$. Of all these recursive terms in the summation, only one will be non-zero.

The next few steps are part of the standard recipe for problems solved with IRV, but in particular are rather similar to the randomized selection analysis. That is, we apply linearity of expectation to get a summation of individual expected values, we observe that the expected value of the non-recursive term is just $\Theta(n)$ (because it isn't random), and for each term we use independence to get the product of two expected values. For the last step, as with selection, an IRV doesn't affect what happens in the future within a recursive call, so the two random events are independent. Finally, we observe that $E[X_k] = 1/n$, regardless of what k is. As before, this is just saying that all ranks are equally likely when we pick a pivot. So we conclude that $E[T(n)] \leq \Theta(n) + \frac{1}{n}\sum_{k=0}^{n-1} E[T(k) + T(n{-}k{-}1)]$. The summation is $\frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n{-}k{-}1)]$, by linearity. These two terms are in fact equal, so we have $E[T(n)] \leq \Theta(n) + \frac{2}{n}\sum_{k=0}^{n-1} E[T(k)] = dn + \frac{2}{n}\sum_{k=0}^{n-1} E[T(k)]$.

Notice how similar our expression is to what we had obtained for the selection problem: $dn + \frac{2}{n}\sum_{k=\lfloor\frac{n}{2}\rfloor}^{n-1} E[T(k)]$. The fact that the summation is now from 0 to $n-1$ makes a big difference. With the substitution method we can show that $E[T(n)] = O(n \log n)$.

Technicalities: To get the substitution to work, unfortunately we resort to using an inequality "from the appendix", which is that $\sum_{k=2}^{n-1} k \log k \leq \frac{1}{2}n^2 \log n - \frac{1}{8}n^2$. Recall that we used an analogous (more intuitive) inequality in the selection proof. To give just a little bit of intuition here, observe that a clear exaggeration of $\sum_{k=2}^{n-1} k \log k$ is $n^2 \log n$, by making every term use n instead of k. If the function were linear, a slightly less exaggerated sum would be half of this amount: $\frac{1}{2}n^2 \log n$, as an average of the lowest and highest term. In fact the function is convex, so the sum will be even less than the average of the extremes. It turns out that it is significantly less, i.e., a fraction of n^2. Details are omitted, but that's enough to give us enough flexibility to prove the upper bound of $O(n \log n)$. Also notice that the appendix inequality is for a sum from $k = 2$ to $n-1$, not from $k = 0$ which is what we had ended up with. So to use the inequality, we had to take care of the cases where $k = 0$ and $k = 1$. The former corresponds to $E[T(0)]$, but this is in fact zero. The latter corresponds to another base case. We could also argue that $E[T(1)]$ is zero, because we don't really need to do anything to Quicksort one element. But even if we adopt the convention that it costs some constant amount of work, if we separate this term from the summation, it gets swallowed up by the non-recursive $\Theta(n)$ term. Think of it this way: if we had to do twice the non-recursive work for partitioning instead of dealing with $E[T(1)]$, the final result would have been the same.

As with the selection analysis, it's easy to get frustrated with the details of the proof and miss the big picture. What matters here is that you now know what is involved in showing that Quicksort is expected to be efficient, even though in the worst case it can perform rather poorly. In fact the efficiency of $O(n \log n)$ matches the best deterministic sorting algorithms (excluding those that make additional assumptions about the data).

Once again, IRVs help to solve a problem that looks quite complicated, in an elegant and (believe it or not) simple way. The set up of this proof is the real lesson: how we formulate the recurrence, and how we use IRVs. After that it's standard business, admittedly with a trick or two mixed in.

Alternate analysis of Quicksort

This analysis gives the same result as the previous section, but is also a nice example of another way that IRVs can be used. Instead of using IRVs to distinguish among several possible scenarios, we're going to use them to count the expected number of comparisons involved in Quicksort. That's good enough to analyze the expected time complexity, because any constant-time operation that Quicksort does can be charged to a comparison that needs to be made (and no comparison gets charged more than $O(1)$ times).

First of all, notice that in Quicksort every pair of elements will be compared either once or not at all. Specifically, all comparisons occur because some random pivot has been chosen. That pivot is compared to everything in a specific range (in between previous pivots). Then elements are partitioned into two groups and they never interact again. So we can set up an IRV for every pair of elements, to be 1 if that pair gets compared (precisely once), or 0 if the pair never gets compared. In fact we will do this after relabeling the input. We let the input be labeled as $\{z_1, z_2, z_3, \ldots, z_n\}$, such that all the elements appear in sorted order. Now, that's what we're trying to get as output, so remember that this is just a proof, not an actual algorithmic reordering. We're ready to define our IRVs as described above:

$X_{ij} = 1$ if z_i is ever compared to z_j, otherwise $X_{ij} = 0$.

Clearly the total number of comparisons, X, in Quicksort is equal to the sum of X_{ij} over all pairs, i.e., for all $1 \leq i < j \leq n$. In the course notes this is a double summation. We want to know what $E[X]$ is, so we care about the expected value of the sum (over all pairs) of X_{ij}. By linearity of expectation, we want the sum (over all pairs) of $E[X_{ij}]$. So all we need to figure out now is what $E[X_{ij}]$ is, for any given i, j.

In other words, we need to know the probability that z_i and z_j will be compared. As mentioned, in Quicksort every comparison involves a pivot and one other element, and in fact that other element must not have been a pivot yet. Moreover, to be compared, both elements must have fallen on the same side of every pivot that was considered beforehand, otherwise they'd end up on different sides and would not be compared.

Let Z_{ij} represent a subsequence of our sorted list, from z_i to z_j (inclusive). While we happen to pick pivots that are outside Z_{ij}, both z_i and z_j will land on the same side and will still have a chance of being compared eventually. They will only be separated if we pick a pivot that is in Z_{ij} but not z_i or z_j. So basically we don't care how many pivots were chosen outside this range, all want to know is what is the first element in Z_{ij} that became a pivot. If it's z_i or z_j, then they will be compared. If it's anything else, they will not be compared. Therefore the probability that z_i and z_j get compared is 2 divided by the size of Z_{ij}, i.e., $2/(j - (i - 1)) = 2/(j - i + 1)$.

Now the rest is just algebra. We know that $E[X_{ij}] = 2/(j - i + 1)$, so over all pairs $(1 \leq i < j \leq n)$ we have:

$$E[X] = \textstyle\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (\frac{2}{j-1+1}).$$

It is very easy to show that this is $O(n \log n)$. See the very end of the course notes. What really matters is the setup.

Binary search trees (BST)

A binary search tree (BST) is a data structure that stores a set of elements, and supports one main type of operation / query: searching for a particular given value. Secondary (but rather common) operations are to add elements to the set, or delete them. As the name suggests, the structure is a binary tree. The property that makes it a B**ST** is that every node has a value larger than that of all nodes in the subtree to its left, and smaller than all nodes in the subtree to its right (assuming distinct values). This means that if you perform an in-order walk on the tree, you get all the values in sorted order. This also means that the primary operation, i.e., searching, is easy to do. Given a target value, x, we compare to the root, and if not equal then we go to the left or right subtree depending on whether x is smaller or larger than the root, respectively. Eventually either we find x or reach an empty subtree, which means x is not in the BST. The cost of searching is equal to the length of the path that descends from the root until x is found or we reach a leaf. So, in the worst case, the cost of searching in a BST is equal to the height of the tree. Inserting an element is just the same as searching for it, except at the end we create a new node.

Deleting an element depends on how we are accessing it. We might have a direct pointer to the node, or we might have to search for the value in the BST, in order to access the corresponding node. Once we have access to the node, if it has at most one child then we can in delete it "instantly". This is easy to see for a leaf, or for the root of a tree (in other words for nodes of degree 1). If there is one child and a parent, we can still just link the parent and child after deleting our node. On the other hand, if the node, x, that we want to delete has two children, then the standard procedure is to replace x with its predecessor or successor within the tree. The successor is the smallest value in the subtree to the right of x (and this subtree exists in the case that we are now considering). This is found by first visiting the right child of x and then following a path only visiting left children, recursively, as long as possible. When we find a node without a left child, that is the successor of x. Finding the predecessor of x is symmetric. It doesn't matter which one we use to replace x. Once we do replace x though, say with the successor, y, we might have to replace the former position of y with another node, in order to keep a connected tree structure. Given that y had no left child, an "instant delete" operation does the trick. In other words, either there is nothing to do, or we just link the parent of y to the right child of y, effectively promoting the entire subtree to the right of y up by one level. The cost of deleting is that of finding the successor, so this is like a search (although technically we're not looking for a specific value), which costs the height of the tree in the worst case.

The bottom line is that the cost of searching, adding and deleting elements is proportional to the height of the BST, in the worst case. This is why we are interested in keeping the height as small as possible.

Worst, best, and expected cost of constructing a random BST

It is easy to see that a BST could have linear height. For example, sequential insertion of $1, 2, 3, \ldots, n$ will create one long path. The cost of inserting any element is equal to its depth: Recall that to insert an element, we search for it and then add it as a leaf. This means that when we look at a BST, we know exactly which elements were compared to each other during its construction, assuming no deletions were made. For any particular node, we know that it was compared to the entire path up to the root, and nothing else. So in the example above, the k-th element costs $\Theta(k)$ to insert, because it is compared to all values between 1 and $k-1$. Thus the cost of iteratively building a BST can be quadratic in the worst case.

On the other hand, construction of every BST requires $\Omega(n \log n)$ comparisons. The "best" situation is a BST that is perfectly balanced. In this case, each of the leaves is at depth $\log n$, so the sum of their insertion costs is roughly $\frac{n}{2} \log n$. Why is the balanced tree the least costly BST? Any other tree of the same size must have nodes at greater depths. We can imagine plucking out nodes from the balanced tree to place them at deeper positions. Thus their insertion costs increase. This is illustrated in the class notes.

There is another simple argument for why constructing a BST on n nodes costs $\Omega(n \log n)$ time in the worst case. As mentioned, an in-order walk provides the elements of a BST in sorted order. So the entire process of building a BST and then performing an in-order walk is a sorting algorithm. Specifically, it is based on comparisons. So the classic sorting lower bound applies, and this entire process must make $\Omega(n \log n)$ comparisons. Because the in-order walk is cheap, it must be the construction of the BST that is the bottleneck. In other words, a fast (say, $O(n)$-time) BST construction would imply that sorting could be done faster as well, but we know that's impossible.

The time bounds mentioned above are identical to those that we concluded for Quicksort: quadratic in the worst case, $\Theta(n \log n)$ if we're lucky. For Quicksort, we're lucky if we keep getting balanced pivots / partitions. For a BST, we're lucky if we end up with a balanced tree. In fact it turns out that the similarity is no coincidence. To get a balanced tree, we'd need a root that is the median of the elements (similar to a perfect Quicksort pivot). The same holds for the root of every subtree, recursively. So there is a close relationship between tree roots and pivots.

This remarkable similarity is illustrated in an example in the class notes. Take any array with randomly distributed data, and iteratively build a BST by inserting elements with increasing indices. Then take the same input array and run *stable* Quicksort. The term "stable" means that when we use an element x as a pivot, all terms greater than x are placed after x in the array, in the same order as they appeared in the input. This is a standard way of doing Quicksort, and is easy to implement. So, when we start constructing a BST from the array, we pick the first element and place it at the root of the tree. Correspondingly, for Quicksort, we use it as a pivot. In both cases, smaller values will end up "to the left", and larger values will end up "to the right". In Quicksort, this happens immediately, as we partition, and then we recurse on each side. For the BST, every element is compared to the root, but then also to other elements further down, before inserting a new element. So the exact order of operations is not the same, but what matters here is that *eventually* every element will be compared to the pivot, or, respectively the root, and these two are the same element. Also what matters is that this element partitions the rest of the data into two groups that never interact with each other. Recursively, the same thing will happen again. New pivots will correspond to new subtree roots. So, for instance, every element in the left subtree of x will be compared to the root of that subtree, but correspondingly that root will be the same as the new pivot in the subarray to the left of x in Quicksort. Following the example in the class notes should make this clear.

The bottom line is that iterative BST construction and stable Quicksort perform *exactly* the same comparisons, just in different order. Thus they have the same time complexity. The expected number of comparisons in Quicksort is exactly the expected number of comparisons for constructing a BST on n elements. This means that if we are given a random permutation as input, we expect to construct a BST on the input in $\Theta(n \log n)$ time. If you are given an array that might have some structured order that causes BST construction to run slowly, you could randomly permute it first, and then construct a BST. This would minimize any possibility of obtaining a slow performance.

Note that if the expected number of comparisons is $\Theta(n \log n)$, then we expect a logarithmic number of comparisons per node (on average). Think of the perfectly balanced tree again. Half of the nodes are leaves, and they will have gone through $\log n$ comparisons. So even if all the other nodes only needed one comparison (a clear underestimate), we still get $\frac{1}{2} \log n$ as the average. It turns out that a logarithmic expected number of comparisons per node does not imply that a randomly constructed BST has $O(\log n)$ depth. One could create a BST which looks perfectly balanced, except for one long path containing a super-logarithmic but sub-linear number of nodes. For instance, let the chain have length $\sqrt{n}$. Thus the bulk of the nodes would be in the nice balanced position with low depth, and the size of the chain would not affect the average depth asymptotically. Yet the tree depth would no longer be logarithmic, it would be $\Theta(\sqrt{n})$. This is illustrated in the course notes. The bottom line is that the equivalence to Quicksort gives us a quick result about the time to build a randomized BST, but it doesn't promise us much about the worst-case time complexity of searching in that tree. However, there is a different analysis (beyond the scope of this course) showing that the expected depth of a randomly built BST is in fact $O(\log n)$. It requires IRVs and a few other fun tools as well.

Conclusion: Building a random BST has the time complexity of Quicksort. If we don't want to rely on randomness and expectation, we can still get deterministic logarithmic depth for a BST, and thus deterministic logarithmic-time search / insert / delete, with a number of carefully designed trees. One way to do this is to use a *red-black* tree (see next page).

Red-black trees

Red-black trees are just binary search trees with additional restrictions. These restrictions are designed so that any valid red-black tree will have $O(\log n)$ depth. First, every node is colored either red or black. To implement this, we just add a variable to every node, setting it to 1 or 0. Second, the root is always black. Third, we add black leaves to the BST, wherever possible. Thus all proper nodes are non-leaves. The important restrictions are:

- 4. We never allow two successive red nodes on a path from root to leaf. In other words, every red node has a black parent.
- 5. The black-height property: For *every* node x in the BST, the following holds. If we count the number of black nodes from x down to every leaf (separately), we obtain the same number on every path.

The last restriction *must* apply to every node, but of course we treat each node independently: we can't possibly expect to get the same number for every node. However, when we focus on any single node, we must get the same number on every descending path starting at that node. The black-height of a node x does not count x itself, but it does count the extra leaves that we placed in step 3. This is just convention.

Red-black trees have logarithmic height / depth:
When we look at any two paths descending from a particular node x, the longer path cannot have more than twice the length of the shorter path. Why? Both paths must have the same number of black nodes, by property 5. Let that number be k. To make the shorter path as short as possible, we give it k black nodes and no red nodes. To make the longer path as long as possible, we must place k black nodes, and insert as many red nodes as possible. By property 4, we can't insert two of them in a row, so we can place at most $k{+}1$, alternating with the k black ones. In fact, given that the last node is a black leaf, we can only place k red nodes (one above every black node). Hence the factor of 2. Now consider a perfectly balanced tree, which we know has logarithmic depth. From this tree we could create any other binary tree, by plucking out nodes and placing them at greater depths (in a nice way that creates a connected tree). But by plucking out any node, the minimum depth drops below $\log n$. By the factor-2 argument above, the maximum height can't go beyond $2\log n$.

Another classic way to prove the logarithmic depth of a red-black tree starts by removing every red node. To do this, for every such red node, y, we let the parent of y become the parent of the children of y. This operation is called a *contraction.* The parent and children of y are black (by rule 4). The new parent might thus obtain up to 4 children, if it initially had two red children. In any case, by property 5, the black-height of the root tells us how many black nodes are found on any path from the root down to any leaf. With all the red nodes removed, we now know that all leaves are at the same level (depth). In other words, the contracted tree is balanced (by depth, not by number of children for each node).

Next, consider that the contraction did not affect the number of leaves in the tree, because they are black. Given n initial nodes, we have $n{+}1$ (fake) leaves. A height-balanced *binary* tree on this many leaves has logarithmic height. The contracted tree that we just created has the same number of leaves but is not binary. Every node has at least two children, but might have three or four. This extra branching factor means that the height is at most that of the perfectly balanced binary tree on the same number of nodes (or leaves). In other words, for a tree with a fixed number of nodes, the more it branches out, the shallower it must be. Thus our contracted tree also has at most logarithmic depth. Now re-insert all the red nodes, and consider the longest path from root to leaf. By property 4, it can double in length after re-inserting, but no more. Thus the depth of the initial tree is at most $2 \log n$.

So, by the properties that define a red-black tree, the tree must have $O(\log n)$ height. It isn't necessarily fully balanced, but it is reasonably balanced. What matters is that if we have a red-black tree, we can search, insert and delete in logarithmic time.

Maintaining a dynamic red-black tree: The next thing to consider about red-black trees is how to *maintain* all the restrictions after inserting or deleting, and in fact how to do so in logarithmic time. If we can do that, then we have a data structure that supports any sequence of searching, inserting, and deleting, in logarithmic time per operation.

There is a fairly simple algorithm to handle insertions, relying on a case analysis for a constructive proof of correctness. See the class notes for an illustration. Briefly, when we insert a node, it becomes a leaf, as with any BST insertion. Then we greedily try to color it red, which means that it won't affect the black-height count for any existing node. There is a problem only if its parent is also red. When this happens, we perform some local recoloring (according to a case analysis), and push the problem upward. For instance, we might create two consecutive red nodes a bit higher up, but at least the node we just dealt with will no longer be in violation of property 4. In fact, this is done in a way such that the black-height property is not violated anywhere. Recursively, we recolor and push the problem up, until we reach a particular case that resolves the problem, or until we reach the root. If we reach the root, it is black, so property 4 can no longer be violated. Otherwise, the particular resolving case that was just mentioned will involve not only some recoloring, but an actual transformation to the structure of the tree, via an operation called a *rotation*. This is done in a way that preserves black-height, and property 4 is no longer violated anywhere.

The **rotation** just mentioned exists in two symmetric forms: there are left-rotations and right-rotations. To right-rotate a node x, the node must have a left child, y. Visualize placing your right hand on x and the left hand on y, which is slightly below and to the left of x. Imagine the line segment between your two hands and rotate it clockwise, while bringing the child up to take the place of x. Thus x becomes the right child of y. Next, consider what happens to the children (or in general, to the subtrees) of x and y. There were three such subtrees: two belonging to y and one belonging to x (on its right). These are just cut and pasted back in, at the unique positions that they can belong. In fact, only the right subtree of y must change links. It now becomes the left subtree of x. All of this is much easier to understand by looking at the simple illustration in the notes. A left-rotation on x is symmetric. This time, x must have a right child, and we place the left hand on x, and rotate counterclockwise. Again, if the left child of x is y, then performing a right rotation on x followed by a left rotation on y will restore the original tree.

The conclusion is that when we insert into a red-black tree, we might have to perform a logarithmic amount of recoloring, as we ascend towards the root, and finally one or two rotations. Note that most of the work just involves flipping bits (recoloring), so in fact further searching can be done *while* we are maintaining a red-black tree, because the structure is only affected at the very end of the insertion. Deleting a node is a little trickier, but relies on the same principles. This is handled in the book, but we won't cover it in class.

Applications of dynamic BSTs

Dynamic balanced binary search trees are used frequently in computer science. The word "dynamic" means that data can be inserted and deleted. The non-dynamic (static) case is relatively uninteresting. In the dynamic context, it is implied that we will be performing a series of actions: queries, insertions and deletions. The goal is to perform each of these operations relatively quickly, compared to what we could achieve in the static case. It is often assumed that we are allowed a certain amount of *pre-processing* time, to at least build a basic data structure. This section outlines three applications.

Rank-finding and Selection. We know how to find the element with rank k in $O(n)$ time, for any k. That is the Selection problem. In fact we can solve this problem in $O(\log n)$ time, as long as we are first allowed some pre-processing time to build a balanced BST. Now, we know that building such a tree on n elements costs $\Omega(n \log n)$ time. So the only point of doing this is if we are going to perform many rank *queries*, and also if we might insert and delete data in between.

Determining the rank of a given node in a BST can be done quite simply, by *augmenting* the tree. That is just a fancy word for saying that we'll store a little more information at every node. For every node, we will store the size of the subtree that has its root at the node. Then the rank of the root of the BST can be found by just visiting the left child of and checking the tree size there. The root's rank is just the size of that left subtree plus one. If we're looking for the rank of a non-root node, x, we perform a search for x but do some counting along the way. If x is to the left of the root (because it's smaller than the root), we simply search in the left subtree. On the other hand, if x is larger than the root, we go right but add one plus the size of the left subtree of the root to our count. We are simply recording that the root and every node to its left are smaller than x, and thus must contribute to the rank of x. That is the simple rule: when going left do nothing; when going right, add a subtree size (plus one) to the current count. This involves a constant amount of work per level, so it is logarithmic overall, as long as the tree is reasonably balanced. Note that we could also calculate this sum by traveling up the tree from x towards the root. All that matters is the path between the root and x. Subtrees to the left should be counted, and subtrees to the right should be ignored.

The selection problem (finding the node that has a given rank k) is solved in a similar way. We first compare the given k to the rank, r, of the root (by checking the left subtree size). If they match, we're done. Otherwise, if $r > k$, we search in the left subtree. Finally, if $r < k$, we recurse in the right subtree, but we now look for rank $k - r$ instead of k, because we've just excluded r elements from the picture (the left subtree, and the root). So this search takes logarithmic time if the tree is balanced, because we simply walk on a path from the root down to the leaf level.

The previous arguments simply tell us that we can do rank queries and selection in logarithmic time, if we are allowed preprocessing. To allow a dynamic data set, we have to show how to maintain the subtree size at every node when inserting and deleting data (for instance, in a red-black tree). We'll just focus on inserting, because that's all we covered in class. When inserting a node into a RB-tree, as we search for its position, we increment the subtree sizes of all its ancestors. Then to rebalance, we may do some recoloring and finally a couple of rotations. The recoloring doesn't affect subtree sizes. The rotations will. For example, a right-rotation of a node x will affect the subtree size of x but also that of its left child, y. The good news is that we have immediate access to the subtree sizes of the children of x and y, and that's all we need to reassign values to x and y. No other subtree sizes are affected. The conclusion is that subtree sizes are easy to maintain when rebalancing a RB-tree after insertion. Deletion also involves rotations, although we didn't cover it so it won't be mentioned further.

Note that given the pre-processing time of $\Theta(n \log n)$, using a BST for these problems is worth it if we will have more than a logarithmic number of queries.

Range counting. The input to this problem is a set of numbers (elements), and two values, L, R, representing the left and right ends of a *range*. For simplicity let's assume that L and R are not in the set, so elements are either strictly inside or outside of the range. We want to know how many elements have values within the range. Clearly, without any preprocessing this takes $O(n)$ time, and in fact there is a matching lower bound because you can't afford to not look at all of the input. Preprocessing could simply involve sorting the set, after which a range query takes $O(\log n)$ time: we binary search for L and R to find their ranks with respect to the set, and subtract one from the other. So, if you are going to make many range queries, it is worth it to pay for sorting once at the beginning.

The next thing to consider is performing range queries on a dynamic data set. Once again, augmenting a BST with subtree sizes solves this problem. In fact, this is just dynamic rank-finding. In other words, we can just find the ranks of L and R, in logarithmic time as mentioned above. Note that we don't need to insert L and R in the tree, we just traverse from root down to the positions that they would be in, each time counting subtree sizes.

Interval trees. The input to this problem is a set of (one-dimensional) intervals, each consisting of a left and right endpoint. Then, given a query interval, Q, we want to know if Q partially overlaps any of the intervals in the set. This is a yes or no answer, although a positive answer is supplied by providing an overlapping interval.

It is easy to tell if an interval V overlaps Q. Let subscripts L and R denote left and right endpoints of an interval. If $V_L < Q_L$ there will be overlap if and only if $Q_L < V_R$. On the other hand if $Q_L < V_L$ there will overlap if and only if $V_L < Q_R$. All this takes $O(1)$ time. Obviously we could compare Q to the entire set and get our answer in linear time, but by building a balanced BST as pre-processing, we can perform queries in logarithmic time. Now, how do we build a BST on objects that are described by two values (endpoints)? The point of a BST is to exploit some sorted ordering, so we can try to use left endpoints as keys. In other words, as far as the BST is concerned, there is no structure whatsoever regarding right endpoints. But at least every node is aware of what its right endpoint is. This doesn't count as augmenting, it's just extra info that is already present in the input. Let's see what we can do with this tree. We can compare Q to the interval represented at the root, in $O(1)$ time. If we're lucky, there will be an overlap and we're done. Otherwise, there are two cases: Q is either entirely to the left or entirely to the right of the interval at the root. In the former case we're almost as happy as if we had found an overlap. We now know that the entire right subtree can't possibly overlap Q. Why? Suppose that the root is called V. For Q to be entirely to the left of V, we must have $Q_R < V_L$. But all left endpoints in the right subtree are greater than V_L, thus there is no possible overlap there either. So we ignore the right and recurse on the left subtree, without knowing if we will find an overlap or not. We just got rid of a constant fraction of the problem in constant time, which is great news. In other words, we moved one level down in the tree in constant time; which means we're well on track to get logarithmic time overall. But not so fast... what if Q is entirely to the right of the root? Now we have gained no extra information at all: Q may or may not overlap any given interval on either side. This approach has reached its limits. Except, we can augment the tree to handle this case. At every node, we will store the value of the maximum right endpoint found among all intervals represented within that subtree. Now, if Q is entirely to the right of the root, we can compare Q_L to the *maximum right endpoint*, M, stored at the left child of the root. If $M > Q_L$, there must be some interval in the left subtree that overlaps Q. That is because all left endpoints of intervals in this subtree are to the left of Q_L. So we recurse to the left, knowing that eventually we will find an overlap. We don't care if there might be an overlap in the right subtree, because we're only looking for one. On the other hand, if $M < Q_L$, there is no possible overlap within the left subtree, so we recurse right. We don't know if we'll find an overlap there, but we're happy enough to have spent a constant amount of time to discard the left subtree and move one level down. The conclusion is that in each case we know which direction deserves recursion, and there is only one such direction. So in logarithmic time we will either find an interval that overlaps Q, or report that no such interval exists.

Finally, we should mention how to maintain our augmented data (the maximum right endpoint of each subtree), when inserting into a RB-tree. As with subtree sizes, it is pretty easy. When searching for the (leaf) position of a new node, we update the maximum of any node it is compared to on the way down. Then, when rebalancing, we only have to worry when we perform a rotation. Consider right-rotating a node x, with left child y. When we do this, x will inherit the right subtree of y, which will become the left subtree of x. So we recalculate x by computing the max among its two subtrees. Similarly, we can recalculate the max for y. Basically, maxima are easy to maintain when rotating.

It should be intuitive that problems such as rank finding, selection, range counting, and interval overlap are fundamental for data analysis and retrieval. There are also high-dimensional extensions, which you can learn more about in a followup course (ask me).

Dynamic Programming

Dynamic programming is an important and common algorithmic technique. It is used for problems that have recursive formulations, where specific recursive calls (i.e., with specific parameters) are made several times. In a nutshell, dynamic programming is just a method of keeping track of the results of recursive calls, so that they don't need to be solved more than once (each). Doing this in a *top-down* manner is also called *memoization*. What this means is that you proceed with regular recursion, but just before recursing with certain parameters you check to see if you've already done the exact same thing, with exactly the same parameters. How can you tell? You have a table where you store *memos*; one for every possible combination of parameters you might recurse on. So if you do find a memo then you return the stored answer and do not recurse. If you don't find a memo then after the recursive call is complete you store a memo with the answer, to be ready for the next time. By avoiding repeated recursive calls, the savings can be huge.

In *bottom-up* dynamic programming all base cases are solved first, followed by subproblems that rely only on the base cases, and so on. This is done iteratively, so recursion is avoided, but this requires identifying an ordering among all possible subproblems, to iterate through. As with memoization, solutions are stored in an array, so that solving larger problems relies mainly on looking up solutions to smaller problems.

The top-down approach might get lucky in the sense that it's possible that only part of the table will be filled in. For some problems, this top-down approach could work faster on certain input. In the worst case the behavior of both approaches is the same. The bottom-up approach can save space in certain situations, and as mentioned avoids recursion.

The key to using dynamic programming is to get a recursive formulation for the solution, and then to recognize that there is a limited number of distinct recursive calls that might be made. This distinct number will often also be the size of the table that you will end up using.

Longest Common Subsequence (LCS). The input to this problem is two strings of characters, of length n and m, respectively. The question is, what is the longest subsequence that is common to both strings? A subsequence is not a consecutive substring; it is just a set of characters read from left to right, with possible ignored characters in between. For instance, let string X be ABCBDAB, and let string Y by BDCABA. So, $n = 7$ and $m = 6$. By inspection we can find a subsequence of length 4 that is common to both, like BDAB. In fact we can find three such answers, and no common subsequence of greater length exists. So BDAB is a LCS. To visualize an LCS, write down the two strings horizontally, one above the other, and draw straight segments between matching characters, without any of the segments crossing. What you're looking for is the largest number of such segments that can be drawn.

Let's assume that we can compare characters in constant time. This is quite reasonable if the alphabet we are using has a constant number of characters. One way to find the LCS is to enumerate all possible subsequences of X, and check to see if they are found in Y. The checking part is easy. Given a candidate subsequence, we just scan through Y trying to find the next matching character. But the number of subsequences in a string of length n is exponential. Every index could be used, or not, so every binary number of length n maps to a subsequence. This is a terrible algorithm.

We will temporarily relax the problem to ask only for the length of the LCS, not the string itself. The solution can be found recursively. To do so, we will need to find the length of the LCS among sets of *prefixes* of the input. So, let $c(i, j)$ be the length of the LCS among the strings $X[1, \ldots, i]$ and $Y[1, \ldots, j]$. That is, among the prefix of size i in X, and the prefix of size j in Y. What we are really looking for overall is $c(n, m)$. It turns out that this value can be found by looking at the last character in each string of the input, and making a couple of recursive calls. Here's why: Consider trying to compute $c(i, j)$ in the case where $X[i] = Y[j]$, i.e., if the last characters match. We claim that $c(i, j) = 1 + c(i-1, j-1)$, i.e., one recursive call is involved. To prove this, notice that either $X[i]$ or $Y[j]$ must be involved in the LCS of $X[1, \ldots, i]$ and $Y[1, \ldots, j]$. If both were not involved, we could just match them up and improve the solution. But if one of the two characters has been matched, and not with the other, that prevents the other from being matched at all (thing of the visualization described in the first paragraph). Therefore you might as well match the two respective last characters instead, and get a solution that is at least as good. Hence the $+1$ and the recursive call on the two prefixes.

So, what if the last two characters don't match, i.e., $X[i] \neq Y[j]$? Then we will solve two new subproblems, each obtained by ignoring one of the last characters. The solution will be either $c(i, j) = c(i, j-1)$, or $c(i, j) = c(i-1, j)$, whichever is larger. In other words, $c(i, j) = max\{c(i, j-1), c(i-1, j)\}$. The reason for this is that in this situation we can't use both of the last characters to obtain an LCS, so one of the two must be ignored (again think of the visualization; if both are are involved in a match then the corresponding edges that we draw will cross). Thus we examine the only two possible options: hide $X[i]$ or hide $Y[j]$. Technically both could also recurse with both of them hidden, but this scenario will actually be considered within each of the first two.

So, with the recursive solution above, we obtain a recursion tree that branches out once or twice at each node, depending on whether two characters match or not. Clearly the worst behavior happens when we don't see a match, in which case we have to try out two possible avenues. In each such case one parameter is decremented. So it is easy to see that certain smaller recursive calls will be reached in many ways. See the example in the class notes, where we begin with computing $c(7, 6)$ and end up needing to compute $c(5, 5)$ several times. It is precisely this repetitive usage of the same recursive call that memoization and dynamic programming are meant to deal with. As pure recursion goes, our very simple recursive algorithm is still exponential.

As mentioned, memoization checks if a solution to a subproblem has already been computed before proceeding with recursion. Such solutions are stored in a table (array), where the size is the number of subproblems (i.e. solutions) that we might have to deal with. In the case of the LCS problem, we can use an array of size $n \times m$, because there are n prefixes in X and m prefixes in Y. In the recursion tree, we get to ignore every subtree that corresponds to a node that is a copy of one that has been visited before. For instance, the first time we visit a node corresponding to solving $c(5, 5)$, we check to see if we've solved this subproblem before, realize that we haven't, and continue normally. But the next time we need to solve $c(5, 5)$, we can realize that we've done this before, via table lookup. The values stored in the table are simply the lengths of longest common subsequences. At index i, j, we store $c(i, j)$. We initialize with a negative value to signal that no length has been computed yet.

The time complexity of solving the LCS length problem via memoization is $\Theta(mn)$. That is because there are just as many subproblems, but for each one all we have to do is look at the table and then either return a value in constant time, or recurse, again in constant time. I'm not saying that the entire remaining recursive procedure takes constant time, but rather that the cost to make a new recursive call is just a constant. So, table lookup has just brought the complexity down from exponential to quadratic!

Bottom-up dynamic programming bypasses recursion altogether. We use the same table, but fill in entries in a different order. Top-down memoization will recurse normally until some base case is solved, at which point a subproblem that relied on it will also be solved, and the table will be filled in accordingly. Eventually some other base case will be solved, and so on. With bottom-up dynamic programming, we start by solving all base cases. It is conceivable that we won't even need some of them, but we basically don't care. We just solve all of them, fill in the table, and move on to subproblems that are slightly more general. For the LCS problem, we begin by implicitly noting that the length of the LCS of an empty string with any other string is zero. We don't really need to record this in our table, but conceptually we can form a topmost row of $n+1$ zeros, and a leftmost column of $m+1$ zeros. The intersection at the top left corresponds to comparing two empty strings. The first real base case is that where both strings have length 1, i.e. it is $c(1,1)$. In this case, either they match or they don't, and the answer is 0 or 1. As explained earlier, recursively this would have been computed by looking at the two characters and using a smaller subproblem. In fact the entries in the array that we need, i.e., $c(0,0)$, $c(1,0)$ and $c(0,1)$ are all located immediately above, to the left, and diagonally (above and left). This is true of any entry that we need to compute. It will always rely on those three relative positions. So at this point we can repeat filling in entries for which the corresponding immediate neighbors are already filled in. According to our recursive formulation, for a particular entry at position (i,j), if $X[i] = Y[j]$ then we should look at the value above and to the left, and add 1. Otherwise we should copy over the max among the entries above and left. Thus we just fill the table in, row by row, or column by column, or via any other valid pattern that we like. At the very end we will find the answer to the problem, at the bottom-right corner. The time to fill in the table is $\Theta(nm)$, just as it was for memoization.

We can actually obtain the LCS itself, by tracing back in the table from the bottom-right up to a base case. Tracing is done according to the recursive formulation. When we are at an entry that corresponds to matching characters, we move diagonally, up and to the left. This corresponds to using the particular (common) character in the LCS, and continuing to write down more of it in a right-to-left manner. Whenever we are at an entry for which the corresponding indices in the strings do not hold matching characters, we either move up or to the left, depending on which of those entries holds a larger number. All we are doing is following the recursive reasoning that defines the LCS substructure and dependence on solving subproblems. So, any path formed in this way will literally spell out an LCS, read in reverse order. We can actually get all possible longest common subsequences in this way! Of course, there would be an additional cost for writing them all down. In $\Theta(nm)$ time though, we find at least one, and we have all the information available to produce all of them.

Finally, here is a neat little trick for the bottom-up approach. We don't actually need the entire table, to get the length of the LCS. We can use $\Theta(min\{n,m\})$ space, by overwriting information that we no longer need. For instance, to compute the values in a row of our table, we only need the row above it. So we can get the length using linear space, by using two arrays of linear size; one representing odd rows in a table, and the other representing even rows. This will still take quadratic time. Unfortunately this makes it impossible to simply trace back to get the LCS itself. But it turns out that with some more care this can also be computed in linear time. That is beyond the scope of this class.

Longest Increasing Subsequence (LIS). In this problem, we are given one sequence, S, of n characters (or anything that can be ordered). For example, $WTEHKRDGAKRGFD$. Within S, we want to find a subsequence that is in sorted (increasing) order, such that there is no other increasing subsequence that is longer. Recall that a subsequence does not necessarily consist of consecutive characters from S. In our example, EKR is an increasing subsequence. So is AKR, but $EGKR$ and $DGKR$ are longer. I probably would have had a more interesting example if I had bashed my keyboard differently.

As with the LCS problem, let's first focus on finding the length of the LIS. We can solve this problem by constructing an array A of size n, to store certain scores. $A[j]$ will store the length of the LIS that is restricted to ending at $S[j]$. In other words we mean the LIS of the prefix of size j in S, with a forced usage of $S[j]$. We will fill the array incrementally from left to right. To start, the first index gets a score of 1, trivially. That is our base case. Inductively, assume that we have correctly filled our array up to index $j-1$. Now to compute the value of $A[j]$, we need a LIS that ends by using $S[j]$. That means that $S[j]$ must be larger than the previous character in this LIS, so we can scan through $S[1, \ldots, j-1]$, and for each character $S[i]$ that is smaller, we map to $A[i]$ and see what subsequence length we achieved up to that position. We should clearly add $S[j]$ to the largest solution among all those candidates. But what happens if $S[j]$ is not larger than any of the characters to its left in S? In that case, we set $A[j] = 1$, because the LIS that is forced to end with $S[j]$ consists of $S[j]$ itself.

When all this is done, we scan through A and report the largest value. Basically, the LIS of S must end at some index. We have found n constrained solutions, one of which is the actual solution. The time complexity of this algorithm is $O(n^2)$. To compute $A[j]$, we spend $O(j)$ time scanning to the left of $S[j]$, and then $O(j)$ time selecting a maximum value from a subset to the left of $A[j]$. Obtaining the actual LIS is not difficult. Each $A[j]$ "knows" where it got its score from, and we just work backwards from the max in $A[n]$.

As an aside, here is the first example of a *reduction* shown in this class. Instead of solving LIS from scratch, we could make a sorted copy, C, of S. Then the LCS of C and S is what we're looking for. That is because any such LCS must be an increasing subsequence in S, and any increasing subsequence in S is a candidate solution for the LCS. Transforming the LIS problem to an LCS problem, and mapping back the solution, takes linear time. The LCS problem can be solved in quadratic time, as we have seen.

Rod Cutting. The input here is a rod of length n, that can be cut at integer positions. After the rod has been cut several times, each piece is sold. For every possible size, there is a given price. The question is, where to make the cuts, to maximize profit.

This can be solved recursively as follows. Decide where the leftmost cut will be, sell the piece to the left, and recurse on the rest. We have n options for the leftmost cut, including the option of not cutting at all (equivalent to cutting at position n). Let $P(k)$ represent the profit for a rod of size k, and $c(k)$ represent the profit of selling a piece of size k without cutting it. Then we want $P(n)$, which is the maximum of $c(k)+P(n-k)$, for $k = 1, \ldots, n-1$. Note that we also need to consider $k = n$ but in this case we only care about $c(n)$ and we don't recurse.

It is not difficult to see that this recursive formulation will require subproblems to be solved repeatedly. So we can use dynamic programming, by creating an array to hold all of the scores for rods of size 1 to n. Each time we need the score for a larger rod size, we rely on having computed the score for all smaller sizes, and we spend linear time finding the best combination of the sum given above. So the entire algorithm takes quadratic time.

Hashing

As far as we are concerned here, hashing is a way to access data (and possibly insert and delete data). The objective is to *expect* constant time operations, as much as possible. Constant time access is already possible with an array, assuming we have a reliable and quick way to map our data (keys) into the array. This would be possible if all our keys were distinct, with values $1, \ldots, m$, and we used an array of size m. We still use an array as the primary structure in a *hash table*, but if we don't have the conditions just mentioned, we are no longer able to map, or hash, our keys into the table both reliably and quickly, in the worst case. For instance if the keys have values not in the range of indices of the table, we need some way to map them into that range. Also if we have duplicates, we need to deal with them. The fundamental issue here is to construct a *hash function* that maps keys to slots in the table. Note that multiple keys can map to the same slot. This is in fact unavoidable if the number of keys, n, is larger than the table size, m.

There are many functions that map keys to slots. Most likely modular arithmetic will be involved so that no key maps out of bounds. Besides that, there are heuristics involving powers and prime numbers, designed to avoid certain pitfalls, but we won't deal with that here. Lots of analysis of hashing involves **assuming** that one has a function that behaves randomly (perhaps a better words is "uniformly"), yet reliably. That means that a specific key (value) will map deterministically to the same slot every time, but also that as we hash many keys we will observe them landing uniformly in the table.

Chaining is a simple way to hash. We simply let our hash function map keys to the table, and if two keys every map to the same slot, we start a chain, or linked list. Then, besides the time to compute the function, insertion takes constant time, because we can just insert directly or append to the list. Searching takes time proportional to the list size, in the worst case. Note that in some applications we would want to search before inserting, to avoid having duplicates. In any case, we want to keep the maximum or average list size as small as possible. In other words, we want as uniform a distribution as possible, after hashing n elements.

The *load factor*, α is simply the number of keys divided by the table size, so, $\alpha = \frac{n}{m}$. If we **assume** that we have a good hash function that maps keys uniformly into the table, then α is the average list size, so the load factor tells us how much time we expect to search in a hash table. The more your function deviates from this assumption of *simple uniform hashing*, the more α fails to capture the search time.

Open addressing is the second main simple way to hash. Here, we assume that the table is at least as large as n, in other words $0 < \alpha \leq 1$. No chaining is done, so every key has to map to the table itself. But as before there is a possibility of two keys *colliding* at the same slot. In other words, a particular key being hashed might find that another key is already occupying the spot that it wants. In this case we essentially re-hash the current key. What this means is that our hash function must describe an entire sequence of hashes, in case we keep getting collisions. Ideally the hash function will produce a permutation of the indices of the array, which will guarantee that eventually we'll find an empty slot for any key. There are many such functions. *Linear probing* operates by using a primary hash function (as with chaining), and then trying consecutive slots until a free slot is found. Instead of consecutive slots, we could also make jumps of fixed size, but then we should make sure that the jump size is not such that slots are revisited. Of course, all of this is taken *mod* m.

Quadratic probing starts with a primary function and then hops around the table in quadratically increasing jumps. The point of this is to make the sequence (or permutation) of visited slots appear to be random, and to avoid *clustering* which occurs in linear probing. Clusters are simply large consecutive occupied slots. They start ruining the performance of open addressing.

With open addressing, given a probing sequence, inserting a key involves going through the sequence of slots until an empty one is found. Searching is the same. Once an empty slot is found, we report that the key is not in the table (because we would have put the key there if we had needed to insert it). However, deleting keys is problematic. To delete a key x, we need to find it, which means we hash to find its primary slot. Of course, we might find another key there, so we continue searching, assuming that the insertion of x went through a series of collisions and hashes. But if one of those keys that x collided with during its insertion has subsequently been deleted, then searching for x again will fail. That is because we will find an empty slot and assume that x isn't in the table.

To analyze open addressing, we make the **assumption** that we have a hash function that generates "random" probing sequences (consistently though). In other words, each key gets a deterministic probing sequence, but the function is such that any permutation is equally possible. This assumption is simply not true for linear probing or quadratic probing, because they generate a rather limited number of probe sequences. In any case, under this assumption of *uniform hashing*, it can be shown that expected search time is inversely related to α, so if we have have a table size m that is a constant multiplicative factor larger than the amount of data, n, we will expect a constant amount of time to search. It is not sufficient to have a table size that is just a little larger than n.

Note: In this class, you should never use hashing as part of an answer to a problem, unless that problem is specifically about hashing. In other words, don't make any of the assumptions that hashing uses, unless you are told that you can make those assumptions.

Amortization

Amortization is just a better way of analyzing the cost of a collection of operations, compared to figuring out the worst case cost of one operation and assuming that this might happen every time. It works when expensive operations are relatively infrequent. The analysis is sometimes non-trivial, but the technique is frequently used.

In class we cover the array-doubling example. As a reminder, this involves inserting elements into a set stored in an array. Every time the array fills up, we copy everything over to a new array of twice the size. This will happen at one particular insertion, so it is true that the worst-case cost of an insertion is linear. But it is clear that this doesn't happen every time. By careful counting, we can show that the cost of n insertions is linear, i.e., the average cost per element is constant. In other words, the amortized cost of an insertion is constant.

In problems where amortization is possible, there are cheap operations and expensive operations. The goal is to get some intuition about the frequency of expensive operations and/or figure out a pattern that must exist in a sequence of operations. The accounting method, or banker's method, involves estimating an average (amortized) cost and assuming that this will occur for every single operation. Then whenever we get a cheaper operation we know that we have overestimated the cost. We carefully calculate this overestimate and how many times it occurs, so that we can use the excess to pay for expensive operations in the future. It is not permitted to let expensive operations happen and then "promise" to pay for them later via cheap operations. As long as we always have an overestimate of what has already happened, we're ok. In the array-doubling example, we "guess" an amortized cost of 3. Had we guessed some other larger constant, or $\log n$ or some larger function, the analysis would have still worked. In fact we would have started to notice that we are overestimating a bit too much, so eventually we'd probably be able to bring the amortized cost down to the right constant.

The potential method involves coming up with a function of the structure holding your data, such that the function will decrease a lot whenever you have an expensive operation, and not increase a lot whenever you have a cheap operation. The idea is that changes in this function will balance the individual operation costs. The potential, Φ_i, for the structure at operation i, can depend on any number of things: the size of the set, the way things are arranged, the structure of your tree, graph, stack, or whatever you're using, etc. You can use any anything quantifiable that has to do with your data, but it has to be something *static*. Your potential function measures something in a *snapshot* of the data structure; it is illegal to use something that has (quantifiably) changed between two iterations, i.e., between two states of the data structure. For any of this to make sense, we need to be more clear about what a potential function is and what it's effect is on amortizing. First of all, we define the amortized cost of the i-th operation, $\hat{c}_i$, as $\hat{c}_i = c_i + \Delta\Phi_i = c_i + \Phi_i - \Phi_{i-1}$. In other words, by definition, the amortized cost is equal to the actual cost plus the difference in the potential function between consecutive states of the structure. If we have a function that is zero when we have no data (i.e. at iteration zero, $\Phi_0 = 0$), and the function is always non-negative, then we can take advantage of a telescoping series to claim that the sum of all amortized costs is an overestimate of real costs. This is shown in the class notes. Essentially, the Φ values all cancel out once we sum over all operations (i.e. over i), except for Φ_n and Φ_0, although the latter one is zero itself. So, by definition, the sum of all amortized costs is equal to the sum of all real costs plus the function evaluated only at the very end, and this is non-negative. Therefore the amortized cost is an overestimate. You don't need to re-prove this every time you use the potential method. It's a proof that holds for any function that is zero at the beginning, and otherwise always non-negative. This condition for Φ_n and Φ_0 is really useful but amortization can still work if it doesn't hold, in certain situations. The course notes do mention this. Note that the *change* in potential, i.e., $\Delta\Phi$, can be negative. It better be sometimes, for any hope of getting a result.

Once you've got a potential function, what remains is to evaluate the amortized cost, by the definition $\hat{c}_i = c_i + \Delta\Phi_i$, for all different types of operations that exist. Some of the operations will have cheap real costs, and some will be expensive, relative to the amortized cost that we would like to obtain. But there could also be various levels of cheap or expensive operations. For each distinct type, we calculate an amortized cost. As mentioned, if you have a potential function that decreases a lot whenever you have an expensive operation, that will make the amortized cost of this operation cheaper. But you have to define your function in a way that accomplishes this without making the cheap operations too expensive (via large increase in potential). In the array-doubling example, an expensive operation occurs when we double the size of the array, which is why it makes sense to have the potential be a function of the array size.

Finally, once you have an amortized cost for each type of operation, you assume that the worst type occurs every time, and thus overestimate the sum of amortized costs. That in turn is an overestimate of the sum of true costs, so now you hopefully have a better bound for this.

To most people, the potential method looks more complicated. There is this mysterious potential function, but ultimately you're probably doing no less work than figuring out what a good guess should be for the accounting method. The potential method is frequently used in advanced results in algorithms and data structures.

The array-doubling problem is a good example to use as an intro to amortization, but it doesn't reveal the true power of amortizing. That is because this problem is very predictable. We know exactly when the array will double. Give me an iteration number, and I can tell you if it will correspond to an expensive or cheap operation. Instead, amortizing is more interesting when we have way less information about when the expensive operations will occur. In such cases, we need to figure out patterns or frequencies for certain operations. For instance, we can come up with arguments to show that expensive operations can only occur if several cheap have preceded (not necessarily all in a row). As an example, in the simulation of a queue by two stacks, we don't know when things will be popped or pushed, but we can still get an amortized cost, for any possible sequence of such operations. The same is true for the problem with coins that I always hand out.

BFS and DFS

I don't think there is much that needs to be said for basic searching in graphs. These are things that everyone should have seen before taking this course anyway. A few things worth mentioning:

BFS and DFS are used in at least two contexts: searching for a particular target vertex, or simply exploring a graph.

BFS finds all shortest paths, in terms of number of edges, from the root (or source) vertex.

Both BFS and DFS are often used as tools for solving other problems on graphs. For instance, finding cycles.

Both BFS and DFS take $O(V + E)$ time, because we reach every vertex and consider every edge. For connected graphs, because $E \geq V - 1$, we can abbreviate this as $O(E)$. However, when using BFS or DFS as a tool when solving some other problem, we might get a faster time (e.g., $O(V)$) if we don't need to carry out the entire search.

Searching on a directed or disconnected graph simply involves restarting when an individual search is over, if there are remaining unmarked vertices.

The time complexity of BFS and DFS depends on the structure used to store the graph. It is best to use an adjacency list, at least if we are not using the search as a subroutine for another problem. BFS is implemented with a queue, and DFS uses a stack.

Topological Sort, and Strongly Connected Components

Topological sorting is a procedure performed on directed acyclic graphs (DAG). It involves listing all vertices of the graph, so that if the graph has an edge from x to y, or if there is a directed path from x to y, then y appears to the right of x in the list. If there is no such path, then x and y can be placed arbitrarily. We can think of topological sorting as squishing a DAG so that all its vertices are on a horizontal line, and all edges are directed to the right. There are potentially many valid topological sorted orders for a given DAG.

To topologically sort a DAG, we just run DFS and list vertices in reverse order of *finishing times*. A vertex is considered to have finished when it has no unmarked neighbors through which to continue the DFS. So, when there is nothing that the vertex can reach, it can be added to the list, and all future vertices can safely go to its left. To prove that this is valid, it is sufficient to look at any DFS tree and consider how it was created (see course notes).

A **strongly connected component** (SCC) in a directed graph is a subset of vertices, such that every vertex in the subset can reach every other vertex in the subset, via directed edges in the graph. In fact the subset should be maximal, meaning that if you can place two vertices in the same component, you do so. We can create a graph G' from G, by representing every SCC of G as a vertex of G'. A vertex in G' will have an edge to another vertex if there was a corresponding path in G between the respective components. Then, no two vertices in G' can mutually reach each other, otherwise we would have merged their SCCs. This means that G' is a DAG.

How to identify strongly connected components:
The algorithm is simple. Run any DFS on G and record finishing times. Then create the transpose G^T of G, which has the same vertex set but all edge directions reversed. Finally perform a DFS on G^T, by processing vertices in the order of finishing times given by the first DFS on G (process larger finishing times first). Whenever we perform a DFS from some unmarked vertex, all vertices that it finds will belong to its component. After marking all such vertices, we continue with a DFS from the next unmarked vertex in our list.

To **prove** that this algorithm works we first establish certain properties about the finishing times generated by the DFS on G. Let A and B be two arbitrary SCCs of G. Of course, A and B are not known in advance. There are two cases to consider. The first is when there is no path from A leading into B and vice versa. Then whichever component is discovered first by the DFS will also finish first, before the other component is discovered at all. For example, if the DFS discovers A before B (either by starting in A or wandering into it from outside), then the DFS will search all of A, possibly exit to discover other parts of the graph and backtrack to A, but it will have no path to B. Thus all vertices of A will finish, before the DFS backtracks out of A or resumes elsewhere to eventually discover B. The conclusion so far is that in this case (where A doesn't have a path to B, and vice versa), whatever DFS we run on G, there will exist some "time" that splits all finishing times of vertices in component A from all finishing times of vertices in B. We don't care which comes first, all we care about is that there is a clear split.

Next consider the only other possible case, without loss of generality: component A has a path into B (but not vice versa, by definition of strongly connected components). In this case, if a DFS begins in B or discovers it first, all of its vertices will finish before A is discovered, by similar reasoning to the first case. On the other hand if a DFS discovers A first, then there will be at least one vertex of A that will finish after B. Why? Because we know there is a path from A to B, so some vertex in A has an edge leading directly outside A and continuing along some path to B. That vertex can't finish before the path to B is followed, all of B is explored, and the search backtracks along the path. But not all vertices of A will necessarily finish after B. For example, suppose that vertex v of A has a path to B, but no other vertex in A does (without going through v). If a DFS begins at v and happens to explore A before following the path to B, then at least one vertex in A will finish before B is discovered. The conclusion about this case (A has a path to B but not vice versa) is that either all vertices in A will finish after all vertices of B, or the finishing times of A will *nest* the finishing times of B but not vice versa. (Once we discover B, we explore it all before returning to A).

Putting both cases together, we conclude that regardless of how we run a DFS on G, the list of finishing times will place the vertices of any two strongly connected components into two or three "blocks" (ignoring vertices of other components). Specifically we will get two blocks in arbitrary order if the components are unordered, whereas if component A has a path to component B then we know the order of the two blocks (A to the left of B), or we get two blocks of A with one block of B in between.

Now let's look at how edges of G might be directed, between these blocks. Given our definition of A and B, we know that there are no paths from B leading to A. So what about edges (and paths) from A leading to B? We have already established that if there is such a path then there must be such a path originating at a vertex of A that is in a block to the left of B (i.e., with higher finishing times). We're happy with such paths. What we want to show is that there can't be a directed edge from the right block of A (if it exists) pointing into B. In other words, **if there is any edge at all pointing to the left in our list representation of G, it must go within a strongly connected component.** We can prove this by contradiction: If some vertex v belonging to A is in the right block of A, then it finished before B. If v also has an edge pointing to the left into B, then v could not have finished before using that edge in a DFS, which means B must have finished before v.

Finally, we can establish that the DFS in G^T recognizes components. Whenever we start a new DFS at the leftmost unmarked vertex, u, we claim that u will find all vertices of its component U and nothing else. We know that u has a path to all of U, by definition. We can assume that none of it will be marked, by induction. In other words no vertex to the left of u mislabeled an element of U. Now, how might u mislabel an unmarked vertex, x, to its right in the list? The DFS from u would have to follow a path along G^T, eventually using an edge from some element u' of U to x. Could that edge point to the right? No, because that would imply that there was an edge from x to u' in G pointing to the left, but we established that this is impossible. So, what if u leads to some u' that is to the right of x? Then there is an edge from x to u' in G. But u is to the left of x, thus x is nested between two elements of U. We have already established that if a block can't have edges towards another block that nests it; contradiction. This last statement can be explained in more direct way as well: if x has an edge to u' in G then x finished after u'. But accessing u' implies that$u$ must have been accessed already or will finish before backtracking to x, and thus u would necessarily finish before x, contradicting the hypothesis that u is to the left of x.

Minimum Spanning Trees (intro)

Given a connected undirected graph G with edge weights, a minimum spanning tree (MST) is a subgraph of G that has three properties:
1) it is a tree.
2) it spans all vertices of G, meaning all the vertices are in the MST.
3) it minimizes the sum of edge weights. This means that no other spanning tree can have a smaller sum.

There are a few observations that help to understand the structure of a MST. For instance, if a graph has a vertex of degree 1, then the single edge incident to that vertex must be in the MST. If a graph has a vertex v of degree 2, then the lighter of the two edges incident to v (i.e., the one with smaller weight) must be in the MST. This can be proved by contradiction. Suppose that the two edges are e_1 and e_2, with the former having a smaller weight. If e_1 were not included in the MST, then v would be a leaf in the MST. So the purpose of e_2 would be solely to connect v to the rest of the vertices, via its other endpoint. But e_1 could also provide that function (via a different vertex of course). So we could remove e_2 and add e_1, and still have a spanning tree. But its weight would be smaller than what we started with, contradicting the claim that we had a MST. Note that this doesn't prove that e_2 shouldn't be in the MST; it only proves that e_1 must be in it.

The degree-2 example given above is a special case of a more general and powerful result about MSTs. What we were doing, essentially, was to isolate v from all other vertices, and find the best way to connect v to them. In general, we can play the same game by partitioning the vertex set, V, of G into two subsets. Suppose one subset is labeled A, and the remaining subset, $V-A$, is labeled B. There are three types of edges in G. Those that connect vertices within A, or within B, and finally those edges that connect a vertex from A to one in B. The claim is the following.
The lightest edge connecting A to B must be in the MST. Observe that this was true in the degree-2 case above, where A was just v. To see why the general claim is true, we need the concept of a **cut**. Visually, this is just an arbitrary loop that separates A from B. It can be visualized or drawn in many ways, but is easy to do, for any graph. No matter how we draw a cut, it will cross over every edge connecting A to B. There is always a way to redraw G so that the cut crosses only such edges. In fact, formally a cut is just a partition of the vertices into two subsets. The cut implicitly identifies all the edges between the subsets.

Now, for the sake of contradiction, suppose that we construct a MST without using the lightest edge as defined above. Let that lightest edge connect u in A to v in B. The edge uv is in G but not in the MST. Draw any cut, and follow the unique path in the MST from u towards v, until an edge is found that is intersected by the cut. Why must such an edge exist? The (connected) path starts in A and ends in B, so the two endpoints must be separated by the cut (loop). This separation must involve the cut crossing over one of the edges on the path. To make this more constructive, start traversing the path from u to v. At some point, you will have to find a vertex y in B. Let x be the preceding vertex on this walk; by definition x belongs to A. So, by definition, edge xy is in the MST, and it must also be crossed by the cut. Now remove xy from the proposed MST, and add uv. We said that uv is the lightest edge crossing the cut, so this swap will make the sum of weights decrease.

This contradicts the assumption that we had an MST to begin with. Therefore uv must be in the MST. Note that if xy and uv have the same weight, then this lemma just implies that one of the two must be in the MST.

Another nice property is that any subtree of an MST, spanning some subset A of vertices in G, must be the minimum spanning tree on A (using edges in G of course). Simply, if there was a better way to span A, we could just use that as part of the MST instead. The rest of the MST would not need to change, because connectivity through A would be maintained. Note that this applies to a subtree of the MST, not an arbitrary (possibly disconnected) subset.

Based on the above observations, we can come up with at least a couple of algorithms to construct a MST. For instance, we could look at each vertex, consider it as a trivial subset A, and conclude that the lightest edge incident to the vertex must be in the MST. After doing this, we would have some number of connected components. Each time we could pick some component and treat it as a subset, A. So each time we would just need to find the edges that connect A to the rest of the graph, and pick the lightest edge. This could be done in parallel, or by growing one component. Another algorithm could be to always add the lightest unused edge, as long as it didn't create a cycle. Not creating a cycle would imply merging two components, and that would be justified by an appropriate cut. Both of these ideas will be explored in the next two sections.

Kruskal's MST algorithm

Kruskal's algorithm is quite simple to describe and prove correct. Properly analyzing the time complexity is just a little trickier. The algorithm considers every edge for inclusion in the MST, in order of edge weights. So we start by sorting all edges according to weight. Let's assume that weights are distinct, to keep the description simpler. The algorithm and analysis are not really affected by this.

At all times, we maintain a *forest* of subtrees of the MST. This is initialized as a forest of all trivial subtrees (i.e., vertices) in the graph. Clearly every vertex will be in the MST. As the algorithm progresses, subtrees will merge, until only one tree remains.

Starting from the initial configuration of just the vertices, we begin considering edges, in the order mentioned above. For every edge e, we decide if it should be added to the MST. Edges that are added are never removed. The rule to decide is simple. If the endpoints of e belong to two different subtrees in the forest, or in other words if they belong to two components in the graph that we are growing, then e belongs in the MST. Otherwise, the two endpoints belong to the same component (or subtree), and we reject e. That's the algorithm. The proof of correctness is easy. If e links two vertices in the same component, then it completes a cycle, but it is also the heaviest edge on that cycle. The MST has no cycles, so one of the edges on that cycle has to be removed, and that should be the heaviest edge, which is e. (This can be proved easily, for instance using cuts). On the other hand, if e links two components, we can use our *cut* result, described in the previous section: Form a cut around one of the two components, separating it from all other components. We know that the lightest edge crossing the cut must be in the MST. That is precisely the edge e that we are currently considering. All other edges crossing the cut have not been considered yet, because they are heavier than e. Had there been a lighter edge crossing the cut, it would have joined two components in a previous iteration, thus contradicting the assumption that our cut surrounds one component. Also, any edge that has previously been rejected (because its two endpoints belonged to one component) can be drawn either within the cut loop or outside it. Or, in other words, we can draw the cut so that rejected edges do not cross it. So we don't have to worry about any lighter edge than e crossing the cut. Please note that cuts are abstract objects. Coordinates and drawings do not matter.

To recap, first the algorithm sorts all edges, which costs $O(E \log E)$ time. Then for every edge we have to decide if the two endpoints belong to the same component, and if they don't, we have to merge two components. One way to accomplish this is to give an extra label to every vertex, to tell us what component it belongs to. To start out, every vertex would have a different label, and progressively we would start forming duplicates. With such labels, testing the endpoints of an edge is trivially done in constant time. Merging components is also trivial, but what about the cost? If we don't analyze carefully, we will come to an incorrect conclusion, as follows. Changing the label of several vertices every time two components merge results in a cost of $O(V)$ per merge. Since this is done $O(E)$ times (once per edge), the total time complexity would become $O(VE)$. However, the first thing to notice is that even with E rounds, we actually only actually merge only $V-1$ times. This means that the cost is $O(V^2)$. In fact, a closer analysis will reduce this even further.

The “Union-Find” data structure is essentially a collection of linked lists, each one representing one component. The name of the structure signifies that we will be merging (or forming the union) of components, but also that we can find out what component any element belongs to, quickly. We give every element a pointer to one representative element in the linked list, and that representative is responsible for carrying a label that makes the component unique. Thus whenever we look at one of the elements, we find out its label by following a link to the representative. It’s important to note that this is an auxiliary data structure, and that we still want to use a primary structure (such as an adjacency list) to store the graph. The two structures are linked, so that when we move around from vertex to vertex in the primary structure, we can instantly locate where we are in the auxiliary structure. Now, given two vertices in the primary structure, we can instantly get to their “copies” in the Union-Find structure, and from there we can instantly tell if they belong to the same component, by looking at the corresponding representatives. The big question is, how do we merge two components? Well, each component is a linked list, so we can just append one to the other. The only problem is, we need one label for this component, and currently we have two representative labels. So one must be relabeled to the other. That means reassigning pointers for possibly many vertices. Note that we can’t work around this by just letting one representative point to the other. That would eventually mean that to determine the component of some vertex we would have to follow a path of representatives. So, one particular merge can cost $O(V)$: for instance, if we merge two components, each of size $V/2$. Heuristically, it makes sense to always relabel the smaller of two lists that must be merged. In fact it turns out that this is critical in getting a better time complexity. The reasoning is simple: now every vertex can only be relabeled $O(\log V)$ times. That is because every time a vertex v is relabeled, it means it belonged to a list that was at most half the size of the newly merged list. This new list would be at most half the size of some other list, if v is going to be relabeled again. So the size of the component that v belongs to will at least double every time v is relabeled, thus this can only happen $O(\log V)$ times before we get one component of size V. So we should maintain the size of each list, so that we can select the smaller one each time.Overall we get $O(V \log V)$ time for V unions, even though one union can cost $O(V)$. (this is an amortization result)

All together, it costs $O(1)$ to check the endpoints of any edge, and thus $O(E)$ for all such checks. All of the work required to maintain the data structure that permits that constant time lookup amounts to $O(V \log V)$. The bottleneck of this algorithm is in fact sorting the edges in the very first step. In other words, $O(E \log E)$ is at least expensive as $O(V \log V)$, because $E \geq V - 1$ for the graph that we are assuming is connected. Finally, we can rewrite $O(E \log E)$ as $O(E \log V^2) = O(E \log V)$, which is the standard way of describing the time complexity of this algorithm.

Notes: the time complexity of Kruskal’s algorithm is much smaller if we have pre-sorted edge weights (for whatever reason). In that case, we just need to scan through the list, and then merge components, all in $O(E + V \log V)$ time. Also, there is a fancier data structure that gives a better time complexity, but that is beyond the scope of this class. Details are easily found in CLRS.

Prim's MST algorithm

Prim's algorithm grows a single spanning tree, unlike Kruskal's that grows several subtrees in parallel. We start with a trivial tree consisting of just one vertex. Again, we can rely on the concept of a cut to greedily add the next edge. Given a current tree, we form a cut around it. The lightest edge crossing the cut is the next one to join the tree. Once an edge (meaning also a new vertex) is added, it is never removed. Clearly we should never add an edge that links two vertices already in the tree. This is actually something we won't need to worry about, because at every iteration we will make sure to be adding a new vertex to the tree.

In general, we will have three types of edges; those in the current spanning tree, those joining a vertex in the tree to a vertex outside, and those not touching the tree at all. Clearly, to correctly add a new edge to the tree, we need to be looking at the second type of edge. In fact it is precisely these edges that cross the cut around the current tree. The next edge to add should be the lightest of all such edges, that join a vertex in the tree to one outside. Instead of maintaining a sorted order of such edges, or at least the lightest edge in this group, the algorithm maintains a set of vertices that are not in the tree. The lightest edge is found by actually finding a "cheapest" vertex. This means that we want to assign a score to every vertex, such that the vertex with the lowest score is the one that should be added to the tree. The score of every vertex will just be the weight of the lightest edge that we could use to add the vertex directly to the tree. This means that vertices belonging only to edges that don't touch the tree will have a score of infinity. Vertices in the tree already are not considered. It is only those vertices that are one "hop" away from the tree that can have finite scores and thus be real candidates.

To start things out, we give every vertex a score of infinity, except for the arbitrary vertex that is used as a seed (or root) to grow the tree. As soon as we choose that vertex, we give all of its neighbors a finite score, based on the edge linking them to the root. Now that we have vertices with finite scores, we pick the cheapest one to add to the tree. Notice that in the very first step, this means that we pick the lightest edge crossing a cut that surrounds the root vertex. Thinking about cuts can quickly provide a proof, or at least intuition, for lots of these algorithms and their variants.

In general, we will have a current tree, and the vertex (outside the tree) with the lowest score will have some edge with that weight linking it to the tree. That edge will be the lightest crossing a cut, so the vertex and the edge can be safely added. So in general the algorithm and proof of correctness are at the same basic level as Kruskal's algorithm. Once again the interesting part is to establish a time complexity. What we've said we need to do is find a vertex with minimum score. We can do this by maintaining all vertex scores in a min-heap. This will be an auxiliary data structure, that we can directly link to whatever primary data structure we are using to store the graph. This means that given a vertex in the primary structure, we can directly access its copy in the auxiliary structure. So, to determine what vertex we should add to the tree, we simply look at the root of the heap. To determine what edge to add, we simply look at all edges incident to the vertex (and not in the tree already), and add the lightest one, that will have a weight equal to the score of the vertex. Note that we find all such edges in the primary structure (such as an adjacency list), and we can directly access the vertex in that structure starting from the heap, because we have pointers between the two structures. The primary cost of adding a vertex v is thus $O(1)$ for accessing the top of the heap, $\Theta(degree(v))$ for looking at all neighbors of v assuming an adjacency list for a primary structure, and $O(\log V)$ for rebuilding the heap once we extract v. Over all vertices, these primary costs will add up to $O(V)$, $\Theta(E)$, and $O(V \log V)$ correspondingly. Notice that the sum of all degrees is $\Theta(E)$, because we are simply double-counting every edge. However, we are not quite done with v yet. Once we add v to the growing tree, some vertices will have their scores reduced. For instance, if any vertices previously not adjacent to the tree become adjacent (i.e., because they are neighbors of v), then their scores will become finite. But it is also possible that a vertex with a finite score will need to have it reduced because of the new incorporation of v into the tree. In other words, a vertex might already be adjacent to the tree (one hop away), but now it has a better alternative to be joined to the tree, via v). All of these vertex score updates can be done by looking at the neighborhood of v. We simply look at each of its neighbors, and for those outside of the tree we compare the current score with the weight of the edge linking to v. That costs $O(1)$ per neighbor of v, so $\Theta(degree(v))$, and over all vertices it is $\Theta(E)$. But we are still not done! Every time we decrease a score, we need to reflect this in our min-heap. This means performing a decrease-key operation on a node in the heap, which means the node might bubble up to the top, with a cost of $O(\log V)$. Every vertex might have its score decreased once per neighbor, so the total number of decrease-keys is $O(E)$. Thus in the worst case this algorithm runs in $O(E \log V)$ time.

The previous analysis assumed the use of an adjacency list as a primary data structure to store the graph. This is intuitive. After all, for every vertex we had at least two reasons to ask for all of its neighbors, and we know that for such a query it is better to use an adjacency list than a matrix. However, for dense graphs, we know that we can actually afford to use a matrix. It turns out that in such cases we can improve the time complexity without even using an auxiliary structure like a min-heap. First of all, recall that a dense graph has $\Theta(V^2)$ edges, so the previous implementation would cost $O(V^2 \log V)$. By just using the primary structure, accessing all neighbors of v would cost $\Theta(V)$ (for a matrix) or $\Theta(degree(v))$ (for an adjacency list). Thus for all vertices the corresponding costs would be $\Theta(V^2)$ or $\Theta(E) = O(V^2)$. Retrieving the vertex of minimum score would take a whopping $\Theta(V)$ time instead of $O(1)$, but this isn't a bottleneck; over all vertices this would cost $O(V^2)$. So, quite simply, the total cost is $O(V^2)$, which beats the min-heap implementation. Of course, for slightly less dense graphs we would still want to use only the primary structure, and at some level of edge density there is a tradeoff.

Finally, just as with Kruskal's algorithm, Prim's algorithm can benefit from the use of a fancier data structure instead of a regular binary heap, but this is beyond the scope of this course. Details are easily found in CLRS, or in the course that follows this one.

Single Source Shortest Paths: intro

The SSSP problem has the following input: a directed weighted graph and a source vertex, s. The goal is to output all "shortest" paths from s to every other vertex. By "shortest" we mean "minimum sum of weights". BFS produces all shortest paths, when all weights are equal to 1.

For a particular target vertex, t, the best path from s to t is not affected by other paths from s to other vertices, although paths can have common parts. So essentially we are solving $V-1$ problems in parallel. Last time I checked, nobody knows how to produce the shortest path from s to a given vertex t in time faster than what it takes to produce all shortest paths from s. The SSSP problem is still well defined on undirected graphs, but we usually just work with directed ones. After all, an undirected graph can be converted to a directed graph by doubling edges. Note that between two vertices there can be multiple paths that are optimal. All we care about is finding one such path.

For efficiency, we express the output in the form of a *shortest paths tree*, of size $\Theta(V)$, rather than as a list of paths that could have total space complexity $O(V^2)$. So each vertex will need to have a pointer to a parent vertex. A big assumption here is that no shortest path contains a cycle. It is clear that if all weights are positive then using a cycle is wasteful. But with negative weights, cycles can potentially be useful to reduce the cost of a path. Such cycles, with a negative total score, are called *negative cycles*. Some applications and/or algorithms assume no negative edges, or no negative cycles, and others make no assumptions (and can in fact detect negative cycles).

It is not difficult to see that any particular shortest path P, from s to some vertex t, must have optimal substructure. In other words, if there is a vertex v on P, then P must contain a shortest path from s to v, and the shortest path from v to t. Otherwise we would be able to improve P. This means that if we want to compute the shortest path to some vertex t, we could compute the shortest path to every neighbor of t (incident to an edge directed into t), then add the corresponding extra edge (weight) to each such solution, and pick the best option. This works really well when there is a particular order in the graph, for instance when it is a DAG. On a DAG, we can compute scores of vertices in any topological sorted order. Then when we compute the score of t, we just need the scores of ancestral neighbors, and the actual computation for t won't affect those scores again. In other words, we "sweep" through the vertices in order and never have to look back. In general graphs though, it is conceivable that updating the score of one vertex will affect the score of another but then this could go back and forth. At the very least, it is more difficult to figure out in what order to process vertices.

Relaxing – The process of updating the score of a vertex y by checking the usage of one of its incoming edges, xy, is part of the main algorithms for SSSP. When doing this, we say that we "relax edge xy". All this means is that we compare the current score of y to the sum of the weight of xy plus the score of x. It might be that the current best path from s to y doesn't use x, and we want to see if going through x is better. Or, it could be that the best path already goes through x, but we are checking to see if we have recently found a better way from s to x (in other words we are checking if the score of x has decreased since the last time y was updated). Relaxing an edge clearly takes $O(1)$ time (add, compare, update).

SSSP: The Bellman-Ford algorithm

This algorithm is very simple, but relies on an interesting lemma involving relaxing edges, that will be presented first.

Relaxing Lemma: Let P be a shortest path between the source s and some vertex t. Suppose that P has k edges, $e_1, \ldots, e_k$. If we relax e_1 before e_2, before e_3, ..., before e_k, then we will have computed the shortest path score for every vertex on P.

The proof is by induction. If we have relaxed in the correct order up to some edge $e_j = xy$, then by the inductive hypothesis we have the score for vertex y. When we relax edge $e_{j+1} = yz$, we get the score for vertex z, because we already know the optimal score from s to y, and yz is part of the optimal path P which means it is the optimal way to get from y to z (recall the optimal substructure claim). Thus we add the two to get the score of z. The base case is trivial.

In other words, when we relax the first edge on P, we get the optimal score to the first vertex (not counting s). Then when we relax the second edge, we get the optimal score to the second vertex, and so on. We rely on the fact that every edge xy on P is locally optimal, meaning that it gives the best way to get from x to y (otherwise P wouldn't be optimal). We also rely on the fact that any optimal path consists of optimal subpaths.

The Relaxing Lemma may seem a bit strange, in the sense that it relies on an optimal path P, when this is what we are trying to find in the first place. All it says though is that if P exists and we happen to relax its edges in a particular order, then the all the scores of vertices on P will be known. In fact we will also get all the parent links that will form part of a SSSP tree. Note that the lemma requires relaxing edges in the order of P but it says nothing about relaxing other edges of the graph in between, or even relaxing the edges of P itself multiple times in the wrong order. The Lemma still holds, as long as the edges of P are relaxed as an ordered subsequence in any sequence of relaxations.

Now we can describe the Bellman-Ford algorithm. First it sets the score of s to zero, and all other scores to infinity. It consists of $V-1$ iterations, and in each iteration it relaxes *all* edges in the input graph, in arbitrary order. That's all. The time complexity is $O(VE)$, because each round takes $O(E)$ time, given that relaxing an edge takes $O(1)$ time.

Why does this work? For every vertex v_i, there is a shortest path P_i from s to v_i, and the following will be true. In iteration j, the j-th edge on P_i will be relaxed.

In other words, in the first iteration of the algorithm, the first edge on every shortest path will be relaxed. In the second iteration, the second edge on every shortest path will be relaxed, and so on. This is guaranteed because we relax *all* edges of the graph in each iteration. So we guarantee the conditions required by the Relaxing Lemma. The reason we only need $V-1$ iterations is that no path can have a longer length, if there are no (negative) cycles. Without negative cycles, after $V-1$ iterations, all scores must have stabilized. If they haven't, then we know there is a negative cycle somewhere. Nothing in the proof prohibits the existence of negative edges though.

SSSP: Dijkstra's algorithm

Dijkstra's algorithm only works with non-negative edge weights. This algorithm is remarkably similar to Prim's MST algorithm. It builds a SSSP tree from s, starting with the trivial tree consisting of s itself. As with Prim, if the graph is in an adjacency list, we use a min-heap as an auxiliary structure to store all vertices not in the tree. We initialize by placing all vertices in the heap, then set the score of s to zero, and every other score to infinity. Again as with Prim, we alternate between extracting the min-score vertex x from the heap, figuring out which edge gave x its score, adding both to the tree, and updating scores of neighbors of x. That last step means relaxing all outgoing edges from x (although we only care about the ones leading to vertices not in the tree). Of course, the first extraction is just s so there are no edges to add.

The proof of correctness relies on a cut (as with Prim). If we cut around the current SSSP tree, we can partition all (directed) edges of the graph as follows. Some are in the tree, within the cut. Some don't touch the tree at all, and they are outside the cut. Their endpoints still have infinite scores. Finally there are edges that cross the cut. Among these, we only care about the edges that cross from the tree towards the exterior of the cut. On such edges, the endpoints (vertices) on the outside must have finite scores, because the endpoints inside have already relaxed all their outgoing edges, given that they are in the tree. Note that edges crossing the cut from outside to inside can be ignored because they can never be part of a SSSP: they lead to vertices that already have a shortest path from s that is final.

Given the above observations, the next vertex x to be extracted from the min-heap is found just across the cut, adjacent to the current tree. The extracted vertex x has the shortest path from s, among all vertices not in the tree yet. We can safely add x to the tree, along with the edge that we relaxed to give x its score, because no better path to x will ever be found. Any other path from s will have to cross the cut somewhere else (through some other vertex q), and then continue from q to x. But we have just said that crossing the cut to x is better than crossing the cut to q, in the sense that the path from s to x is cheaper than the path from s to q. Thus choosing to reach from s to x via q can only be worse than our current option. All of this assumes no negative edges exist, otherwise all bets are off.

Notice that the only real difference from Prim's algorithm is that here the choice of vertex depends on the entire path from the source to vertices just beyond the cut. In Prim's, the choice depends on the lightest edge crossing the cut. In some sense, Dijkstra is like Prim but considering history. Fortunately this history is stored at the vertices on the border of the cut. The time complexity of this algorithm is identical to Prim's, if we use a min-heap. We get $O(E \log V)$ with an adjacency list and a heap. Or, $O(V^2)$ with an adjacency matrix.

Once again, with a fancier heap, these time complexities can be improved. See CLRS or the followup course.

NP-completeness

This is a massive topic. A page or two in summary cannot do justice. I will just mention some **very** informal ideas here.

For a problem to be *in the NP complexity class*, it should be a decision problem (i.e., the output is a yes or a no), and any proposed solution should be verifiable in polynomial time. For example, "Is it possible to find a cycle in this graph, such that every vertex is visited precisely once?". The answer will either be Yes or No, and if you were to propose some cycle, it would be trivial to verify in polynomial time that it's actually part of the graph (and actually visits every vertex precisely once).
NP stands for "non-deterministic polynomial (time)". A subset of NP is the set P of all decision problems that have polynomial-time solutions. So not only can we verify proposed solutions, we can actually solve from scratch in polynomial time. NP contains another subset of great importance: the class of **NP-complete problems** (NPC). Each such problem is at least as hard as any problem in NP. What this means is that if anyone ever solves an NPC problem in polynomial time, then there must be a polynomial-time solution for every problem in NP (because nothing is harder), which means that all NP problems would actually belong in P, hence P=NP. On the other hand, if anyone ever proves that an NPC problem has no polynomial-time solution, then neither can any other NPC problem, because by definition they are at least as hard. So essentially all NPC problems are at least as hard as each other, and have a common fate. When we say "at least as hard", we allow some flexibility, as we do with Θ notation where we allow multiplicative constants (e.g., n^2 is considered to be about the same as $5n^2$). When dealing with P vs NP, we allow polynomial flexibility. So any two algorithms in P are in some sense equivalent, because they both have polynomial-time solutions (e.g., we consider n^2 and n^{50} to be about the same). Two problems in NPC are "equivalent" in the sense that if we knew how to solve one, then we could solve the other with some extra (multiplicative or additive) polynomial-time overhead. At the moment, we don't know how to solve any NPC problem in polynomial time, so this overhead is insignificant.

So far we have focused on decision problems. There are other "hard" problems as well, but they don't get to be in the NP class. In fact many decision problems have optimization versions. For instance, "does this graph have a path of length at least k?" is a decision problem, and the optimization version is "in this graph, find the length of the longest path". Clearly, the ability to solve this optimization problem implies ability to solve the decision problem. Optimization problems are at least as hard as the decision counterparts, but sometimes not much harder at all. For instance, sometimes we can just binary search on the parameter of a decision problem, to optimize.

The class of **NP-hard** problems contains problems that are at least as hard as anything in NP. That means that NPC problems are in fact NP-hard, by definition. On the other hand there are NP-hard problems that are not in NP and thus are not NPC. Again, NPC problems are, by definition, NP-hard problems that are in NP. Thus there are two reasons for a problem, X, to be NP-hard but not NPC: either X is not a decision problem, or we don't know how to verify proposed solutions for X in polynomial time.

Because NP-hard problems are at least as hard as any problem in NP, an efficient solution to an NP-hard problem implies that P=NP. (The same fascinating property that we established for NPC problems).

It's easy to accept that there are decision problems with efficiently verifiable solutions, but no efficient solutions (yet). These are simply problems that are in NP but not in P. But why are there NPC problems? In other words, what gives them that extra status of being at least as hard as any problem in NP? That is a long story, but in a nutshell we can think of all the NPC problems as vertices in a directed graph, where a directed edge from x to y tells us that x is at least as hard as y. For every problem to be as hard as every other one, we need every problem (vertex) to be able to reach every other one along a directed path in this graph. Fortunately, there exists a problem (called circuit-satisfiability) known to be at least as hard as every problem in NP, so half the work is done. (For proper details, please take a complexity theory course). That means that every time we think that we are dealing with a hard problem, x, we just need to show that it is at least as hard as some other problem y, already known to be hard. The existing problem y can trace all the way back to the "root", and because the root is automatically harder than x we will have a cycle.

To show that one problem x is at least as hard as another, y, we perform a **reduction**. Reductions are not restricted to NPC problems. The idea is as follows. Take a problem y that has a known lower bound, or a suspected lower bound (for instance NPC). Figure out how to efficiently (i.e., faster than the lower bound) transform the input for y into input for x. Also figure out how to map the output of x to output of y, efficiently. Then y can be solved by making these transformations and solving x. Given that the transformations are efficient, there must be something else that slows down this particular solution of y. The only culprit can be the step of solving x. Thus solving x is the bottleneck, and x is at least as hard as y. Notice that x could be way harder than y, or maybe y is also at least as hard as x. With a reduction, we don't get this information. All we know is that it x at least as hard as y. This is a one-way conclusion. Of course, if x ends up with an efficient solution, then via this reduction so does y. So if y was NPC, a reduction as above will show that x is NP-hard. Furthermore if x is ever solved in polynomial time then so is y (and in turn this will imply that P=NP).

To recap, we take a known NPC problem, show how to transform its input into a new problem in polynomial time, do the same for the output (decision), and this means that the new problem must be NP-hard. Why just NP-hard? Because all we've shown is that it is at least as hard as an NPC-problem, which by definition is at least as hard as everything in NP. To conclude that the new problem is in fact NPC, first of all it must be a decision problem, but also we would have to show how to verify any proposed solution in polynomial time, i.e., show that it is in NP. Remember, NPC means NP-hard and in NP.