

wk4



# 0wk4-Overview

Part VIII --- LINEAR-TIME SELECTION (Required). These lectures study the problem of computing the ith smallest element of an input array (e.g., the median). It's easy to solve this problem in O(n log n) time using sorting, but we can do better! The required material goes over a super-practical randomized algorithm, very much in the spirit of QuickSort, that has \*linear\* expected running time. Don't forget that it takes linear time just to read the input! The analysis is somewhat different than what we studied for QuickSort, but is equally slick. Basically, there's a cool way to think about the progress the algorithm makes in terms of a simple coin-flipping experiment. Linearity of expectation (yes, it's back...) seals the deal.

Part VIII --- LINEAR-TIME SELECTION (Optional). I couldn't resist covering some advanced related material. The first is an algorithm that has more Turing Award-winning authors than any other algorithm that I know of. It is a deterministic (i.e., no randomization allowed) linear-time algorithm for the Selection problem, based on an ingenious "median-of-medians" idea for guaranteeing good pivot choices. (There are some accompanying lectures notes for this part, available for download underneath each video.) The second optional topic answers the question "can we do better?" for sorting, unfortunately in the negative. That is, a counting argument shows that there is no "comparison-based" sorting algorithm (like MergeSort, QuickSort, or HeapSort) with worst-case running time better than n log n.

Part IX --- GRAPHS AND THE CONTRACTION ALGORITHM. The second set of lectures for this week is a segue from randomized algorithms to graph algorithms. We first review graphs and the most standard ways of representing them (most commonly, by adjacency lists). We then consider the random contraction algorithm, discovered by Karger "only" 20ish years ago (while a PhD student here at Stanford). This algorithm solves the minimum cut problem --- given an undirected graph, separate the vertices into two non-empty groups to minimize the number of "crossing edges". Such problems come up when reasoning about, for example, physical networks, social networks, and images. This algorithm was perhaps the first strong evidence that graph problems could be added to the long list of "killer applications" of random sampling. Don't tune out before the final plot twist --- a simple but useful trick for transforming an algorithm that almost always fails into one that almost always succeeds.

HOMEWORK: Problem Set #4 has five questions about the randomized selection algorithm, cuts in graphs, and the contraction algorithm. Programming Assignment #4 asks you to implement the contraction algorithm and use it to compute the min cut of the graph that we provide.

SUGGESTED READINGS FOR WEEK 3:  
CLRS Chapter 9, 22 (Only 22.1)  
DPV Chapter 3 (only 3.1)  
KT Chapter 13, Sections 13.2,13.5  
SW Chapter 4, Section 4.1

# 0wk4-menu

VIII. LINEAR-TIME SELECTION  
Week 4 Overview   
Randomized Selection - Algorithm

Randomized Selection - Analysis   
Deterministic Selection - Algorithm [Advanced - Optional]   
Deterministic Selection - Analysis I [Advanced - Optional]   
Deterministic Selection - Analysis II [Advanced - Optional]  
Omega(n log n) Lower Bound for Comparison-Based Sorting [Advanced - Optional]   
IX. GRAPHS AND THE CONTRACTION ALGORITHM  
Graphs and Minimum Cuts15 分  
Graph Representations14 分  
Random Contraction Algorithm8 分  
Analysis of Contraction Algorithm30 分  
Counting Minimum Cuts7 分  
Problem Set #4  
测验: Problem Set #45 个试题

截止时间 六月 18, 11:59 晚上 PDT  
Optional Theory Problems (Batch #2)10 分  
Programming Assignment #4  
测验: Programming Assignment #41 个试题  
截止时间 六月 18, 11:59 晚上 PDT  
Final Exam (1 attempt per 24 hours)  
Info and FAQ for final exam10 分

测验: Final Exam10 个试题

截止时间 六月 18, 11:59 晚上 PDT

# 8-1-Randomized Selection - Algorithm

I've said pretty much everything I want to say about sorting at this point but I do want to cover one more related topic. Namely the selection problem. This is a problem of computing ordered statistics of an array with computing the median of an array being a special case. Analogous to our coverage of quick sort the goal is going to be the design and analysis of a super practical randomized algorithm that solves the problem. And this time, we'll even achieve an expected running time that is linear in the length of the input array. That is big O of n for input arrays of length n, as opposed to the o of n log in time that we had for the expected running time of quick sort. Like quick sort, the mathematical analysis is also going to be quite elegant. So in addition these two required videos on this very practical algorithm will motivate two optional videos that are on very cool topics but of a similar more theoretical nature. The first optional video is going to be on how you solve the selection problem in deterministic linear time. That is without using randomization. And the second optional video will be a sorting lower bound that is why no comparison based sort can be better than mergeshort. Can have better running time than big O of n login. So a few words about what you should have fresh in your mind before you watch this video. I have definitely assuming that you watched quicksort videos. And not just watched them but that you have that material pretty fresh in your mind. So in particular the video of quicksort about the partition subroutine, so this is where you take a input ray and you choose a pivot and you do repeated swaps. You rearrange the array so that everything less then the pivot is to the left of it. Everything bigger then the pivot is to the right of it. You should remember that sub routine, you should also remember the previous discussion about pivot choices. The idea that the quality of a pivot depends on how balanced a split into two different sub problems it gives you. Those are both going to be important. For the analysis of this randomized linear time selection algorithm I need you to remember the concepts from probability review part one. And particular random variables, their expectation, and linearity of expectation. That said, let's move on and formally define what the selection problem is. The input is the same as for the sorting problem, just you're giving it array of indistinct entries. But in addition, you're told what order statistic you're looking for. So that's going to be a number I, which an integer between 1 and N. And the goal is to output just a single number. Namely the ith order statistic, that is the ith smallest entry in this input array. So just to be clear, if you had an array entry of let's just say 4 elements, containing the numbers 10, 8, 2 and 4. And you were looking for, let's say, the 3rd or a statistic that would be this 8. The first order statistic is just the minimum element of the array. That's easier to find with a linear scan. The nth order statistic is just the maximum, again easier, easy to find with a linear scan. The middle element is the median. You should think of that as the canonical version of the selection problem. Now when n is odd, it's obvious what the median is, that's just the middle element, so the n plus one over 2th order statistic. If the array has even length, there's two possible medians, so let's just take the smaller of them, that's the n over 2th order statistic. You might wonder why you'd ever want to compute the median of an array rather than the mean, that is the average. It's easy to see you that you can compute the average just with a simple linear scan. And the median you can, one motivation is it's a more robust version of the mean. So if you just have a data entry problem and it corrupts one element of an input array it can totally screw up the average value of the array, but it has generally very little impact on the median. Final comment about the problem is that I am going to assume that the array entries are distinct, that is there's no repeated elements. But just like in our discussions of sorting, this is not a big assumption. I can encourage you to think about how to adapt these algorithms to work even if the arrays do have duplicate. You can, indeed, still get the same very practical, very fast algorithms with duplicate elements. Now if you think about it, we already have a pretty darn good algorithm that solves the selection problem. Here's the algorithm. It's two simple steps and it runs in o of n log n time. Step one, sort the input array. We have various subroutines to do that. Let's say we pick MergeSort. Now, what is it we're trying to do? We're trying to the ith smallest element of the input array. Well, once we've sorted it we certainly know where the ith smallest element is, it's in the ith position of the sorted array. So that's pretty cool, we've just done what a computer scientist would call a reduction and that's a super useful and super fundamental concept. It's when you realize that you can solve one problem by reducing it to another problem that you already know how to solve. So what we just showed is that the selection problem reduces easily to the sorting problem. We already know how to solve the sorting problem n log n time so that gives an n log n time solution to this selection problem. But again remember the mantra of any algorithm designer worth their salt, is can we do better. We should avoid contentedness. Just because we got nlogn we should stop there. Maybe can be even faster. Now certainly we're going to have to look at all the elements in the input array, in the worst case. You shouldn't expect to do better than linear, but hey, why not linear time? Actually if you think about it, we probably should have asked that question back when we were studying the sorting problem. Why were we so content with the end login time bound for merch sort. And the O of N login time on average bound, for quick sort. Well it turns out, we have a really good reason to be happy with our N login upper bounds for the sorting problem. It turns out and this is not obvious, and will be the subject of the optional video. You actually can't sort an input array of length N better than N log n time. Either in the worst case or an average. So another words, if we insist on solving the selection problem via a reduction to the sorting problem then we're stuck with this N log N time bound. Okay, strictly speaking that's for something called comparison sorts, see the video for more details but the upshot is if you want a general purpose algorithm. And we want to do better than N log N for selection we have to do it using ingenuity beyond this reduction, we have to prove that selection is a strictly easier problem then sort it. That's the only way we're going to have an algorithm that beats n log n. That's the only way we can conceivably get a linear time algorithm. And that is exactly what is up next on our plates. We're going to show selection is indeed fundamentally easier than sorting. We can have a linear time algorithm for it, even though we can't get a linear time algorithm for sorting. You can think of the algorithm we're going to discuss as a modification of quick sort and in the same spirit of quick sort it will be a randomized algorithm. And the running time will be an expected running time that will hold for any input array. Now, for the sorting problem we know that quick sort that's n log in time on average, where the average is over the coin flips done by the code. But we also know that if we wanted to, we could get a sorting algorithm in n log n time that doesn't use randomization. The merge sort algorithm is one such solution. So here, we're giving a linear time solution for selection, for finding order statistics that uses randomization. And it would be natural to wonder, is there an analog to merge sort? Is there an algorithm which does not use randomization, and gets this exact same linear time down. In fact there is. The algorithm's a little more complicated, and therefore not quite as practical as this randomized algorithm. But it's still very cool. It's a really fun algorithm to learn and to teach. So I will have an optional video about linear time selection without randomization. So for those of you who aren't going to watch that video or want to know what's the key idea. The idea is to choose the pivot deterministically in a very careful way using a trick called the median of medians. That's all I'm going to say about it now you should watch the optional video if you want more details. I do feel compelled to warn you that if you're going to actually implemented a selection algorithm. You should do the one that we discuss in this video, not the linear time one. because the one we'll discuss in this video has both smaller constants and works in place. So what I want to do next is develop the idea that can modify the QuickSort paradigm in order to directly solve The selection problem. So to get an idea of how that works, let me review the Partition subroutine. Like in Quicksort this subroutine will be our workhorse for the selection algorithm. So, what the Partition subroutine does, it takes as inputs, some jumbled up array and it's going to solve a problem which is much more modest than sorting. So in partitioning, it's going to first choose a pivot element somehow. We'll have to discuss what's a good strategy for choosing a pivot element. But suppose in this particular input array it chooses the first element, this three, as the pivot element, the responsibility of the partition sub-routine then is to rearrange the elements in this array so that the following properties are satisfied. Anything less than the pivot is to the left of it and it can be in jumbled order. But if you're less than pivot you better be to the left like this two and one is less than three. If you're bigger than the pivot than again you can be in jumbled order amongst those elements but all of them have to be to the right of the pivot and that's true for the numbers four through eight. They all are to the right of the pivot three in a jumbled order. So this in particular puts the pivot in its rightful position, where it will belong in the final sorted array. And at least for Quicksort, it enabled us to recursively sort to smaller subproblems. So this is where I want you to think a little bit about how we should adapt this paradigm. So, suppose I told you the first step of our selection algorithm is going to be choose a pivot and partition the array. Now the question is, how are we going to recurse? We need to understand how to find the ith order statistic of the original input array. It suffices to recurse on just one sub problem of smaller size, and find a suitable or a statistic in it. So how should we do that? Let me ask you that with some very concrete examples. About what pivot we choose and what order statistic we're looking for and see what you think. So the correct action to this quiz is the second answer. So we can get away with recursing just once, and then this particular example, we're going to recurse on the right side of the array. And instead of looking for the fifth order statistic like we would originally, we're going to recursively search for the second order statistic. So why is that? Well first why do we recurse on the right side of the array? So by assumption we have this array of ten elements, we choose the pivot, we do partitioning, remember the pivot winds up in its rightful position. That's what partitioning does. So in the bid it winds up in the third position, we know it's the third smallest element in the array. Now that's not what we were looking for. We were looking for the fifth smallest element in the array. That, of course, is bigger than the third smallest element of the array. So by partitioning, where is the fifth element going to be? It's gotta be to the right of this third smallest element, to the right of the pivot. So we know for sure that the fifth order statistic of the original array lies to the right of the pivot. That is guaranteed. So we know where to recurse on the right hand side. Now, what are we looking for? We are no longer looking for the fifth order statistic, the fifth smallest element. Why? Well we've thrown out both the pivot and everything smaller than it. Remember we're only recursing on the right hand side. So we've thrown out the pivot, the hird element, and everything less than it, the minimum and the second minimum. Having deleted the three smallest elements and originally looking for the fifth smallest of what remains, of what we're recursing on. We're looking for the second smallest element. So the selection algorithm in general, is just the generalization of this idea. So arbitrary arrays and arbitrary situations of whether the pivot comes back equal to less or bigger than the element you are looking for. So let me be more precise, I am going to call this algorithm R select for randomized selection, and according to the problem definition it takes as input, as usual an array A of some length n. Then also the order statistic that we are looking for, so we are going to call that i, and of course we assume that i is some integer between one and inclusive. So for the base case, that is going to be if the array has size one, then the only element we could be looking for is the oneth order statistic and we just return the sole element of the array. Now we have to partition the array around the pivot element. And just like in quick sort, we're going to very lazy about choosing the pivot. We're going to choose it uniformly at random from the n possibilities, and hope things work out. And that will be the crux of the analysis, proving that random pivots are good enough sufficiently often. Having chosen the pivot, we now just invoke the standard partitioning and subroutine. As usual, that's going to give us the partitioned array. You'll have the pivot element, you'll have everything less in the pivot to the left, everything bigger, to the right. As usual, I'll call everything to the left, the first parts of the partitioned array. And everything bigger, the second part. Now we have a couple of cases, depending on whether the pivot is bigger or less then the element we are looking for. So I need a little notation to talk about that. So let's let j be the order statistic that p is. So if p winds up being the third smallest element like in the quiz, then j's going to be equal to three. Equivalently we can think of j as defined as the physician of the pivot in the partition version of the array. Now there's one case, which is very unlikely to occur, but we should include it just for completeness. If we're really lucky, then, in fact, a random pivot just happens to be the order statistic we were looking for. That's when i equals j. We're looking for the ith smallest element. If by dumb luck the pivot winds up being the ith smallest element, we're done. We can just return it. We don't have to recurse. Now in general of course, we don't randomly choose the element we are looking for. We choose something that, that could be bigger or could be smaller then it. In the quiz we chose a pivot that was smaller then what we were looking for. Actually, that's the harder case. So, let's first start with a case, where the pivot winds up being bigger then the element we were looking for. So that means that j is bigger than i. We're looking for the i smallest. We randomly chose the j smallest for j bigger than i. So this is the opposite case of the quiz. This is where we know what we're looking for has to be to the left of the pivot. The pivot's the j smallest everything less than is to the left. We're looking for the i smallest, i is less than j, so that's got to be on the left. That's where it recurs. Moreover it clear we're looking for exactly the same order statistic. If we're looking for the third smallest element, we're only throwing out stuff which is bigger than something even bigger hthan the third smallest element so we're still looking for the third smallest of what remains. And naturally the new array size is j minus 1 because that's what's to the left of the pivot. And then finally, the final case is when the random element that we choose is less than what we're looking for and then we're just like the quiz. Namely what we're looking for is bigger than the pivot. It's got to be in the right-hand side. We know we've got a recurse in the right-hand side. Whenever the right-hand side has n minus j elements, we throw out everything up to the pivot. So we throw out j things. There's n minus j left. All of those j things we threw out are less than what we're looking for. So if we used to be looking for the i smallest element now we're looking for the i minus j smallest element. So that is the whole algorithm. That is how we adopt the approach we took to the sorting problem in quick sort and adapt it to the problem of selection. So, is this algorithm any good? Let's start studying its properties and understand how well it works. So let's begin with correctness. So the claim is that, no matter how the algorithm's coin flips come up, no matter what random pivots we choose, the algorithm is correct. In the sense that it's guaranteed to output the ith order statistic. The proof is by induction. It proceeds very similarly to quick sort. So I'm not going to give it here. If you're curious about how these proofs go, there's an optional video about the correctness of quick sort. If you watch that and understand it, it should be clear how to adapt that inductive argument to apply to this select algorithm as well. So as usual for divide and conquer algorithms, the interesting part is not so much knowing, understanding why the algorithm works, but rather understanding how fast it runs. So the big question is, what is the running time of this selection algorithm? Now, to understand this we have to understand the ramifications of pivot choices on the running time. So you've seen the QuickSort videos they're fresh in your mind so what should be clear is that just like in QuickSort how fast this algorithm runs is going to depend on how good the pivots are and what good pivots means is pivots that guarantee a balanced split. So, the next quiz, we'll make sure that you understand this point and ask you to think about just how bad the running time of the selection algorithm could be if you get extremely unlucky in your pivot choices. So the correct answer to this question is exactly the same as the answer for QuickSort. The worst case running time, if the pivots are chosen just in a really unlucky way. Is actually quadratic in the array length. Remember, we're shooting for linear time. So this quadratic is a total disaster. So how could this happen? Well suppose you're looking for the median, and suppose you choose the minimum element as the pivot every single time. So if this is what happens, if every time you choose a pivot to be the minimum, just like in QuickSort, this means every time you recurse, all you succeed in doing is peeling off a single element from the input array. Now, you're not going to find the median element until you've roughly n over 2 recursive cause, each on an array that has size at least a constant fraction of the original one. So it's a linear number of recursive calls, each on an array of size at least some constant times n. So that gives you a total running time of quadratic overall. Of course, this is an absurdly unlikely event. Frankly, your computer is more likely to be struck by a meteor than it is for the pivot to be chosen as the minimum element in every recursive call. But if you really have an absolutely worst case choice of pivots, it would give this quadratic run time down. So the upshot then is that the running time of this randomized selection algorithm depends on how good our pivots are. And for a worse case chose of pivots the running time can be as large as m squared. Now hopefully most of the time we're going to have much better pivots. So the analysis receives by making that idea precise. So the key to a fast running time is going to be the usual property that we want to see in the divide and conquer algorithms, namely every time that recurse the problem size better not just be smaller but it better be smaller by a significant factor. How would that happen in the selection approach based on the partition subroutine? Well if both of the sub-problems are not too big, then we're guaranteed that when we recurse we make a lot of progress. So let's think about what the best possible pivot would be in the sense of giving a "balanced" split, right, so of course in some sense the best pivot is you just choose your statistic group you're looking for. Then you're done in constant time. But that's extremely unlikely, and it's not worth worrying about. So ignore the fact that we might guess the pivot. What's the best pivot if we want to guarantee an aggressive decrease in the input side for the next iteration. Well, the best pivot is the one that gives as most balanced split as possible. So what's the pivot that gives us the most balanced split? A 50/50 split. If you think about it it's exactly the median. Of course, this is not super-helpful, because the median might well be what we're looking for in the first place. So this is sort of a circular idea. But for intuition, it's still worth exploring what kind of running time we would get in the best case, right? If we're not going to get linear time even in this magical best case, we certainly wouldn't expect to get it on average over random choices of the pivots. So what would happen if we actually did luckily choose the median as the pivot every single time? Well we get the recurrence that the running time that the algorithm requires at a rate of length n. Well, there's only going to be one recursive call. So this is the big difference from QuickSort where we had to recurse on both sides and we had two recursive calls. So here, we're only going to have one recursive call. In the magical case where our pivots are always equal to the median, both sub-problem sizes are only half as large as the original one. So when we recurse, it's on a problem size guaranteed. Could be at most n over two and then outside the recursive call pretty much all we do is a partitioning invocation, and we know that that is linear time. So the recurrence we get is T of N is the most T of N over two plus big O of N. This is totally ready to get plugged into the master method. It winds up being two of the master method and indeed we get exactly what we wanted, linear time. To reiterate this is not interesting in its own right. This is just for intuition. This was a sanity check that at least for a best case choice of pivots we'd get what we want, the linear time algorithm and we do. Now, the question is how well do we do with random pivots? Now the intuition, the hope is exactly as it was for QuickSort which is the random pivots are perfectly good surrogate for the median, for the perfect pivot. So having the analysis of Quicksort under our belt where indeed random pivots do approximate very closely to the performance you get with best case pivots maybe now we have reason to believe that this is hopefully true. That said, as a mathematical statement this is totally not obvious and it's going to take a proof. That's the subject for the next video. Let me just be clear exactly what we're claiming. Here is the running time guarantee the random Rselection provide. For an arbitrary input array of input length n, the average running time of this randomized selection is linear. Big O of n. Let me reiterate a couple of points I made for the analogous guarantee for the QuickSort algorithm. The first is that we're making no assumptions for data whatsoever. In particular we're not assuming that the data is random. This guarantee holds, no matter what input array you feed into this randomized algorithm. In that sense, this is a totally general purpose subroutine. So where then does this averaging come from? Where does the expectation come from? The randomness is not in the data, rather, the randomness is in the code. And we put it there ourselves. Now let's proceed to the analysis.

# 8-2-Randomized Selection - Analysis

In this video I'll explain the mathematical analysis of the randomized linear time selection algorithm that we studied in the previous video. Specifically, I'm going to prove to you the following guarantee for that algorithm. For every single input array of length n the running time of this randomized selection algorithm on average will be linear. Pretty amazing if you think about it because that's barely more what the time it takes just to read the input. And in particular this linear time algorithm is even faster than sorting. So this shows that selection is a fundamentally easier problem than sorting. You don't need to reduce to sorting. You can solve it directly in O(n) time. I want to reiterate the same points I made about quick sort. The guarantee is the same. It is a general purpose subroutine. We make no assumptions about data. This theorem holds no matter what the input array is. The expectation, the average that's in the theorem statement is only over the coin flips made by the algorithm made inside it's code of our own devising. Before we plunge into the analysis, let me just make sure you remember what the algorithm is. So it's like quick sort. We partition around a pivot except we only recurse once, not twice. So we're given an array with some length n. We're looking for the ith order statistic, the ith smallest element. The base case is obvious. You're not in the base case; you choose a pivot p, uniformly at random from the input array just like we did in quick sort. We partition around the pivot just like we did in pic, in quick sort. That splits the array into a first part of those elements less than the pivot and the second part of those elements which are bigger than the pivot. Now, we have a couple of cases. The case which is very unlikely so we don't really worry about, if we're lucky enough to guess the pivot as the ith order statistic what we're looking for. That's when the new position j. Of the pivot element happens to equal I. What we're looking for. Then, of course, we just return it. That was exactly what we wanted. In the general case, the pivot is going to be in the position J, which is either bigger than what we're looking for I, that's when the pivot is too big or J. It's position will be less than the ith order statistic we're looking for. That's when the pivot is too small. So if the pivot's too big, if J is bigger than i that when we're looking for is on the left hand side amongst the elements less than the pivot. So that's where we recurse. We've thrown out both the pivot and everything to the right of it. That leaves us with an array of J minus I elements and we're still looking for the ith smallest among these J minus1 smallest elements. And then the final case, this is what we went through in the quiz and last video, is if we choose a pivot who's smaller than what we're looking for, that's when J is less than I, then it means we're safe to throw out the pivot and everything less than it. We're safe recursing on the second part of those elements bigger than the pivot. Having thrown out the J's smallest elements, we're recursing on an element of length of N-J and we're looking for the i-j smallest element in those that remain, having already thrown out the J smallest from the input array. So that's randomized selection. Let's discuss why it's linear time on average. The first thought that you might have, and this would be a good thought, would be that we should proceed exactly the same way that we did in Quick Sort. You recall that when we analyzed Quick Sort, we set up these indicator random variables, x, i ,j determining whether or not a given, pair of elements got compared at any point in the algorithm. And then we just realized the sum of the comparisons is just the sum, overall, of these x,i, js. We applied linearity of expectation and it boiled down to just figuring out the probability that a given pair of elements gets compared. You can analyze this randomized selection algorithm in exactly the same way. And it does give you a linear time bound on average. But it's a little messy. It winds up being not quite as clean as in the quick sort analysis. Moreover, because of the special structure of the selection problem, we can proceed in an even more slick way here than the way we did with quick sort. So, again we'll have some constituent random variables. We'll again apply linearity of expectation but the definition of those random variables is going to be a little bit different than it was in quick sort. So, first a preliminary observation. Which is that the workhorse for this randomized selection procedure is exactly the same as it was in quick sort. Namely it's the partition subroutine. Essentially all of the work that gets done outside of the recursive call just partitions the input array around some pivot element as we discussed in detail in a separate video that takes linear time. So usually when we say something's linear time we just use big O notation. I'm gonna go ahead and explicitly use a constant c here for the operations outside the recursive call. That'll make it clear that I'm not hiding anything up my sleeves when we do the rest of the analysis. Now what I wanna do on this slide is introduce some vocabulary, some notation which will allow us to cleanly track the progress of this recursive selection algorithm. And by progress I mean. The length of the array on which is currently operating. Remember we're hoping for a big win over quick sort, cuz here we only do one recursive call, not two. We don't have to recurse on both sides of the pivot just on one of them. So it stands to reason, that we can think about the argument making more and more progress as a single recursive calls operating on arrays of smaller and smaller length. So the notion that will be important for this proof is that of a phase. This quantifies how much progress we've made so far, with higher numbered phases corresponding to more and more progress. We'll say that the r select algorithm at some midpoint of its execution is in the middle of phase J. If the array size that the current recursive call is working on has length between 3/4th raised to the J times N and the smaller number 3/4th J+1 times N. For example think about the case where J equals zero. That says phase zero recursive calls, operate on arrays with size of N and 75 percent of N. So, certainly, the outermost recursive call is going to be in phase zero. Because the input array has size N. And then, depending on the choice of the pivot, you may or may not get out of phase zero in the next recursive call. If you choose a good pivot, and you wind up recursing on something, that has, at most, 75 percent of the original elements, you will no longer be in phase zero. If you recurse on something that has more than 75 percent of what you started with, of the. Input array, then you're still gonna be in phase zero even in the second recursive call. So overall the phase number J, quantifies the number of times we've made 75 percent progress, relative to the original input array. And the other piece of notation that's going to be important is what I'm going to call Xj. So for a phase J, Xj simply counts the number of recursive calls in which a randomized selection algorithm is in phase J. So this is gonna be some integer. It could be as small as zero, if you think about it, for some of the phases. Or it could be larger. So why am I doing this? Why am I making these definitions of phases and of these XJs? What's the point? We're gonna remember the point was we wanna be able to cleanly talk about the progress that the randomized selection algorithm makes through its recursion, and what I wanna now show you is that in terms of these XJs, counting the number of iterations in each phase, we can derive a relatively simple upper bound on the number of operations that our algorithm requires. Specifically the running time of our algorithm, can be bounded above by the running time in a given phase, and then summing those quantities over all of the possible phases. So we're gonna start with a big sum, over all the phases J. We want to look at the number of recursive calls that we have to endure in phase J, so that's XJ by definition. And then we look at the work that we do outside of the recursive calls in each recursive call during phase J. Now, in a given recursive call, outside of its recursive call, we do C times M operations where M is the length of the input array and during phase J we have an upper bound on the link of the input array. By definition it's at most three quarters raised to the J times N. So that is, we multiply the running time times this constant C this, we inherit from the partition subroutine and then we can, for the input length, we can put an upper bound of three quarters raised to the J times N. So just to review where all of these terms come from, there's three quarters J times N is an upper bound on the array size. During phase J, this by the definition of the phase. Then, if we multiply that times c, that's the amount of work that we do on each phase J sub-problem. How much work do we do in phase J overall or we just take the work per sub problem that's what's circled in yellow and we multiply it times the number of such sub problems we have. And, of course, we don't wanna forget any of our sub problems so we just make sure we sum all of our phases, j, to insure that at every point we count the work done in each of the sub problems. Okay? So, that's the upshot of this slide. We can upper bound the running time of our randomized algorithm very simply in terms of phases and the XJ's, the number of sub problems that we have to endure during phase j. So, this upper bound on our running time is important enough to give it notation, we'll call this star, this will be the starting point of our final derivation when we complete the proof of this theorem. Now don't forget, we're analyzing a randomized algorithm so therefore the left hand side of this inequality the running time of r select, that's a random variable. So that's a different number depending on the outcome of the random coin flips of the algorithm. Depending on the random pick it has chosen, you will get different random running times. Similarly the right hand side of this inequality. Is also a random variable. That's because the X J's are random variables. The number of sub problems in phase j depends on which pivots get chosen. So. To analyze, what we care about is the expectations of these quantities, their average values. So we're gonna start modestly and as usual, this will extend our modest accomplishments to much more impressive ones using linearity of expectation, but our first modest goal is just to, to understand the average value. Of an XJ, the expected value of XJ. We're gonna do that in two steps. On the next slide, I'm going to argue that to analyze the expectation of XJ, it's sufficient to understand the expectation of a very simple coin flipping experiment. Then, we'll analyze that coin flipping experiment. Then we'll have the dominos all set up in a row. And on the final slide, we'll knock'em down and finish the proof. So let's try to understand the average number of recursive calls we expect to see in a given phase. So, again, just so we don't forget. Xj is defined as the number of recursive calls during phase J. Where a recursive call is in phase J, if and only if the current sub array length lies between three-fourths raised to the J+1 times N. And then, the larger number of three-fourths raised to the J times N. So again, for example, phase zero is just the recursive calls under which the array length is between 75 percent of the original element and 100 percent of the original elements. So what I wanna do next is point out that a very simple sufficient condition guarantees that we'll proceed from a given phase onto the next phase. So it's a condition guaranteeing termination of the current phase. And it's an event that we've discussed in previous videos. Mainly that the pivot that we choose gives a reasonably balanced split. 25-75 or better. So recall how partitioning works, we choose a pivot P. It winds up wherever it winds up. And the stuff to the left of it's less than P. The stuff to the right of it is bigger than P. So 25 to 75 split or better, what I mean is that each of these, each, the first part and the second part has, at most, 75 percent of the elements in the input array. Both have twen-, both have at least 25%, and, at most, 65%. And the key point is, that if we wind up choosing a pivot that gives us a split that's at least as good the current phase must end. Why must the current phase end? Well, to get a 25, 75 split or better than no matter which case we wind up in, in the algorithm we're guaranteed to recurse on a sub problem that has at most 75 percent of what we started with. That guarantees that whatever phase we're in now, we're going to be in an even bigger phase when we recursed. Now, I want you to remember something that we talked about before, which is that you've got a decent chance when you pick a random pivot of getting something that gives you a 25, 75 split or better. In fact, the probability is 50 percent. Right? If you have an array that has the integers from one to 100 inclusive, anything from 76 to s, 26 to 75 will do the trick. That'll insure that at least the first 25 elements are excluded from the rightmost call and at least rightmost 25 elements are excluded from the left recursive call. So this is why we can reduce our analysis of the number of recursive calls during a given phase, to a simple experiment involving flipping coins. Specifically, the expected number of recursive calls. Now we are gonna see in a given phase J, is no more than the expected number of coin flips in the following experiment. Okay, so you've got a fair coin, 50 percent heads, 50 percent tails. You commit to flipping it until you see the head and the question is, how many coin flips does it take up to and including the first head that you see? So the minimum it's gonna be one coin flip if you hit a head the first time it's one. If you get a tails and then a head, then it's two. If it's tails, tails, head it's three and so on, and you always stop when you hit that first head. So what's the correspondence? Well, think of heads as being you're in Phase J, and if you get a good pivot, it gives you a 25/75 split. Call that heads. And it guarantees that you exit this Phase J. Just like it guarantees that you get to terminate the coin flipping experience, experiment. Now, if you get a pivot which doesn't give you a 25/75 split, you may or may not pass to a higher Phase J, but in the worst case, you don't. You stick to phase J is you get a bad split, and that's like getting a tails in the coin flipping experiment, and you have to try again. This correspondence give us a very elementary way to think about the progress that, that our randomized selection algorithm is making. So, there's one recursive call in every step in our algorithm, and each time we either choose a good pivot or a bad pivot, both could happen, 50-50 probability. A good pivot means we get a 75-25 split or better. A bad pivot means, by definition, we get a split worse than 25-75. So what have we accomplished? We've reduced the task of upper bounding the expected number of recursive calls in a phase J to understanding the expected number of times you have to flip a fair coin before you get one hit. So on the next slide we'll give you the classical and precise answer to this coin flipping experiment. So, let me use capital N to denote the random variable, which we were just talking about, the number of coin flips you need to do before you see the first heads. And, it's not very important, but you should know that these random variables have their own name. This would be a geometric random variable with parameter one-half. So you can use a few different methods to compute the expected value of a geometric random variable such as this, and brute force using the definition of expectation works fine as long as you know how to manipulate infinite sums. But for the sake of variety, let me give you a very sneaky proof of what it's expectation is. So the sneaky approach is to write to the expected value of this random variable in terms of itself and then solve for the unknown, solve for the expectation. So let's think about it. So how many coins flips do you need? Well for sure you're gonna need one. That's the best case scenario. And now two things can happen, either you get heads and that has 50 percent probability you stop or you get tails that happens with 50 percent probability and now you start all over again. Again you just put points until you get first heads. On average how many times does that take. Well by the definition of capital N you expect. The expectation of N coin flips, in the case where you get tails, and you have to start all over. So this one represents the first coin flip, the one-half is the probability that you can't stop, that you have to start all over probability of tails, and then because it's a memory less process, because when you start anew on the second coin flip having gotten the tails, it's as if you're back at time one all over again. So now we have a trivial equation, in terms of the unknown expected value of N and the unique solution, the unique value, that the expected value of capital N could have, in light of this equation, is two. So, on average if you flip a fair coin and stop when you get heads, you're going to see two coin flips on average. To make sure you haven't sort of lost the forest for the trees, let me remind you why we were talking about this coin flipping analysis in the first place. So recall in the previous slide we showed that XJ, and remember XJ is the number of recursive calls you'd expect to see in a given phase J, and we argued that the number of recursive calls you're gonna see is bounded above. By the expected number of coin flips until the heads. So this exact calculation of two for the coin flips gives us an upper bound of two for the number of recursive calls on average in any given phase J. So now that we've got all our ducks lined up in a row, let's wrap up the proof on this final slide. So, inherited from part one of the proof, we have an upper bound. On the expected running time. Of the R select algorithm. This is what we were calling star on the first brief slide In star, it looked a little messy, but we had the sum over the phases J. But we had two things that were independent of j: the constant c and the original input length n, so let me just yank the c and the n out front. And then we have this residual sum over the phases J. Of three quarters raised to the J remember that comes from our upper bound on the sub problem size during phase J and then of course we have to keep track of how many phase J sub problems we have solved that by definition is XJ. Now star was written as a rand in accordance terms to the random variables. Now we're gonna go ahead and take the expectations and again I have said this over and over but don't forget where's the expectation come from. This is over the random pivot choices that our code makes. So the expected running time of the algorithm is most the expectation of this start quantity. So like I said earlier, pretty much every time we're gonna do any analysis of [inaudible] process, we're gonna wind up using linearity of expectation at some point. Here is where we do it. Linear expectation says the expectation of a sum is just the sum of the expectations. So we yank the c and the n outside of the expectation. We yank this sum over phases. Outside of the expectation. We yank this three-fourths raised to the J outside of the expectation and then we just have the expected value of XJ, the average number of recursive calls we expect to see in HJ. Now on the previous two slides, we figured out an upper bound on how many recursive calls we expect to see in each phase. So first by the coin flip analysis, by the reduction of the coin flip analysis, this is the most expected number of coin flips N, which on the previous slide, we argued was exactly two. So bringing that two out in front of the sum, that no longer depends on J. So we get a most 2CN. Times the sum over phases J, of three quarters raised to the J. Now this kind of sum we have seen previously in the course. It came up when we were analyzing the master method and we summed up our running time upper bounds over the levels of our recursin tree. And if we're not in case one if we're in case two or three we had geometric sums that were nontrivial. They require a certain formula to calculate, so let me remind you of that formula here, when the three quarters are being powered up to the J. So this has value at most, one over one minus, the number that's getting powered, so in this case it's three quarters, one minus three quarters is a quarter check reciprocal, you got four. And the upshot is that the expected number of operations that this randomized selection algorithm uses to find the [inaudible] ordered statistic in a given input array, is eight times C times N. Where C is the, hidden constant in the linear running time of partition. And so that completes the proof. The input array was arbitrary. We showed the expected running time over the random choices of the algorithm is linear in N. That is, only a constant factor larger than what is required to read the input. Pretty amazing.

# 8-3-Deterministic Selection - Algorithm [Advanced - Optional]

Previous videos covered an outstanding algorithm for the selection problem, the problem of computing the Ith statistic of a given array. That algorithm which we called the R select algorithm was excellent in two senses. First its super practical, runs blazingly fast in practice. But also it enjoys a satisfying theoretical guarantee. For every input array of length N at the expected running time of R select is big O of N, where the expectation is over the random choices of the pivots that R select makes during execution, now in this optional video I'm going to teach you yet another algorithm for the selection problem. Well why bother given that our select is so good? Well frankly, I just can't help myself. The ideas of this algorithm are just too cool not to tell you about, at least in optional video like this one. The selection algorithm , we cover here is deterministic. That is, it uses no randomization whatsoever. And it's still gonna run in linear time, big O of N time. But now, in the worst case for every single input array. Thus, the same way Merge Short gets the same asymptotic running time, big O of N log N, as quick sorts gets on average. This deterministic algorithm will get the same running time O of N, as the R select algorithm does on average. That said, the algorithm we're gonna cover here, well, it's not slow. It's not as fast as R select in practice, both because the hidden constants in it are larger. And also because it doesn't' operate in place. For those of you who are feeling keen, you might wanna try coding up both the randomized and the deterministic selection algorithms, and make your own measurements about how much better the randomized one seems to be. But if you have an appreciation for Boolean algorithms, I think you'll enjoy these lectures nonetheless. So let me remind of the problem. This is the i-th order statistic problem. So we're given an array, it has N distinct entries. Again, the distinctness is for simplicity. And you're given a number I between one and N. You're responsible for finding the i-th smallest number, which we call the i-th order statistic. For example, if I is something like N over two, then we're looking for the median. So let's briefly review the randomized selection algorithm. We can think of the deterministic algorithm covered here as a modification of the randomized algorithm, the R select algorithm. So when that algorithm is passed in array with length N, and when you're looking for the i-th order statistic, as usual, there's a trivial base case. But when you're not in the base case, just like in Quick Sort, what you do is you're gonna partition the array around pivot element P. Now, how are you gonna choose P? Well, just like Quick Sort, in the randomized algorithm, you choose it uniformly at random. So each of the N elements of the input array are equally likely to be chosen. As the pivot. So, call that pivot p. Now, do the partitioning. Remember partitioning puts all of the elements less than the pivot to the left of the pivot. We call that the first part of the partitioned array. Anything big, bigger than the pivot gets moved to the right of the pivot. We call that the second part of the array. And let j denote the position of the pivot in this partitioned array. Equivalently, let j be what order statistic that the pivot winds up happening to be. Right? So, we happen to choose the minimum element then J's gonna be equal to one. If we happen to choose the maximum element, J's gonna be equal to n. And so on. So, there's always the lucky case, chance one in N, that we happen to choose the Ith order statistic as our pivot. So, we're going to find that out when we notice that J equals I. In that super lucky case, we just return the pivot and we're done. That's what we're looking for in the first place. Of course, that's so rare it's not worth worrying about, so really the two main cases depend on whether the pivot that we randomly choose is bigger than what we are looking for or if it's less than what we are looking for. So, if it's bigger than what we are looking for, that means J is bigger than I, we're looking for the Ith smallest, we randomly chose the J'th smallest. Then remember we know that the Ith smallest element has to lie to the left of the pivot. Good element in that first part of the partition array. So we recurs there. It's an array that has J-1 elements in it, everything less than the pivot. And we're still looking for the Ith smallest among them. In the other case, this was the case covered in a quiz a couple videos back, if we guess a pivot element that is less than what we're looking for, well then we should discard everything less than the pivot and the pivot itself. So we should recurs on the second part of A, stuff bigger than the pivot. We know that's where what we're looking for lies. And having thrown away J elements, the smallest ones at that. We're rehearsing on a ray of [inaudible] and minus J, I'm looking for the [inaudible] smallest element in that second part. So, that was the randomized selection algorithm, and you'll recall the intuition for why this works is random pivot should usually give pretty good splits. So the way the analysis went is we should. Each iteration, each recursive call, with 50 percent probability, we get a 25/75 split or better. Therefore, on average, every two recursive calls, we are pretty aggressively shrinking the size of the recursive call. And for that reason, we should get, something like a linear time bound. We do almost as well as if we picked the median in every single call, just because random pivots are a good enough proxy of best case pivots, of. The median. So now the big question is: suppose we weren't permitted to make use of randomizations. Suppose this choose-a-random-pivot trick was not in our tool box. What could we do? How are we going to deterministically choose a good pivot? Let's just remember quickly what it means to be a good pivot. A good pivot is one that gives us a balanced split, after we do the partitioning of the array. That is, we want as close to a 50/50 split between the first and the second parts of the partitioned array as possible. Now, what pivot would give us the perfect 50/50 split? Well, in fact, that would be the median. Well, that seems like a totally ridiculous observation, because we canonically, are trying to find the median. So previously we were able to be lazy, and we just picked a random pivot, and used that as a pretty good proxy for the best case pivot. But now, we have to have some subroutine that deterministically finds us a pretty good approximation of the median. And the big idea in this linear time selection algorithm, is to use what's called the median of medians as a proxy for the true meaning of the input array. So when I say median of medians, you're not supposed to know what I'm talking about. You're just supposed to be intrigued. Now, let me explain a little bit further. Here's the plan, we're gonna have our new implementation of chose pivot and it's gonna be deterministic. You will see no randomization on this slide, I promise. So the high-level strategy is gonna be we're gonna think about the elements of this array like sports teams, and we're gonna run a tournament, a 2-round. Knockout tournament, and the winner of this tournament is going to be who we return as the proposed pivot element. Then we'll have to prove that this is a pretty good pivot element. So there's gonna be two rounds in this tournament. Each element, each team is gonna first participate in a world group, if you will. So they'll be, small groups of five teams each, five elements each. And to win your first round, you have to be the middle element out of those five. So that'll give us N over five first round winners. And then the winner of that second round is going to the med-, be the median of those N over five winners from the first round. Here are the details. The first step isn't really something you actually do in the program, it's just conceptually. So logically, we're going to take this array capital A, which has N elements, and we're gonna think of it as comprising N over five groups with five elements each. So if N is not a multiple of five, obviously, there'll be one extra group that has size between one and four. Now for each of these groups of five, we're going to compute the median, so the middle element of those five. Now for five elements, we may as well just invoke a reduction to sorting; we're just gonna sort each group separately, and then use the middle element, which is the median. It doesn't really how you do the sorting. Because after all, there's only five elements. But you know, let's use [inaudible] sort, what the heck. Now what we're going to do is we're going to take our first round winners and we're gonna copy them over into their own private array. Now this next step is the one that's going to seem dangerously like cheating, dangerously like I'm doing something circular and not actually defining a proper algorithm, so c you'll notice has linked over n over five. We started with an array of link n. This is a smaller input. So let's recursively compute the median of this array capital c. That is the second round of our tournament amongst the n over five first-round winners, the n over five middle elements of the sorted groups. We recursively compute the median, that's our final winner, and that's what we return as the pivot element from this subroutine. Now I realize it's very hard to keep track of both what's happening internal to this juice pivot subroutine and what's happening in the calling function of our deterministic selection algorithm. So let me put them both together and show them to you, cleaned up, on a single slide. So, here is the proposed deterministic, selection algorithm. So, this algorithm uses no randomization. Previously, the only randomization was in choosing the pivot. Now we have a deterministic subroutine for choosing the pivot, and so there's no randomization at all. I've taken the liberty of in-lining true's pivot subroutine. So that is exactly what lines one, two, and three are. I haven't written down the base case just to save space I'm sure you can remember it, so if you're not in the base case. What did we do before? The first thing we do is choose a random pivot. What do we do now? Well, we have steps one through three. We do something much more clever to choose a pivot. And this is exactly what we said on the last slide. We break the array into groups of five. We sort each group, for example, using merge sort. We copy over the middle element of each of the n over five groups into their own array capital c. And, then, we recursively compute the median of c. So, when we recurs on select that we pass the input c. C has n over five elements so that's the new link. That's a smaller link than what we start with so it's a legitimate recursive call refining the median of n over five element. So, that's gonna be the n over tenth order statistic. As usual. Well to keep things clear I'm ignoring stuff like fractions, in your real implementation you'd just round it up or down. As appropriate. So steps one through three are the new [inaudible] step routine that replaces the randomized selection that we had before. Steps four through seven are exactly the same as before. We've changed nothing. All we have done is ripped out that one line where we chose the pivot randomly and pasted in these lines one through three. That is the only change to the randomized selection algorithm. So, the next quiz is a standard check that you understand this algorithm, at least, not necessarily why it?s fast; but, at least, just how it actually works. And I only ask you to identify how many recursive calls there are, each time. So, for example in [inaudible] there's two recursive calls, in quick-sort there's two recursive calls, in r-select there's one recursive call. How many recursive calls do you have each time, outside of the base case in the d-select algorithm? All right, so the correct answer is two. There are two recursive calls in deselect, and maybe the easiest way to answer this question is not to think too hard about it and literally just inspect the code and count, right namely there's one recursive call in line three, and there's one recursive call in either six or seven, so quite literally, you know there's seven lines of code, and two of the ones that get executed have a recursive call so the answer is two. Now what's confusing is that in the random, a couple things, first in the randomized selection algorithm, we only have one recursive call. We have the recursive call. In line six or seven, we didn't have this in line three. That one in line three is new compared to the randomized procedure. So we're kind of used to thinking of one recursive call using the divide and conquer approach to selection, here we have two. Moreover. Conceptually. The roll of these two recursive calls are different. So the one in line six or seven is the one we're used to. That's after you've done the partitioning so you have a smaller sub-problem and then you just recursively find the residual or statistic in the residual array. That's sort of the standard divide and conquer approach. What's sort all crazy. Is this second use of a recursive call which is part of identifying a good pivot element for this outer recursive call and this is so counter-intuitive, many students in my experience don't even think that this algorithm will hold, sort of, they sort of expect it to go into an infinite loop. But again, that sort of over thinking it. So let's just compare this to an algorithm like merge sort. What does merge sort do? Well it does two recursive calls and it does some other stuff. Okay. It does linear work. That's what it does to merge. And then there are two recursive calls on smaller sub problems, right? No issue. We definitely feel confident that merge [inaudible] is gonna terminate because the sub problems keep getting smaller. What does deselect do, if you squint? So don't think about the details just [inaudible] high level. What is the work done in deselect? Well. There are two recursive calls, there's [inaudible] one's in line three, one's in line six or seven, but there's two recursive calls on sm, smaller sub problem sizes. And there's some other stuff. There's some stuff in steps one and two and four, but whatever. Those are recursive calls. It does some work. Two recurs have caused the smaller sub-problems, TI's got to terminate. We don't know what the run time is, but it's got to terminate, okay? So if you're worried about this terminating, forget about the fact that the two recurs of cause have different semantics and just remember, if ever, you only recurs on smaller sub-problems, you're definitely going to terminate. Now, of course who knows what the running time is? I owe you an argument on why it would be anything reasonable, that's going to come later. In fact what I'm gonna prove to you is not only does this selection algorithm terminate, run in finite time, it actually runs in linear time. No matter what the input array is. So where as with R select, we could only discuss its expected running time being linear. We showed that with disastrously bad choices for pivots, R selects can actually take quadratic time. Under no circumstances will deselect ever take, ever take quadratic time. So for every input array it's big O of N time. There's no randomization because we don't randomly do anything in choose pivot, so there's no need to talk about average running time; just the worst case running time over all inputs is O of N. That said, I want to reiterate the warning I gave you at the very beginning of this video which is, if you actually need to implement a selection algorithm, you know, this one wouldn't be a disaster. But it is not the method of choice, so I don't want you to be misled. As I said there are two reasons for this. The first is that the constants hidden in the begon notation are larger for V select than for R select. That will be somewhat evident from the analyses that we give for the two algorithms. The second reason is, recall we made a big deal about how partitioning works in place and therefore quicksort and r select also work in place, that is, with no real additional memory storage. But in this deselect algorithm we do need this extra array c to copy over the middle elements, the first round winners. And so the extra memory, as usual, slows down the practical performance. One final comment. So for many of the algorithms that we cover, I hope I explain them clearly enough that their elegance shines through and that for many of them you feel like you could have up with it yourself, if only you'd been in the right place at the right time. I think that's a great way to feel and a great way to appreciate some of these very cool algorithms. That said, linear time selection, I don't blame you if it, if you feel like you never might have come up with this algorithm. I think that's a totally reasonable way to feel after you see this code. If it makes you feel better, let me tell you about who came up with this algorithm. It's quite old at this point, about 40 years, from 1973. And the authors, there are five of them and at the time this was very unusual. So, Manuel Blum. Bob Floyd. Vaughn Pratt. Ron Rivest. And Bob Targen. And this is a pretty heavy weight line up, so as we've discussed in the past, the highest award in computer science is the ACM Turing award given once each year. And I like to ask my algorithms classes how many of these authors do they think, have been awarded a Turing award. I've asked him many times. The favorite answer anyone's ever given me has been. Six, which I think is in spirit should be correct. Strictly speaking the answer is four. So, the only one of these five authors that doesn't have a Touring award is Von Pratt, although he's done remarkable things spanning the gambit from co-founding Sun Systems to having very famous theorems about, for example, testing primality. But the other four have all been awarded the Touring award at some point. So in chronological order, so the late Bob Floyd who was a professor here at Stanford. Was awarded the 1978 [inaudible] award, both for contributions to algorithms but also to program languages and compilers. So Bob Targen who, as we speak, is here as a visitor at Stanford and has spent his Ph. D here and has been here as a faculty at times, was awarded it for contributions to graph algorithms and data structures. We'll talk some more about some of his other contributions in future courses. Manuel Blum was awarded the Turing award in'95 largely for contributions in cryptography, and many of you probably know Ron Rivest as the R in the RSA cryptosystem. So he, won the, Turing award along with Shamir and Adleman back in'02. So in summary, if this algorithm seems like one that might have alluded you even on your most creative days, I wouldn't feel bad about it. This is a, this is a quite clever algorithm. So let's now turn to the analysis and explain why it runs in linear time in the worst

# 8-4-Deterministic Selection - Analysis I [Advanced - Optional]

Now let's turn to the analysis of the deterministic selection algorithm that we discussed in the last slide by Blum, Floyd, Pratt, Rivest, and Tarjan. In particular, let's prove that it runs in linear time on every possible input. Let's, remind you what the algorithm is. So the idea is, we just take the R select algorithm. But instead of choosing a pivot at random, we do quite a bit more work to choose what we hope is going to be a guaranteed pretty good pivot. So again, lines one through three are the new choose pivot subroutine. And it's essentially implementing a two round knockout tournament. So first, we do the first round matches. So what does that mean? That means we take A we think of it as comprising these groups of five elements. So the first five elements one through five and the elements six through ten and points eleven through fifteen and there again and so on. If we sort each of those five using, let's say, merge sort although it doesn't matter much, then the winner in each of these five first round matches is the median of those five. That is the third highest element, third largest element out of the five. So we take those in over five first round winners the middle element of each of the five and the sorted groups, we copy those over into a new array of capital [inaudible] and [inaudible] in it for five. And then we. Second round of our tournament at which we elect the medium of these N over five, first round winners as our final pivot, as our final winner. So, we do that by recursively calling deselect on C. It has a length N over five [inaudible] for the medium. So that's the N over tenth [inaudible] statistic in that array. So, we call the pivot P and then we just proceed exactly like we did. And in the randomized case. That is, we partition A around the pivot, we get a first part, a second part, and we recurs on the left side or the right side as appropriate, depending on whether the pivot is less than or bigger than the element that we're looking for. So the claim is, believe it or not, that this algorithm runs in linear time. Now, you'd be right to be a little skeptical of that claim. Certainly, you should be demanding from me some kind of mathematical argument about this linear time claim. It's not at all clear that that's true. One reason for skepticism is that this is an unusually extravagant algorithm. In two senses for something that's gotta run in linear time. First is, first is it's extravagant use of recursion. There are two different recursive calls, as discussed in the previous video. We have not yet seen any algorithm that makes two recursive calls and runs in linear time. The best case scenario was always [inaudible] for our two recursive call algorithms like merge sort or quick sort. The second reason is that, outside the recursive calls, it seems like it?s just kind of a lot of work, as well. So, to drill down on that point, and get a better understanding for how much work this algorithm is doing, the next quiz asks you to focus just on line one. So when we sort groups of five in the input array how long does that take. So the correct answer to this quiz is the third answer. Maybe you would have guessed that given that I'm claiming that the whole algorithm takes linear time, you could have guessed that this sub-routine is going to be worse than linear time. But you should also be wondering you know, isn't sorting always N log N so, aren't we doing sorting here. Why isn't the N log N thing kicking in? The reason is we're doing something much, much more modest than sorting the linked N input array, all we're sorting are these puny little sub-arrays that have only five elements and that's just not that hard, that can be done in constant time so let me be a little more precise about it. The claim is that sorting an element, an array with five elements takes only some constant number of operations. Let's say 120. Where did this number, 120 come from? Well, you know, for example, suppose we used merge sort. If you go back to those very early lectures, we actually counted up the number of operations that merge sort needs to sort an array length of M. For some generic M, here M is five, so we can just plug five into our previous formula that we computed from merge sort. Right if we plug amicle five into this formula, what do we get, we get six times five times log base 205+1. Who knows what log base 205 is, that's some weird member but it's gonna be a most three right. So that's the most three of three+1 is four multiply that by five and again time six and will get you 120. So it's constant time to sort just one of these groups of five. Now of course, we have to do a bunch of groups of five because there's only a linear number of groups. Constant for each, so it's gonna be linear time overall. So to be really pedantic. We do 120 operations at most per group. There's N over five different groups. We multiply those, we get 24 N operations. So do all the sorting and that's obviously a big O event. So linear time for step one. So having warmed up with step one. Let's look now at the whole seven line algorithm, and see what's going on. Now I hope you haven't forgotten the paradigm that we discussed for analyzing the running time of deterministic divide and conquer algorithms like this one. So namely we're gonna develop a recurrence and remember a recurrence expresses the running time, the number of operations performed, in two parts. First of all, there's the work done by the recursive calls on smaller sub-problems. And secondly, there's the work done locally, not in the recursive calls. So let's just go through these lines one at a time, and just do a running tally of how much work is done by this algorithm, both locally and by the recursive calls. So the quiz was about, step number one. We just argued that since it's constant time to sort each group, and there's a linear number of groups, we're gonna do linear work, theta of N. For step one. So copying these first round winners over in to their special array C is obviously linear time. Now, when we get to the third line, we have a recursive call, but it's a quite easy recursive call to understand. It's just, recursing on a, a ray that has size twenty percent as large as the one we started with, on the N over five elements. So this, remember the notation we used for recurrences. Generally, we denote by capital T the running time of an algorithm on [inaudible] of a given length. So this is going to be the running time that our algorithm has in the worst case on inputs of length N over five. Cuz N over five is the length of the array that we're passing to this recursive call. Good. Step four, partition. Well we had. Videos about how they were going to partition the Y to linear time. We knew that all the way back from quick sort, so that's definitely Theta of N. Step five is constant time, I'm not going to worry about it. And finally we get to lines six and seven so at most one of these will execute so in either case there's one recursive call. So that's fine, we know in recurrences when there's recursive call we'll just write capital T of whatever the input length is. So we just have to figure out what the input length here is. It was N over five in step, in line three so we just have to figure out what it is in line six or seven. Oh yeah, now we're remembering why we didn't use recurrences when we discussed randomized quick sort and. The randomized selection algorithm. It's because we don't actually know how big the recursive call is, how big the input passed to this recursive call in line six or seven is. Line three, no problem. It's guaranteed to be twenty percent of the input array cuz that's how we defined it. But for line six or seven, the size of the input array that gets passed to the, to the recursive call depends on how good the pivot is. It depends on the splitting of the array A into two parts, which depends on the choice of the pivot P. So at the moment all we can write is T. Of question mark. We don't know. We don't know how much work gets done in that recursion, cause we don't know what the input size is. Let me summarize the results of this discussion. So write down a recurrence for the D select algorithms. So with T of N denote the maximum number of operations the D select ever requires to terminate an array of input [inaudible]. It's just the usual definition of T of N when using recurrences. What we established in our tally on the last slide is that deselects does linear stuff outside the recursive calls. It does the sorting of groups of five. It does the copying, and it does the partitioning. Each of those is linear, so all of them together is also linear. And then it does two recursive calls. One whose size we understand, one whose size we don't understand. So, for once I'm not going to be sloppy and I'm going to write out an explicit constant about the work done outside of the recursive cause. I'm going to write [inaudible], I'm going to actually write C times N for some constant C. So of course no one ever cares about base cases, but for completing this let me write it down anyways. When D select gets an input of only one element it returns it, what's called that one operation for simplicity. And then in the generals cases and this is what's interesting. When you're not in the base case and you have to recurs, what happens? Well you do a linear work outside of the recursive call. So that's C times N for some constant C. C is just the [inaudible] constant on all of our big thetas on the previous slide. Plus the recursive call in line three, and we know that happens on an array of size [inaudible] five. As usual, I'm not gonna worry about rounding up or rounding down, it doesn't matter. Plus our mystery recursive call on an array of unknown size. So that's where we stand and we seem stuck because of this pesky question-mark. So, let's prove lemma which is gonna replace this question-mark with something we can reason with, with an actual number that we can then analyze. So the upshot of this key lemma is that all of our hard work in our choose pivot subroutine in lines one through three bears fruit in the sense that we're guaranteed to have a pretty good pivot. It may not be the median, it may not give us a 50/50 split. Then we could replace the question mark with, one-half times N. But it's gonna let us replace the question mark by seven-tenths times N. Now, I don't wanna lie to you, I'm gonna be honest, it's not quite 7/10N, it's more like 7/10N minus five, there's a little bit of additive error, so, taking care of the additive error adds nothing to your conceptual understanding of this algorithm or why it works. For those of you who want a truly rigorous proof, there are some posted lecture notes which go through all the gory details. But in lecture I'm just gonna tell you what's sort of morally true and ignore the fact that we're gonna be off by three here and four there. And then we'll be clear when I show you the proof of this limit, where I'm being a little bit sloppy and why it really shouldn't matter, and it doesn't. So to explain why this key limit is true why we get a 30 70 split or better guaranteed, let me set up a little notation. I'm getting sick of writing N over five over and over again, so let's just give that a synonym, let's say, K. So this is the number of different sort of first round matches that we have, the number of groups. I also want some notation to talk about the first round winners, that is the medians of these groups of five, the K first round winners. So, were gonna call XI the [inaudible] smallest of those who win their first round match and make it to the second round. So just to make sure the notation is clear, we can express the pivot element in terms of these X?s. Remember, the pivot is the final winner. It wins not only its first round tournament, but it also the second round tournament. It's not only the middle element of the first group of five. It's actually the median of the N over five middle element. It's the median of the medians. That is, of the K middle elements, it's the K over two order statistic, [inaudible] K over two smallest. I'm saying this, assuming that K is even. If K was odd, it would be some slightly different formula as you know. So let's remember what we're trying to prove. We're trying to prove that for our proposed pivot, which is exactly this element X sub K over two, it's exactly the winner of this 2-round knockout tournament. We're trying to argue that for this proposed pivot, we definitely get a 30-70 split or better. So what that means is, there better be at least 30 percent of the elements that are bigger than the pivot. That way if you recurs on the left side on the first part, we don't have to deal with more that more than 70 percent of the original elements. Similarly, there better be at least 30 percent of the elements that are smaller than the pivot. That way if we recurs on the right hand side we know we don't have to deal with more than 70 percent of the original input elements. So if we achieve this goal, we prove that there's at least 30 percent on each side of XK over two, then we're done. That proves the key lemma that would get a 30/70 split or better. So I'm gonna show you why this goal is true. I'm gonna introduce a thought experiment. And I'm gonna lay out it abstractly. Then we'll sorta do an example to make it more clear. And then we'll go back to the general discussion and finish the proof. So what we're gonna do is a thought experiment, for the purposes of counting how many elements of the input array are bigger than our pivot choice, and how many are smaller. So in our minds we're going to imagine that we're taking elements in A and rearrange them in a 2D grid. So here are the semantics of this grid. Each column will have exactly five elements that will correspond to one of the groups of five. So we'll have N over five columns corresponding to our N over five groups in our first round of our tournament. [inaudible] is not a multiple of five then one of these groups has size between one and four but I'm just not gonna worry about it, that some of the additive loss, which I'm ignoring. Moreover were going to arrange each column in a certain way so that going from bottom to top the entries of that go from smallest to largest. So this means that in this grid we have five rows. And the middle row, the third row, corresponds exactly to the middle elements, to the winners of the first round matches. So because these middle elements these first round winners are treated specially, I'm going to denote them with big squares, the other four elements of the group two of which are smaller two of which are bigger are just going to be little circles. Furthermore, in this thought experiment, in our mind, we're going to arrange the columns from left to right in order of increasing value of the middle element. Now remember, I introduced this notation X of I is the [inaudible] smallest amongst the middle elements. So a different way of what I'm trying to say is that the leftmost column is the group that has X1 as its middle element. So among the N over five middle elements, one of the groups has the smallest middle elements. We put that all the way on the left. So this is gonna be X1 in the first column, the smallest of the first round winners. X2 is the second smallest of the first round winners, X3 is the third smallest and so on. At some point we get to the median of the first round winners, XK over two. And then, way at the rights is the largest of the first round winners. And I'm sure that you remember that the median of medians which is XK over two is exactly our pivot. So this is our lucky winner. I know this is a lot to absorb, so I'm gonna go ahead and go through an example. If what I've said so far makes perfect sense, you should feel free to skip the following example. But if there's still some details you're wondering about, and hoping this example will make everything crystal clear. So let's suppose we have an input array. I need a, a slightly big one to [inaudible] grid make sense. Let's say there's an input array of twenty elements. So there's going to be the input array, which is in a totally arbitrary order. There's gonna be the vertical [inaudible] after we sort each group of five. And then I'm gonna show you the grid. So this is the input we're all gonna use. Let's now go ahead and delineate the various groups of five. So after sorting this group, you get the following. From each group there's a single winner mainly the middle element so that would be the twelve, and the six, and the nine, and the fourteen, those are the four survivors from the first round of the tournament. And the median of these four elements which, at the end of the day is gonna to be our pivot is the second smallest of the four, that's how we define the median from an even number of elements, so that's gonna be the nine. So, this first transformation from the input array, to this vaguely mini sorted version of the input array with the groups of five sorted, this we actually do in the code. This happens in the algorithm. Now, this grid we're just doing in our minds. Okay? We're just in the middle of proving why the algorithm is fast. Why the fit bits guaranteed to give us close to a, a 30 70 split or better. So, let me show you an example of this grid in our mind, what it looks like for this particular input. So the grid always has five rows. The columns always have five elements cause the columns correspond to the groups. Here because N equals twenty and over five is four. So there's gonna be, four columns and five rows. And moreover we arrange the columns from left to right so that these middle elements go from smallest, to largest. So the middle elements are six nine twelve and fourteen and we're gonna draw the columns in that order from left to right. So first we'll write down the middle elements, the middle row from decreasing to increasing, six, nine, twelve, fourteen. Again the median of this is our pivot, which is the nine. And then each column is just the other four elements that goes along with this middle element from decreasing to increasing as we go from bottom to top. So this is the grid that we're been talking about on the other slide, in this particular example. So I hope that makes what we're talking about clear, what these X?s mean, and what worry we have amongst the rows, amongst the columns and so on. So let?s go back to the general argument. Here is the key point, here is why were doing this entire thought experiment, it?s going to let us prove our key limit. We're going to get a 30/70 split or better. 30 percent of the stuff at least is less than the pivot; 30 percent at least is bigger than the pivot. So why is there at least 30 percent of the stuff below the pivot? Why is the pivot bigger then at least 30%? Well, it's bigger then everything to the left and everything below the stuff to the left. That is we know that XK over two is bigger than the K over two minus one elements. That is to the left of it, those other middle elements that it's bigger then. That's because it's the median of the medians. >> So, if we just go straight west from the pivot we only see stuff which is less. Furthermore, these columns are arranged from decreasing to increasing order as we go from south to north, from bottom to top. So if you travel south from any of these smaller XMI we only see stuff which is still smaller. So all we're using in here is transitivity of the less than relation. If you go straight west you see stuff which is only smaller from any of those points if you go southward you'll see stuff which is even smaller than that. So this entire yellow region, everything southwest of the pivot element, is smaller than it. And that's a good chunk of the grid. Right? So for all of these columns, it's basically three out of the five, or 60 percent of them are smaller than the pivot, and half of the columns, essentially, are in this part of the grid. So if the pivots bigger than 60 percent of the stuff in 50 percent of the groups that means it's bigger than 30 percent of the elements overall. And if we reason in an exactly symmetric way, we find that the pivot is also smaller than at least 30 percent of the array. So to find things bigger than the pivot, what do we do? First we travel eastward. That gives us middle elements that are only bigger than it and then we stop wherever you want on our eastward journey and we head north, and we're gonna see stuff which is still bigger. So this entire north eastern corner. Is bigger than the pivot element, and again that's 50%, that's at 60 percent of roughly 50 percent of the groups. Returning to our example, the southwest region of the nine. Is this stuff, one, three, four, five, six. Certainly, all of that is smaller than the nine. You'll notice there's other things smaller than the nine as well. There's the eight, there's the two, there's the seven, which we're not counting. But it depends on the exact array. Whether or not, in those positions, you're gonna have stuff smaller than the pivot or not. So it's this yellow region we're guaranteed to be smaller than the pivot. Similarly, everything northeast of the pivot is bigger than it. Those are all double digit numbers and our pivot is nine. Again there's some other stuff in other regions bigger than the pivot, the twenty, the ten, the eleven, but again those are positions where we can't be guaranteed that it will be bigger than the pivot. So it's the yellow regions that are guaranteed to be bigger and smaller than the pivot, and that gives us the guaranteed 30 70 split. Okay, so that proof was hard work, showing that this deterministic choose pivot subroutine guarantees a 30-70 split or better. And you probably feel a little exhausted and like we deserve a QED at this point. But we haven't earned it. We have not at all proved that this deterministic selection algorithm runs in linear time. Why doesn't a guaranteed 30-70 split guarantee us linear time automatically? Well, we had to work pretty hard to figure out this element guaranteeing this 30-70 split. In particular we had to invoke another recursive call. So maybe this was a Pyrrhic victory. Maybe we had to work so hard to compute the pivot that it outweighs the benefit we'd get from this guarantee. 30 70 split. So, we still have to prove that's not the case even in conjunction doing both of these things, we still have our linear time bound. We'll finish the analysis in the next video. [sound].

# 8-5-Deterministic Selection - Analysis II [Advanced - Optional]

So the time has arrived for us to finish the proof of the fact that this deterministic algorithm based on the median of median ideas, does indeed run in linear time. We've done really all the [inaudible] difficult work. We've discussed the algorithmic ingenuity required. To choose a pivot deterministically that's guaranteed to be pretty good. So remember the idea was you take the input array, you logically break it into groups of five, you sort each group. That's like the first round of a two round knockout tournament. The winners of the first round are the middle elements of each group of five. That's the initial set of medians. And then the second around we take a median of these N over five first round winners, and that's what we return as the pivot. And we proved this key lemma which is the 30/70 lemma, which says that if you choose the pivot by this two round knockout tournament, you're guaranteed to get a 30/70 split or better. So your recursive call in line six or seven. Of having a de-select is guaranteed to be on an array that has at most 70 percent of the elements that you started with. In other words you're guaranteed to prune at least 30 percent of the array before you recurs again. But what remains to understand is whether or not we've done a sensible trade off. Have we kept the work required to compute this 30/70 split small enough. That we get the desired linear running time. Or have we, is the cost of finding a pretty good pivot outweighing the benefit of having guaranteed good splits? That's what we gotta prove. That's the next subject. Here's the story so far. You'll recall that, as usual, we define T of N to be the worst case running time of an algorithm. In this case, D select on inputs of array length. I didn't put arrays of length N. And we discussed, okay, there's the base case as usual. But in the general case, we discussed how, outside of the two recursive calls. The deselect algorithm, there's a linear number of operations. What does it have to do? It has to do the sorting, but each sorting is on a group of sized constants, size five, so it takes constant time for a group. There's a linear number of groups, so step one takes linear time, the copying takes linear time, and the partitioning takes linear time. So, there's some constant C, which is gonna be bigger than one, but it's gonna be constant. So, then outside of a recursive cause. Deselect always does at most C times N operations. Now what's up with the recursive calls. Well, remember there's two of them. First, there's one on line three that's just responsible for helping choose the pivot. This one we understand. It's always on twenty percent of the imputed rate of like the first round winners, so we can very safely write T of N over five for the work done, in the worst case, by that first recursive call. What we didn't understand until we proved the key lemma was what's up with the second recursive call, which happens on either line six or line seven. The size of the imputed rate on which we recursed depends on the quality of the pivot, and it was only when we proved the key lemma that we had a guarantee on the. [inaudible] 30-70 split or better what does that mean? That means the largest sub-array we could possibly recurs on has seven-tenths N elements. So what remains is to find the solution for this recurrence and hopefully prove that it is indeed big O event. So I'm going to go ahead and rewrite the occurrence at the to of the slide. We're not really going to worry about the T to one equal one. What we're interested in is the fact that the running time on an input of length N is at most C times N. Where again c is some constant which is gonna have to be at least one, given all the work that we do outside of the recursive calls. Plus the recursive call on line three on an array of size n over five. Plus the second recursive call, which is on some array that has size in the worst case seven-tenths n. So that's cool. This is exactly how we handle the over deterministic divide and conquer algorithms that we studied in earlier videos. We just wrote down a recurrence and then we solve the recurrence, but now, here's the trick. And all of the other recurrences that came up. For example, Merge Short, Strassner's Matrix Multiplication Algorithm, [inaudible] multiplication, you name it. We just plug the parameters into the masters method. And because of the power of the master method, boom! Out popped up an answer. It just told us what the recurrence evaluated to. Now, the master method, as powerful as it is, it did have an assumption, you might recall. The assumption was that every sub-problem solved had the same size. And that assumption is violated by this linear time selection algorithm. There are two recursive calls. One of 'ems on twenty percent of the original array. The other one is probably on much more than twenty percent of the original array. It could be as much as 70 percent of the original array. So because we have two recursive calls, and sub problems of different size, this does not fit into the situations that the master method covers. It's a very rare algorithm in that regard. Now, there are more general versions of the master method, of the master theorem which can accommodate a wider class of recurrences including this one here. Alternatively we could push the recursion tree proof so that we could get a solution for this recurrence. Some of you might want to try that at home. But I want to highlight a different way you can solve recurrences just for variety, just to give you yet another tool. Now the good news of the, about this approach that I'm gonna show you is that it's very flexible. It can be used to solve sort of arbitrarily crazy recurrences. It's certainly going to be powerful enough to evaluate this one. The bad news is that it's very out of hock. It's not very necessarily very easy to use. It's kind of a dark art figuring out how to apply it. So it's often just guess and check, is what it's called. You guess what the answer to the recurrence is and then you verify it by induction. Here, because we have such a specific target in mind, the whole point of this exercise is to prove a linear is not bound, I'm gonna call it just hope and check. So we're gonna hope there's linear of time and then we're gonna try to produce a proof of that just that verifies the linear time bound using induction. Specifically what are we hoping for, we're crossing our fingers that there's a constant, I'm going to call it A, A can be big but it's got to be constant, and again remember constant means it does not depend on N in any way. Such that our recurrence at the top of this slide T-N is bound above by A times N for all and at least one. Why is this what we're hoping? Well suppose this were true. By definition T of N is a upper bound of the running time of our algorithm. So it's bound and [inaudible] by A times N then it's obviously an O event. It's obviously a linear time algorithm. It's obviously A gets that gets suppressed in the big rotation. So that's the hope, now let's check it. And again, check mean just verify by induction on N. So the precise claim that I'm going to prove is the following. I'm gonna go ahead and choose the constant A. Remember all we need is some constant A, no matter how big as long as it's independent of N. That'll give us the big O of N time. So I'm actually gonna tell you what A I'm gonna use for convenience. I'm gonna choose A to be 10C. Now what is C? C is just a constant that we inherit from the recurrence that we're given. Now remember what this recurrence means is this is what the running time is of the deselect algorithm and the C times N represents the work that's outside of the recursive calls. So this is just a constant multiple on the amount of linear work that deselect does for sorting the groups, for doing the partitioning and for doing the copying. And so there's gonna be some small task at a reasonable cost and, and for the proof I'm just gonna multiply that by ten and use that as my A. And the claim is if I define A in that way then indeed, it is true that for all and at least one, T of N is banded above by A times N. Now, I realized I just, I pulled this constant A out of nowhere, right? Y10 times C. Well, if you recall our discussion when we proved that things were big O of something else, there again, there was some constant. So to formally prove that something is big O of something else, you have to say what the constant is. And in the proof, you always wonder how do you know what constant to use? So, in practice, when you're actually, if you have to actually do one of these proofs yourself, you reverse engineer what kind of constant would work. So you just go through the argument with a generic constant. And then you're like, oh, well, if I set the constant to be this, I can complete the proof. So we'll see, that's exactly what's gonna happen in the proof of this claim. It'll be obvious. The very last line you'll see why it shows A equals 10C. So I just reverse engineered what I needed for the proof. But to keep the proof easy to follow line by line I decided to just full disclosure tell you the cost right at the beginning. Now no prizes for guessing that the way this proof proceeds is by induction on N. Induction's the obvious thing to use, we're trying to prove an assertion for every single positive number N and moreover we're given this recurrence which relates solutions of smaller sub-problems to that of bigger problems. So that sets things up for use of the inductive hypothesis. If you want a longer review of what proofs by induction are, I suggest that you go back and re-watch the optional video where we prove the correctness of quicksort. That is, is a fairly formal discussion of what the template is like for a proof by induction. And that's the one we're gonna apply here. So, there's two ingredients in any proof by induction is, is a usually trivial one in the form of a base case. That's also gonna be trivial here. In the base case you just directly establish the assertion when n equals one. So, we're trying to prove that t of n is the most a times n for every n when n equals one we could just substitute. But what we're trying to prove is that t of one is at most a time one also known as a. And we're given that t of one is one. Right that's the base case of the recurrence that we're given. So that's what we're using here. What we want to be true is that this isn't the most a times one, but it is. So the constant c we're assuming is at least one, so it certainly can multiply c times ten to get a. It's definitely at least one. So the right hand side here is unquestionably bigger than the left hand side. A in fact is bigger than ten, let alone bigger than ten. So the interesting ingredient is generally the inductive step so remember what you do is here is you assume you've already proven the assertion that, in this case the T of N is at most AN for all smaller integers, and now you just merely have to prove it again for the current integer. So we're now interested in the case where N is bigger than one and the assumption that we've already [sound] proven to everything smaller is called inductive hypotheses. So what does it mean that we already proved it for all smaller numbers, that means we can use in the proof of our inductive step the fact that P of K is the most a times K for all K strictly less than N. All we gotta do is enlarge the range of N's to which this holds to one more to the current value N. And all we have to do is follow our nose. So pretty much, we, we have to start on the left hand side with T of N, and we have to wind up on the right hand side with A times N. And pretty much, at every step of the proof, there's just gonna be one conceivable thing we could do. So we just follow our nose. We start with what we wanna upper bound, T of N. Well, what do we got going for us? The only thing we can do at this point is invoke the recurrence that we were given up here. So we have an upper bound on T of N in terms of the T value of smaller integers. So we are given that T of N is at most C times N, plus T of N over five, plus T of seven-tenths N. Of course ignoring fractions, you would round up or round down, if you wanted to be precise, and the auxiliary lecture notes are more precise, if you want to see what the gory details look like. But it's really just exactly the same argument. One just has to be a little bit more anal about it. So now that we've invoked the recurrence, what can we possibly do, right? We can't really do any direct manipulation on any of these three terms. But fortunately, we have this inductive hypothesis. That applies to any value, any integer which is less than N. So we have her, N/5, that's certainly less than N. We have 70 percent of N. That's certainly less than N. So we can apply the inductive hypothesis twice. We already know that these T values are bounded above by A times their arguments. So T of N over 5's at most A, times N over five. T of seven-tenths N is at most A, times seven-tenths N. Now we can group terms together, not we're comparing apples to apples. So we have N, times quantity C, plus A/5, plus seven-tenths A. Let me just go ahead and group the two A turns together. And that's gonna be nine-tenths A. No, don't forget where we're going, what the end goal is. We want a upper bound T of N by AN. So we wanna write that this is bounded above by A times N. And now you see exactly how I reverse engineered our choice of A, as a function of the given constant C. Since A is ten times as big as C, if I take 90 percent of A and add C, I just get A back. So by our choice of A. This equals AN. And that pretty much wraps things up. So don't forget what all this stuff stands for. So what did we just prove? What did we just prove by induction? We proved T of N is, at most, a constant times N for every N. That is, T of N is Big O of N. What was T of N? That was the running time of our algorithm. That's all we cared about. So because T of N is Big O of N, indeed, deselect runs in O of N time.

# 8-6-Omega(n log n) Lower Bound for Comparison-Based Sorting [Advanced - Optional]

This optional video will be, more or less, the last word that we have on sorting for the purposes of this course. And it'll answer the question, can we do better? Remember, that's the mantra of any good algorithm designer. I've shown you N log N algorithms for sorting, Merge Short in the worst case, Quick Sort, on average. Can we do better than N log N? Indeed, for the selection problem, we saw we could do better than N log N. We could linear time. Maybe we can do linear time for sorting as well. The purpose of this video is to explain to you why we cannot, do sorting, in linear time. So this is a rare problem where we understand quite precisely how well it can be solved at least for a particular class of [inaudible] called comparison based sorts which I'll explain in a moment. So here's the form of the theorem, I want to give you the gist of in this video. So in addition to restricting to comparison based sorts which is necessary as we'll see in a second, I'm going to make a second assumption which is not necessary but is convenient for the lecture which is that I'm only going to think about deterministic algorithms for the moment. I encourage you to think about why the same style of arguments gives an N log and lower bound on the expected running time of any randomized algorithm. Maybe I'll put that on the course site as an optional theory problem. So, in particular, a quick sort is optimal in the randomized sense. It have average and long end time and then again claims that no comparison based sort can be better than that, even on average. So, I need to tell you what I mean by a comparison based sorting algorithm. What it means, it's a sorting algorithm that accesses the elements of the input array. Only via comparisons, it does not do any kind of direct manipulation on a single array element. All it does, is it picks pairs of elements and asks the question is the left one bigger or is the right one bigger. I like to think of comparison based sorts as general purpose sorting routines. They make no assumptions about what the data is other than that it's from some totally ordered set. I like to think of it really as a function that takes as an argument a function pointer that allows it to do comparisons between abstract data types. There's no way to access the guts of the elements. All you can do is go through this API, which allows you to make comparisons. And indeed if you look at the sorting routine and say the unit's operating system, that's exactly how it's set up. You just patch in a function pointer to a comparison operator. I know this sounds super abstract so, I think it becomes clear once we talk about some examples. There's famous examples of comparison based sort including everything we've discussed in the class so far. There's also famous examples of non comparison based sort which we're not gonna cover, but perhaps some of you have heard of or at the very least they're very easy to look up on Wikipedia or wherever. So examples include the two sorting algorithms we discussed so far, mergesort. The only way that mergesort interacts with the elements in the input array is by comparing them and by copying them. Similarly, the only think Quick Sort does with the input array elements is compare them and swap them in place. For those of you that know about the heap data structure which we'll be reviewing later in the class. Heap sort. Where you just, heapify a bunch of elements, and then extract the minimum N times. That also uses only comparisons. So what are some famous non examples? I think this will make it even more clear what we're talking about. So bucket sort is one very useful one. So, bucket sort's used most frequently when you have some kind of distributional assumption on the data that you're sorting. Remember that's exactly what I'm focusing on avoiding in this class. I'm focusing on general purpose subroutines where you don't know anything about the data. If you do know stuff about the data, bucket sorting can sometimes be a really useful method. For example, suppose you can model your data as I-I-D samples from the uniform distribution on zero one. So they're all rational numbers, bigger than zero, less than one, and you expect them to be evenly spread through that interval. Then what you can do in bucket sort is you can just. Preallocate end buckets where you're gonna collect these elements. Each one is gonna have the same width, width one over n. The first bucket you just do linear pass with the input array. Everything that's between zero and one over n you stick in the first bucket. Everything in between one over n and two over n you stick in the second bucket. Two over end and three over n you sick in the third bucket and so on. So with the single pass. You've classified the input elements according to which bucket they belong in, now because the data is assumed to be uniform at random, that means you expect each of the buckets to have a very small population, just a few elements in it. So remember if it. Elements are drawing uniform from the interval zero one, then it's equally likely to be in each of the N available buckets. And since there's N elements that means you only expect one element per bucket. So that each one is gonna have a very small population. Having bucketed the data, you can now just use, say, insertion sort on each bucket independently. You're gonna be doing insertion sort on a tiny number of elements, so that'll run in constant time, and then there's gonna be linear number of buckets, so it's linear time overall. So the upshot is. If you're willing to make really strong assumptions about your data like it's drawn uniformly at random from the interval zero one then there's not an N log in lower bound in fact you can allude the lower bound and sort them in your time. So, just to be clear. In what sense is bucket sort not comparison based? In what sense does it look at the guts of its elements and do something other than access them by pairs of comparisons? Well, it actually looks at an element at input array and it says what is its value, and it checks if its value is.17 versus.27 versus.77, and according to what value it sees inside this element, it makes the decision of which bucket to allocate it to. So, it actually stares at the guts of an element to decide how, what to do next. Another non-example, which eh, can be quite useful is count and sort. So this sorting algorithm is good when your data again we're gonna make an assumption on the data, when their integers, and their small integers, so they're between zero and K where K is say ideally at most linear in N. So then what you do, is you do a single pass through the input array. Again, you just bucket the elements according to what their value is. It's somewhere between zero and K, and it's an integer by assumption. So you need K buckets. And then you do a pass, and you sort of depopulate the buckets and copy them into an output array. And that gives you a, a sorting algorithm which runs in time, O of N Plus K. Where K is the size of the biggest integer. So the upshot with counting sort is that, if you're willing to assume that datas are integers bounded above by some factor linear in N, proportional to N, then you can sort them in linear time. Again county sort does not access the rail and it's merely through comparisons. It actually stares at an element, figures out what it's value is, and uses that value to determine what bucket to put the element in. So in that sense it's not a comparison case sort and it can under compare it's assumptions to beat the end log and lower it down. So a final example is the one that would [inaudible] them rated sort. I think that this is sort of an extension of counting sort, although you don't have to use counting sort as the interloop you can use other so called stable sorts as well. It's the stuff you can read about in many programming books or on the web. And up shot at rated sort. [inaudible]. You, you again you assume that the date are integers. You think of them in digit representation, say binary representation. And now you just sort one bit at time, starting from the least significant bits and going all the way out to the most significant bits. And so the upside of rating sort, it's an extension of counting sort is the sense that if your data is integers that are not too big, polynomially bounded in N. Then it lets you sort in linear time. So, summarizing, a comparison based sort is one that can only access the input array through this API, that lets you do comparisons between two elements. You cannot access the value of an element, so in particular you cannot do any kind of bucketing technique. Bucket sort, counting sort, and rating sort all fundamentally are doing some kind of bucketing and that's why when you're willing to make assumptions about what the data is and how you are permitted to access that data, that's when you can bypass in all of those cases, this analog and lower value. But if you're stuck with a comparison based sort, if you wanna have something. General purpose. You're gonna be doing n log n comparisons in the worst case. Let's see why. So we have to prove a lower band for every single comparison based sorting method, so a fixed one. And let's focus on a particular input length. Call it N. Okay, so now, let's simplify our lives. Now that we're focused on a comparison based sorting method, one that doesn't look at the values of the array elements just in the relative order. We may as well think of the array as just containing the elements... One, two, three, all the way up to N, in some jumbled order. Now, some other algorithm could make use of the fact that everything is small integers. But a comparison based sorting method cannot. So there's no loss in just thinking about an unsorted array containing the integers [inaudible] N inclusive. Now, depsite seemingly restricting the space of inputs that we're thinking about, even here, there's kind of a lot of different inputs we've gotta worry about, right? So N elements can, can show up, and N factorial different orderings, right? There's N choices for who the first element is, then N-1 choices for the second element, M minus two choices for the third element, and so on. So, there's N factorial for how these elements are, are arranged in the input array. So I don't wanna prove this super formally, but I wanna give you, the gist, I think, the good intuition. Now, we're interested in lower bounding the number of comparisons that, this method makes in the worst case. So let's introduce a parameter K, which is its worst case number of comparisons. That is, for every input, each of these end factorial inputs, by assumption, this method makes no more than K comparisons. The idea behind the proof is that, because we have N factorial fundamentally different inputs, the sorting method has to execute in a fundamentally different way on each of those inputs. But since the only thing that causes a branch in the execution of the sorting method is the resolution of the comparison, and we have only [inaudible] comparisons, it can only have two to the K different execution paths. So that forces two to the K to be at least N factorial. And a calculation then shows that, that forces K to be at least Omega N log N. So let me just quickly fill in the details. So cross all in-factorial possible inputs just as a thought experiment. We can imagine running this method in factorial times just looking at the pattern of how the comparison is resolved. Right? For each of these in-factorial inputs, we run it through this sorting method, it makes comparison number one, then comparison number two, then comparison number three, then comparison number four, then comparison number five, and you know it gets back a zero, then a one, then a one, then a zero. Give in some other input and it gets back a one, then a one, then a zero, then a zero and so on. The point is, for each of these in-factorial inputs, it makes at most K comparisons, we can associate that with a K bit string, and because it. Is there's only K bits we're only going to see two to the K different K-bit strings two to the K different ways that a sequence of comparisons results. Now to finish the proof we are gonna apply something which I don't get to use as much as I'd like in an evident class but it's always fun when it comes up, which is the pigeon-hole principle. The [inaudible] principle you recall is the essentially obvious fact that if you try to stuff K plus one pigeons into just K cubby holes, one of those K cubby holes has got to get two of the pigeons. Okay at least one of the cubby holes gets at least two pigeons. So for us what are the pigeons and what are the holes? So our pigeons are these in factorial different inputs. The different ways you can scramble the images one through. And, what are our holes? Those are the two indicate different executions that the sorting method can possibly take on. Now if. The number of comparisons K used is so small, that two to the K, the number of distinct execution, number of distinct ways comparisons can resolve themselves, is less than the number of different inputs that have to be correctly sorted. Then by the pivotal principal. One Color [inaudible] gets two holes. That is, two different inputs get treated in exactly the same way, by the sorting method. They are asked, exactly the same k comparisons and the comparisons resolve identically. [inaudible] one. Jumbling of one through N, then you get a 01101 then it's a totally different jumbling of N and then again you get a 01101 and if this happens the algorithm is toast, in the sense that it's definitely not correct, right, cuz we've fed it two different inputs. And it is unable to resolve which of the two it is. Right? So, it may be one premutation of one through N, or this totally different premutation of one through N. The algorithm has tried to learn about what the input is through these K comparisons, but it has exactly the same data about the input in two, the two cases. So, if it outputs the correct sorted version in one case, it's gonna get the other one wrong. So, you can't have a common execution of a sorting algorithm unscramble totally different premutations. It can't be done. So what have we learned? We've learned that by correctness, two to the K is in fact at least in the factorial. So how does that help us? Well, we wanna lower bound K. K is the number of comparisons this arbitrary storing method is using. They wanna show that's at least N log N. So we, to lower bound K, we better lower bound N factorial. So, you know, you could use Stirling's Approximation or something fancy. But we don't need anything fancy here. We'll just do something super crude. We'll just say, well, look. This is the product of N things, right? N times N minus one time N minus two, blah, blah, blah, blah. And the largest of those, the N over two largest of those N terms are all at least N over two. The rest we'll just ignore. Pretty sloppy, but it gives us a lower bound of N divided by two raised to the N divided by two. Now we'll just take log base two of both sides, and we get the K is at least N over two, log base two of N over two, also known as omega of N log N. And that my friends is why a heretics deterministic sorting algorithm that's comparison based has gotta use N log N comparisons in the worst case.

# 9-1-Graphs and Minimum Cuts

So, in this set of lectures we'll be discussing the minimum cut problem and graphs and we'll be discussing the randomized contraction algorithm. Randomize algorithm which is so simple and elegant and it's almost impossible to believe that it can possibly work but that's exactly at what we'll be approving. So one way you can think about these set of lectures, as a segue of sorts, between our discussion of randomization and our discussion of graphs. So we just finished talking about randomization in the context of sorting and searching. We'll pick it up again toward the end of the class when we discuss hashing. But while we're in the middle of randomization probability review, I'm going to give you another application of randomization in a totally different domain. In particular to the domain of graphs, rather than to sorting and searching. So that's one high level goal of these lectures. The second one, is we'll get our feet wet talking about graphs, and a lot of the next couple weeks, that's what we're going to be talking about, fundamental graph primitives. So this will give us an excuse to start warming up with the vocabulary, some of the basic concepts of the graphs and what a graph algorithm looks like. Another perk, although it's not one of the main goals, but I want to do, I do want to point this fact, compared to most of this stuff that we're discussing in this class, this is a relatively recent algorithm, the contraction algorithm. By relatively recent I mean, it's 20 years old. But at least that means most of us, I know not all of us, but most of us at least were born at the time that this algorithm was invented. And so just one quick digression. In an intro course like this, most of what we're going to cover are oldies but goodies, stuff from as much as 50 years ago. And while it's kind of amazing, given how much the world and how much technology has changed over the past 50 years, that ideas in computer science from that long ago are still useful, they are. It's just sort of an amazing thing about the stuff that the first generation of computer scientists figured out. It's still relevant to this day. That said, algorithms is still a vibrant field with lots of open questions. And when I have an opportunity, I'll try and give you glimpses of that fact. So I do want to point out here that this is a somewhat more recent algorithm than most of the other ones we'll see, which dates back to the 90s. So let's talk about graphs. Fundamentally, what a graph does is represent pair-wise relationships among a set of objects. So, as such, a graph is going to have two ingredients. So first of all, there's the objects that you're talking about. And these have two very common names and you're just going to have to know both of the names, even though they're completely synonymous. The first name is vertices. So vertex is the singular, vertices is the plural. Also known interchangeably as nodes. I'll be using the notation V for the set up of vertices. So those are the objects, now I want to represent pair wise relationships so these pairs are going to be called edges. Will be noted by, denoted by E. And there's two flavors of graphs and both are really important. Both come up all the time in applications, so you should be aware of both kinds. So there's undirected graphs and directed graphs and that just depends on whether the edges themselves are undirected or directed. So edges can be undirected by which I mean this pair is unordered. There are just two vertices of an edge the two endpoints, say U and V, and you don't distinguish one as the first and one as the second. Or edges can be directed, in which case you have a directed graph. And here, a pair is ordered, so you do have a notion of a first vertex, or a first end point. And the second vertex or second end point of an edge. Those are often called the tail and the head respectively. And once in a while, although I will try to not use this terminology, you hear directed edges called arcs. Now I think all of this is much clearer if I just draw some pictures. Indeed one use to call graphs, dots and lines. The dots would refer to the vertices, so here's four dots, or four vertices. And the edges would be lines, so the way you denote one of these edges is you just draw a line between the two end points of that edge, the two vertices that it corresponds to. So this is undirected graph with four vertices and five edges. We can equally we'll have a directed version of this graph. So let's still have four vertices and five edges, but to indicate that this is directed graph and then each edge was first vertex and the second vertex, were going to add arrows to the line. So the arrow points to the second vertex, or to the head of the edge. So, the first vertex is often called the tail of the edge. So, graphs are completely fundamental, they show up not just in computer science but in all kinds of different disciplines, social sciences and biology being two prominent ones. So, let me just mention a couple of reasons you might use them just off the top of my head but literally there's hundreds or thousands of others, so a very literal example would be road networks. So imagine you type in asking for your driving directions from point A to point B in some web application or software, or whatever, it computes a route for you. What it's doing, is it's manipulating some representation of a road network, which inevitably is going to be stored as a graph, where the vertices corresponds to intersections and the edges correspond to individual roads. The Web is often fruitfully thought of as a directed graph, so here the vertices are the individual web pages, and edges correspond to hyperlinks. So the first vertex in an edge detail is going to be the page that contains the hyperlink. The second vertex, or the head of the edge, is going to be what the hyperlink points to. So that's the Web as a directed graph. Social networks are quite naturally represented as graphs. So here the vertices correspond to the individuals in the social network. And the edges correspond to relationships. They have friendship links. I encourage you to think about among the popular social networks these days, which ones are undirected graphs and which ones are directed graphs, we have some interesting examples of each of those.. And often graphs are useful even when there isn't such an obvious network structure. So just to mention one example. Let me just write down precedence constraints. So to say what I mean, you might think about, let's say you're a freshman in college and you're looking at your majors, you're a science major and you want to know what courses to take in what order. And you can think about the following graph where each of the courses in your major corresponds to a vertex And you draw a directed edge from course A to course B, if course A is a prerequisite for course B. That is, it has to be completed before you begin course B. Okay, so that's a way to represent dependencies, sort of a temporal ordering, between pairs of objects using a directed graph. So that's the basic language of graphs. Let me now talk about cuts in graphs. because again, this set of lectures is going to be about so called minimum cut problem. So, the definition of a cut of a graph is very simple, it's just a grouping, a partition of the vertices of the graph into two groups, A and B, and both of those two groups should be non-empty. So, to describe this in pictures, let me give you a cartoon of the cut in both the undirected and directed cases. So for an undirected graph, you can imagine drawing your two sets, A and B. And once you've defined the two sets A and B, the edges then fall into one of three categories. You've got edges with both of the endpoints in A. You've got edges with both of the endpoints in B. And then, you've got edges with one endpoint in A, and one endpoint in B. So that's generically what the picture of the graph looks like viewed through the lens of a particular cut, A B. The picture for directed graphs is similar. You would again have an A, and you'd again have a B, you have directed edges with both endpoints in A, directed edges with both endpoints in B. And now you should have two further categories, so you have edges who cross the cut from left to right, that is tail vertex is in A and the head vertex is in B and you can also have edges which cross the cut in the opposite direction, that is their tail is in B and their head is in A. Usually when we talk about cuts, we're going to be concerned with how many edges cross the given cuts. And by that I mean the following, the crossing edges of a cut (A,B) are those that satisfy the following property. So in the undirected case, it's exactly what you think it would be, one of the endpoints is an A, the other endpoint is in B, that's what it means to cross the cut. Now in the directed case, there's a number of reasonable definitions you could propose, about which edges crossed the cut. Typically and in this course, we're going to focus on the case where we only think about edges that cross the cut from the left to the right, and we ignore edges which cross from the right to the left. So that is the edges that cross the cut are those with tail in A and head in B. So referring to our two pictures, our two corrections of cuts for the underrated one all three of these blue edges would be the edges crossing the cut AB. because they're the ones that have one end point on the left side and one end point on the right side. Now for the directed one, we only have two crossing edges. So the two that cross from left to right. We have tail in A and head in B. The one that's crossing backwards does not contribute. We don't count it as a crossing edge of the cut. So the next quiz is just a sanity check that you've absorbed the definition of a cut of a graph. All right, so the answer to this quiz is the third option. Recall what is the definition of a cut, it's just a way to group the vertices into two sets A and B, both should also be not empty. So we have N vertices and essentially we have one binary degree of freedom for each, for each vertex, we can decide whether or not it goes in set A or it goes in set B, so two choices for each of the N vertices, that gives us a two to the N possible choices, two to the N possible cuts overall. Now that's slightly incorrect because we call that a cut. You can't have a non empty set A or a non empty set B, so those are two of the two to the N options which are disallowed. So strictly speaking the number is two to the N minus two, but two to the N is certainly the closest answer of the four provided. Now, the minimum cut problem is exactly what you'd think it would be. I give you as input a graph and among these exponentially, many cuts, I want you to identify one for me with the fewest number of crossing edges. So a few quick comments, so first of all the name for this cut is a min cut. A min cut is one with the fewest number of crossing edges. Secondly, to clarify, I am going to allow in the input what's called parallel edges. There will be lots of applications where parallel edges are sort of pointless, but for minimum cut actually it's natural to allow parallel edges. And that means you have two edges that correspond to exactly the same pair of vertices. Finally, the more seasoned programmers among you are probably wondering what I mean by, you're given the graph as input. You might be wondering about how exactly that's represented, so the next video's going to discuss exactly that, the popular ways of representing graphs and how you're usually going to do it in this course, specifically via what's called an adjacency list. Okay, so I want to make sure that everybody understands exactly what the minimum problem is asking. So, let me draw for you a particular graph with eight vertices and quite a few edges. And what I want you to answer is what is the min cut value in this graph? That is, how many edges cross the minimum cut, the cut with the fewest number of crossing edges? All right, so the correct answer is the second choice. The min cut value is 2 and the cut which shows that, is just to break it basically in half. And there were two halves. In this case, there are only two crossing edges, this one and this one. And I'll leave it for you to check that there's no other edge that has as few as two edges. Now in this case, we got a very balanced split when we took the minimum cut. In general, that need not be true. Sometimes even a single vertex can define the minimum cut of a graph, and I encourage you to think about a concrete example that proves that. So why should you care about computing the minimum cut? Well, this is one problem among a genre called graph partitioning, where you're given a graph and you want to break it into two or more pieces. And these kinds of graph partitioning problem comes up all the time, in a surprisingly diverse array of applications. So let me just mention a couple at a high level. So one very obvious one when your graph is representing its physical network, when identifying something like a min cut allows you to do, is identify weaknesses in your network. Perhaps it's your own network, and you want to understand where you soup of the infrastructure because it's, in some sense, a hot spot of your network or a weak point. Or, maybe there's someone else's network and you want to know where the weak spot in their network. In fact, there are some declassified documents about 15 years ago or so. Which showed that the United States and Soviet Union militaries, back during the Cold War, were actually quite interested in computing minimum cuts, because they were looking for things like, for example, what's the most efficient way to disrupt the other country's transportation network? Another application, which is a big deal in social network analysis these days, is the idea of community detection. So the question is among the huge graph, say the graph of everybody who is on Facebook or something like that. How can you identify small pockets of people that seem tightly knit, that seem closely related, from which you like to infer that there are community of some sort? Maybe they all go to the same school, maybe they all have the same interest, maybe they're part of the same biological family whatever. Now, it's to some degree still an open question how to best define communities and social networks. But as a quick and dirty sort of first order heuristic, you can imagine looking for small regions, which on the one hand, are highly interconnected among themselves, but quite weakly connected to the rest of the graph. So sub-routines like the minimum cut problem, can be used for identifying these small densely interconnected, but then weakly connected to everybody else, pockets of a graph. Finally, cut problems are also used a lot in vision. So for example, one way you can use them in what's called image segmentation. So here what's going on is you're given as input a 2D array where each entry is a pixel from some image. And there's a graph, which is very natural to define, given a 2D array of pixels. Namely, you have an edge between two pixels if they are neighboring. So for two pixels that are immediately next to each other left and right or top to bottom, you put an edge there. So that gives you what's called a grid graph. And now unlike the basic minimum cut problem that we're talking about here, in image segmentation it's most natural to use edge weights. Where the weight of an edge is basically how likely you expect those two pixels to be coming from a common object. Why might you're expect to enabling pixels to come from the same object, well perhaps their color maps were almost exactly the same and you just expected that they're part of the same thing. So once you've defined the screen graph which suitable edge ways now you run a graph partitioning or maybe cut type separate team, and the hope is that the cut that it identifies rips off one of the contiguous objects in the picture. And then you do that a few times and you get the major objects in the given picture. So this list is by no means exhaustive of the applications of min cut and graph partitioning server teams, but I hope it serves as sufficient motivation to watch the rest of the lectures in this sequence.

# 9-2-Graph Representations

Okay, so this video's not about any particular graph problem, not about a, any particular graph algorithm. Just, sort of, the preliminaries we need to discuss algorithms on graphs. How do we measure their size? How do we represent them, and so on. Remember what a graph is, it really has two ingredients. First of all, there's this set of objects we're talking about. Those might be called vertices. Synonymously, we might call them nodes. We represent pair wise relationships using edges. These can be either un-directed in which case, they're ordered pairs or an edge can be directed from 1 to another. In that case, they're ordered pairs, and we have a directed graph. Now, when we talk about say, the size of a graph, or the running time of an algorithm that operates on a graph. We need to think about what we mean by input size. In particular, for a graph, there's really two different parameters that control how big it is, unlike an array. For arrays, we just had a single number, the length. For graphs, we have the number of vertices, and we have the number of edges. Usually we'll use the notation n for the number vertices, m for the number of edges. So the next quiz will ask you to think about how the number of edges m, can depend on the number of vertices, n. So, in particular, I want you to think about in this quiz, an un-directed graph It has n vertices. There's no parallel edges. 'Kay, so for a given pair of vertices, there's either zero or one edge between them. Moreover, let's assume that the graph is unconnected. 'Kay? So I don't want you to think about graphs that have zero edges. Now, I haven't defined what a graph is. What it means for a graph to be connected formally, yet, but I hope you get the idea. It means it's in one piece, you can't break it into two parts that have no edges crossing between them. So, for such a graph, no parallel edges, in one piece, n vertices, think about what is the minimum number of edges it could possibly have, and what is the maximum number of edg es, as a function of n, that it could possibly have. All right, so the correct option is the first one The fewest number of edges that a connected undirected graph we can have is n minus 1, and the maximum number of edges that an undirected graph with no parallel edges can have is n times n minus 1 over 2, better known as n choose 2. So why does it need at least n minus 1 edges, if it's going to be in one piece. Well think about at, adding the edges one at a time. Okay, on each of the edges of the graph. Now, initially, you just have a graph with zero edges, the graph has indifferent pieces and isolated vertices has no edges at all. Now each time you add one edge, what you do is you take two of the existing pieces, at best, and fuse them into one. So, the maximum decrease you can have in the number of different pieces of a graph is it can decrease by 1 each time you add an edge. So from a starting point of n different pieces, you've got to get down to 1 piece. So that requires the addition of n minus 1 edges. You can also convince yourself of this best, by drawing a few pictures and noticing that trees achieve this bound exactly, so for example here is a 4 vertex tree that has 3 edges. So this is a case where m is indeed, n minus 1. Now, for the upper bound, why can't you have more than n choose 2? Well, it's clear that the largest number of edges you can have is for the complete graph. Where every single pair of edges has 1 between them. Again, there's no parallel arcs and edges are unordered. So, there's at most, n choose 2 possibilities of where to put an edge. So again, if n equals 4, here would be an example with a maximum possible number, 6 edges. So, now that I've got you thinking about how the number of edges can vary with the number of vertices. Let me talk about the distinction between sparse and dense graphs. It's important to distinguish between these two concepts because some data structures and algorithms are better suited for sparse graphs. Other data structures and algorithms are better suited for dense graphs. So, to make this precise, let me just put down this very common notation n is the number of vertices of the graph under discussion, m is the number of inches. This is quite standard notation. Please get used to it and use it yourself. If you reverse these, you will confuse a lot of people who have familiarity with graph algorithms and data structures. Now one thing we learned from the previous quiz is the following. So in most applications, not all applications, but most applications, m is at least linear in n. Remember in the quiz we saw is at least n minus 1 if you wanted the graph to be connected, and it's also big O of n squared. This is under the assumption that there's no parallel arcs. Now, there are cases where we want to allow parallel arcs. In fact we'll do that in the contraction algorithm for the min cut problem. There are cases where we want to allow the number of edges to drop so low, that the graph breaks into multiple pieces. For example, when we talk about connected components but more often than not, we're thinking about a connected graph with no parallel edges. And then we can pin down the number of edges m to be somewhere between the linear and the number of nodes, linear and n and quadratic in it. Now I'm not going to give you a super formal definition of what a sparse or a dense graph is, and people are a little loose with this, this terminology in practice. But basically, sparse means you're closer to the lower bound, closer to linear. Dense means, you're closer to the upper bound, closer to quadratic. Now I know this leaves ambiguity when the number of edges is something you know like n to the 3 halves. usually in that case you'd think of that as a dense graph. So usually anything which is more than N times logarythmic terms, you'd think of that as a dense graph. But again, people are a little bit sloppy with this when they talk about graphs. Next I want to discuss two representations of graphs and we're mostly going to be using the s econd one in this course, but this first one, the adjacency matrix, I do want to mention just briefly, just on this slide. This is the supernatural idea where you represent the edges in a graph using a matrix. Let me describe it first for undirected graphs. So, the matrix is going to be denoted by capital A, and it says square n by n matrix where n is the number of vertices of the graph. And the semantics are the i-jth entry of the matrix is 1. If and only if there's an edge between the vertices i and j in the graph. I'm assuming here that the vertices are named 1, 2, 3, 4, et cetera all the way up to n. It's easy to add bells and whistles to the adjacency matrix to accommodate parallel edges to accommodate edge weights, which is accommodate directed arcs, directed edges. If you wanted to have parallel arcs, you could just have Aij denote the number of arcs that are between i and j. If edges have different weights, you could just have Aij be the weight of the ij edge. And for the directed graph you could use plus ones and minus ones. So if the arc is directed from i to j, you'd set i, Aij to be plus 1. If the arc is directed from j to i, you'd set Aij to minus 1. There are many metrics by which you can evaluate a data structure, or a representation. two important ones I want to discuss here. First of all, the number of resources it requires and in this context, that's the amount of space that the data structure needs. The second thing is what are the operations of the data structure supports. So let's just begin with space requirements. What are they for the adjacency matrix? Alright, so the answer at least with the sort of straight forward way of storing a matrix is n squared. And this is n dependent of the number of edges. So you could try to beat this down for sparse graphs using sparse matrix tricks. But for the basic idea of just actually representing an n by n matrix, you got n squared entries, you gotta store one bit in each whether the edge is there or not. So that's going to give yo u n squared space. The constants are, of course, very small, because you're just storing one bit per entry. But nonetheless this is quadratic in the number of vertices. Now that's going to be fine if you have a dense graph, the number of edges is as high as n squared, then you're not really wasting anything in this representation. But in a sparse graph, if m is much closer to linear, then this is a super wasteful representation. Let's talk about the ajacently list representation, this is the, the dominant one we'll be using in this class. This has several ingredients. So, first you keep track of both the vertices and the edges as independent entities. So you're going to have an array, or a list of each. And then we want these two arrays to cross-reference each other in the obvious way. So given a vertex, you want to know which edges it's involved in. Given an edge, you want to know what its endpoints are. So, let's say first, most simply, each edge is going to have two pointers, one for each of the two endpoints. And in directed graph, of course, it would keep track of which one is the head and which one is the tail. Now, each vertex is going to point to all of the edges of which it's a member. Now in an undirected graph, it's clear what I mean by that. In a directed graph, you could do it in a couple ways. Generally you'd have a vertex, keep track of all of the edges, for which it is the tail. That is, all of the edges which you can follow one hop out from the edge. If you wanted to, you can also have a second array, at a more expense of storage, where the vertex also keeps track of the edges pointing to it. The edges for which it's the head. So, let me ask you the same question I did with an adjacency matrix. What is the space required of an adjacency list, as a function of the number of edges m, and the number of vertices n, of the graph? So, the correct answer to this question is the third option, theta of m plus n, which we're going to think of as linear space in the size of the gra ph. So, this quiz is, is a little tricky. So, it's explain the answer when we return to the slide with the ingredients of adjacency lists. And let's compute the space for each of these four ingredients separately. Most of them are straightforward. For example, consider the first ingredient. This is just an array, or a list of the n vertices. And we just need constant space per vertex to keep track of its existence. So this is going to be theta of n, linear in the number of vertices. Similarly, for the m edges, we just need linear space in the number of edges to remember their existence. So that's going to be theta of m. Now, each edge has to keep track of both of its endpoints. So that's two pointers, but two is a constant. For each of the m edges, we have a constant space to keep track of endpoints. So that's going to give us another theta of m constant per edge. Now, this fourth case, you might be feeling kind of nervous, because a vertex, in principle could have edges involving all n minus 1 of the vertices. So the number of point or is it a single vertex could be theta of n. Also you could have you know, you do have n vertices that could be theta of n squared. And certainly in something like a complete graph you really would have that function. But the point is in sparse graphs n, n squared is way overkill to the space needed by this fourth set of pointers. Actually, if you think about it for each pointer in the fourth category, a vertex pointing to a given edge, there is a pointer in the third category pointing in the opposite direction, from that edge back to that vertex. So, there's actually a one to one correspondence. Between pointers in the third category, and pointers in the fourth category. Since the third category has space theta of m, so does all of the pointers in the fourth category. So adding up over the four ingredients, we have one theta of n, and three theta of ms, so that's going to give us overall a theta of m plus n. If you prefer, another way you could think about this would be theta of the max of m and n. These are the same up to a constant factor. Now, as we discussed in a previous slide. Often, m is going to be bigger than n, but I wanted to do a generic analysis here, which applies even if the graph is not connected, even, even if it is in multiple pieces. So the space of the adjacency list is within a constant factor the same as the number of ingredients in the graph, the number of vertices plus the number of edges. So in that sense, that's exactly what you want. Now being confronted with these two graph representations that I've shown you I'm sure you're asking, well, which one should you remember? Which one should you use? And the answer, as it so often is, is it depends. It depends on two things. It depends on the density of your graph. It depends on how m compares to n. And it also depends on what kind of operations that you support, want to support. Now given what we're covering in this class, and also the motivating applications I have in mind I can give you basically a clean answer to this question for the purposes of these five weeks. Which is we're going to be focusing on adjacency lists. The reason we're going to focus on adjacency lists in this class, is both, is for both of these reasons, both because of the operations we want and both because of the graph density and motivating applications. So, first of all, most of the graph primitives, not all, but most, will be dealing with graph search and adjacency lists are perfect for doing graph search. You get to a node. You follow an outgoing arc. You go to another node. You follow an outgoing arc and so on. And so, adjacency lists are the perfect thing to do graph search. Adjacency matrices are definitely good for certain kinds of graph operations. But they're not things we're really going to be covering in this class. So that's reason one. Reason two is, a lot of the motivations for graph primitives these days comes from massive, massive networks. I mentioned earlier how the web ca n be fruitfully thought of as a directed graph. Where the vertices are individual web pages. And directed arcs correspond to hyperlinks, going from the page with the hyperlink, pointing to the one that the hyperlink goes to. Now, it's hard to get an exact measurement of the web graph, but a conservative lower bound on the number of vertices is something like 10 billion. So that's 10 to the 10. Now that's pushing the limits of what computers can do, but it's within the limits. So if you work hard, you can actually operate on graphs with 10 to the 10 nodes. Now, suppose we use an adjacency matrix representation. So if n is 10 to the 10, then n squared is going to be like 10 to the 20. And now we're getting close to the estimated number of atoms in the known universe. So that is clearly not feasible now and it's not going to be feasible ever. So the adjacency matrix representation is totally out for, huge sparse graphs like the web graph. Adjacency lists, well, the degree, on average, in the web, is thought to be something like 10. So, the number of edges is only going to be something like 10 to the 11. And then the adjacency of this representation will be proportional to that. And again, that's really pushing what we can do with current technology, but it is within the limits, so using that representation we can do non-trivial computations on graphs, even at the scale of the web graph.

# 9-3-Random Contraction Algorithm

So now I get to tell you about the very cool randomized contraction algorithm for computing the minimum cut of a graph. Let's just recall what the minimum cut problem is. We're given as input an undirected graph. And the parallel edges are allowed. In fact, they will arise naturally throughout the course of the algorithm. That is, we're given pair of vertices, which have multiple edges which have that pair as endpoints. Now, I do sort of assume you've watched the other video on how graphs are actually represented, although that's not going to play a major role in the description of this particular algorithm. And, again, the goal is to compute the cut. So, a cut is a partition of the graph vertices into two groups, A and B. The number of edges crossing the cut is simply those that have one endpoint on each side. And amongst all the exponentially possible cuts, we want to identify one that has The fewest number of crossing edges, or a "min cut". >>So, here's the random contraction algorithm. So, this algorithm was devised by David Karger back when he was an early Ph.D student here at Stanford, and this was in the early 90s. So like I said, quote unquote only about twenty years ago. And the basic idea is to use random sampling. Now, we'd known forever, right, ever since QuickSort, that random sampling could be a good idea in certain context, in particular when you're sorting and searching. Now one of the things that was such a breakthrough about Karger's contraction algorithm is, it showed that random sampling can be extremely effective for fundamental graph problems. >>So here's how it works. We're just gonna have one main loop. Each iteration of this while-Loop is going to decrease the number of vertices in the graph by 1, and we're gonna terminate when we get down to just two vertices remaining. Now, in a given iteration, here's the random sampling: amongst all of the edges that remain in the graph to this point, we're going to choose one of those edges uniformly at random. Each edge is equally likely. Once you've chosen an edge, that's when we do the contraction. So we take the two endpoints of the edge, call them the vertex u and the vertex v, and we fuse them into a single vertex that represents both of them. This may become more clear when I go through a couple examples on the next couple of slides. This merging may create parallel edges, even if you didn't have them before. That's okay. We're gonna leave the parallel edges. And it may create a self-loop edge pointer that both of the endpoints is the same. And self-loops are stupid, so we're just gonna delete as they arise. Each generation decreases the number of vertices that remain. We start with N vertices. We end up with 2. So after N-2 generations, that's when we stop and at that point we return the cuts represented by those two final vertices. You might well be wondering what I mean by the cut represented by the final two vertices. But I think that will become clear in the examples, which I'll proceed to now. >>So suppose the input graph is the following four node, four edge graph. There's a square plus one diagonal. So, how would the contraction algorithm work on this graph? Well, of course, it's a randomized algorithm so it could work in different ways. And so, we're gonna look at two different trajectories. In the first iteration each of these five edges is equally likely. Each is chosen for contraction with twenty percent probability. For concreteness, let's say that the algorithm happens to choose this edge to contract, to fuse the two endpoints. After the fusion these two vertices on the left have become one, whereas the two vertices on the right are still hanging around like they always were. So, the edge between the two original vertices is unchanged. The contracted edge between the two vertices on the left has gotten sucked up, so that's gone. And so what remains are these two edges here. The edge on top, and the diagonal. And those are now parallel edges, between the fused node and the upper right node. And then I also shouldn't forget the bottom edge, which is edge from the lower right node to the super node. So that's what we mean by taking a pair of the vertices and contracting them. The edge that was previously connected with them vanishes, and then all the other edges just get pulled into the fusion. >>So that's the first iteration of Karger's algorithm of one possible execution. So now we proceed to the second iteration of the contraction algorithm, and the same thing happens all over again. We pick an edge, uniformly at random. Now there's only four edges that remain, each of which is equally likely to be chosen, so the 25% probability. For concreteness, let's say that in the second iteration, we wind up choosing one of the two parallel edges, say this one here. So what happens? Well, now, instead of three vertices, we go down to 2. We have the original bottom right vertex that hasn't participated in any contractions at all, so that's as it was. And then we have the second vertex, which actually represents diffusion of all of the other three vertices. So two of them were fused, the leftmost vertices were fused in iteration 1. And now the upper right vertex got fused into with them to create this super node representing three original vertices. So, what happens to the four edges? Well, the contracted one disappears. That just gets sucked into the super node, and we never see it again. Again, and then the other three go, and where there's, go where they're supposed to go. So there's the edge that used to be the right most edge. That has no hash mark. There's the edge with two hash marks. That goes between the, the same two nodes that it did before. Just the super node is now an even bigger node representing three nodes. And then the edge which was parallel to the one that we contracted, the other one with a hash mark becomes a self-loop. And remember what the, what the algorithm does is, whenever self loops like this appear, they get deleted automatically. And now that we've done our N-2 iterations, we're down to just two nodes. We return the corresponding cut. By corresponding cut, what I mean is, one group of the cut is the vertices that got fused into each other, and wound up corresponding to the super node. In this case, everything but the bottom right node, And then the other group is the original nodes corresponding to the other super node of the contracted graphs, which, in this case, in just the bottom right node by itself. So this Set A is going to be these three nodes here, which all got fused into each other, contracted into each other. And B is going to be this node over here which never participated in any contractions at all. And what's cool is, you'll notice, this does, in fact, define a min cut. There are two edges crossing this cut. This one, the rightmost one and the bottommost one. And I'll leave it for you to check that there is no cut in this graph with fewer than two crossing edges, so this is in fact a min cut. >>Of course, this is a randomized algorithm, and randomized algorithms can behave differently on different executions. So let's look at a second possible execution of the contraction algorithm on this exact same input. Let's even suppose the first iteration goes about in exactly the same way. So, in particular, this leftmost edge is gonna get chosen in the first iteration. Then instead of choosing one of the two parallel edges, which suppose that we choose the rightmost edge to contract in the second iteration. Totally possible, 25% chance that it's gonna happen. Now what happens after the contraction? Well, again, we're gonna be left with two nodes, no surprise there. The contracted node gets sucked into oblivion and vanishes. But the other three edges, the ones with the hash marks, all stick around, and become parallel edges between these two final nodes. This, again, corresponds to a cut (A, B), where A is the left two vertices, and B is the right two vertices. Now, this cut you'll notice has three crossing edges, and we've already seen that there is a cut with two crossing edges. Therefore, this is <i>not</i> a min cut. >>So what have we learned? We've learned that, the contractual algorithm sometimes identifies the min cut, and sometimes it does not. It depends on the random choices that it makes. It depends on which edges it chooses to randomly contract. So the obvious question is, you know, is this a useful algorithm. So in particular, what is the probability that it gets the right answer? We know it's bigger than 0, and we know it's less than 1. Is it close to 1, or is it close to 0? So we find ourselves in a familiar position. We have what seems like a quite sweet algorithm, this random contraction algorithm. And we don't really know if it's good or not. We don't really know how often it works, and we're going to need to do a little bit of math to answer that question. So in particular, we'll need some conditional probability. So for those of you, who need a refresher, go to your favorite source, or you can watch the Probability Review Part II, to get a refresher on conditional probability and independence. Once you have that in your mathematical toolbox, we'll be able to totally nail this question. Get a very precise answer to exactly how frequently the contraction algorithm successfully computes the minimum cut.

# 9-4-Analysis of Contraction Algorithm

So in the last video I left you with a cliffhanger. I introduced you to the minimum cut problem. I introduced you to a very simple algorithm, randomized algorithm, in the form of contraction algorithm. We observed that sometimes it finds the main cut and sometimes it doesn't. And so the $64000 question is just how frequently does it succeed and just how frequently does it fail. So now that I hope you've brushed up the conditional probability and independence, we are gonna give a very precise answer to that question in this lecture. >>So recalling this problem we are given as input in undirected graph, possibly with parallel edges, and that the goal is to compute among the exponential number of possible different cuts, that's with the fewest number of crossing edges. So, for example in this graph here, which you've seen in a previous video, the goal is to compute the cut (A, B). Here, cuz there are only two crossing edges, and that's as small as it gets. That's the minimum cut problem and Karger proposed the following random contraction algorithm based on random sampling, so we have N-2 iterations, and the number of vertices gets decremented by 1 in each iteration. So we start with N vertices, we get down to 2. And how do we decrease the number of vertices? We do it by contracting or fusing two vertices together. How do we pick which pair of edges, which pair of vertices to fuse? Well we pick one of the remaining edges, uniformly at random. So there's [inaudible] many edges there are remaining. We pick each one, equally likely. What, if the endpoints of that edge are (u, v), then we collapse u and v together into a single super node. So that's what we mean by contracting two nodes into a single vertex and then if that causes any self-loops, and as we saw the examples, we will in general have self-loops, then we delete them before proceeding. After the N-2 generations, only two vertices remain. You'll recall that two vertices naturally correspond to a cut. The first group of the cut A corresponds to the vertices that were fused into one of the super vertices remaining at the end. The other super vertex corresponds to the set B the other original vertices of the graph. >>So the goal of this lec, of this video is to give an answer to the following question: What is the probability of success? Where by success, we mean outputs a particular min cut (A, B). So let's set up the basic notation. We're gonna fix any with input graph, undirected graph. As usual we use N to denote the number of vertices and M to denote the number of edges. We're also going to fix a minimum cuts (A, B). If a graph has only one minimum cut, then it's clear what I'm talking about here. If a graph has multiple minimum cuts, I'm actually selecting just one of them. Because I'm gonna focus on a distinguished minimum cut (A, B), and we're only gonna define the algorithm as successful if it outputs this particular minimum cut (A, B). If it outputs some other minimum cut, we don't count it. We don't count it as successful. Okay. So, we really want this distinguished minimum cut (A, B). In addition to N and M, we're gonna have a parameter K, which is the size of the minimum cut. That is, it's the number of crossing edges of a minimum cut. For example, that cross (A, B). The K edges that cross the minimum cut (A, B); we're going to call capital F. So the picture you wanna have in mind is, there is, out there in the world, this minimum cut (A, B). There's lots of edges with both end points in A, lots of edges possibly with both endpoints in B. But, there's not a whole lot of edges with one endpoint in A and one in endpoint in B. So the edges F, would be precisely, these three crossing edges here. >>So our next step is to get a very clear understanding of exactly when the execution of the contraction algorithm can go wrong, and exactly when it's gonna work, exactly when we're going to succeed. So let me redraw the same picture from the previous slide. So given they were hoping that the contraction algorithm outputs this cut (A, B) at the end of the day, what could possibly go wrong? Well, to see what could go wrong, suppose,, at some iteration, one of the edges in capital F, remember F are the edges crossing the min cut (A, B), so it's these three magenta edges in the picture. Suppose at some iteration one of the edges of F gets chosen for contraction. Well because this edge of F has one endpoint in A and one endpoint in B, when it gets chosen for contraction, it causes this node from A and this node from B to be fused together. What does that mean? That means, in the cut that the contraction algorithm finally outputs, this node from A and this node from B will be on the same side of the output cut. Okay, so the cut output by the contraction algorithm will have on one side both the node from A and the node from B. Therefore, the output of the contraction algorithm if this happens will be a different cut than (A, B), okay? It will not output (A, B) if some edge of F is contracted. >>And if you think about it, the converse is also true. So let's assume now, that in each of the N-2 iterations, the contraction algorithm never contracts an edge from capital F. Remember capital F are exactly the edges with one endpoint in A and one endpoint in B. So if it never contracts any edge of F, then it only contracts edges where both endpoints lie in capital A or both endpoints lie in capital B. Well, if this is the case then, vertices from A always stick together in the fused nodes, and vertices from B always stick together in the fused nodes. There is never A iteration where a node from A and a node from B are fused together. What does that mean? That means that when the algorithm outputs <i>cuts</i> all of the nodes in A have been grouped together, all of the nodes in B have been grouped together, in each of the two super nodes, which means that the output of the algorithm is indeed the desired cut (A, B). Summarizing, the contraction algorithm will do what we want. It will succeed and output the cut (A, B), if and only if it never chooses an edge from capital F for contraction. Therefore, the probability of success, that is, the probability that the output is the distinguished min cut (A, B), is exactly the probability that never contracts an edge of capital F. >>So, this is what we're gonna be interested in here. This really is the object of our mathematical analysis, the probability that in all of the N-2 iterations we never contact an edge of capital F. So, to think about that, let's think about each iteration in isolation, and actually define some events describing that. So for an iteration I, let Si denote the event, that we screw up an iteration I. With this notation, we can succinctly say what our goal is, so, to compute the probability of success. What we wanna do is we wanna compute the probability that <i>none</i> of the events, S1, S2 up to N minus, S(N-2) never occur. So, I'm gonna use this NOT(卢) symbol to say that S1 does not happen. So we don't screw up in iteration 1, we don't screw up in iteration 2, we don't screw up in iteration 3, and so on. All the way up to, we don't screw up, we don't contract anything from capital F, in the final iteration, either. So summarizing, analyzing the success probability of the contraction algorithm boils down to analyzing the probability of this event, the intersection of the NOT Sis with I ranging from iteration 1 to iteration N-2. >>So we're gonna take this in baby steps, and the next quiz will lead you through the first one, which is, let's have a more modest goal. Let's just think about iteration 1. Let's try and understand, what's the chance we screw up, what's the chance we don't screw up, just in the first iteration? So the answer to this quiz is the second option. The probability is K/M, where K is the number edges crossing the cut (A, B), and M is the total number of edges. And that's just because the probability of S1, the probability we screw up, is just the number of crossing edges. That's the number of outcomes which are bad, which cause which trigger S1, divided by the number of edges. That's the total number of things that could happen. And since all edges are equally likely, it just boils down to this. And by the definition of our notation, this is exactly K/M. So this gives us an exact calculation of the failure probability in the first iteration, as a function of the number of crossing edges, and the number of overall edges. Now, it turns out it's gonna be more useful for us to have a bound not quite as exact, an inequality. That's in terms of the number of vertices N, rather than the number of edges, M. The reason for that is, it's a little hard to understand how the number of edges is changing in each iteration. It's certainly going down by 1 in each iteration, because we contract that in each iteration, but it might go down by more than 1 when we delete self-loops. By contrast the number of vertices is this very steady obvious process. One less vertex with each successive iteration. >>So, let's rewrite this bound in terms of the number of vertices N. To do that in a useful way, we make the following key observation. I claim that, in the original graph G, we are given as input, every vertex has at least K edges incident on it, that is in graph theoretic terminology, every edge has degree at least K. Where, recall, K is the number of edges crossing our favorite min cut (A, B). So why is that true? Why must every vertex have a decent number of neighbors, a decent number of edges incident to it. Well, it's because, if you think about it, each vertex defines a cut by itself. Remember, a cut is just any grouping into other vertices into two groups, that are not empty, that don't overlap. So one cut is to take a single vertex, and make that the first group, A, and take the other N-1 vertices, and make that the second group, capital B. So how many edges cross this cut? Well, it's exactly the edges that are incident on the first note, on the note on the left side. So every single cut, fall exponentially many cuts, have at least K crossing edges, then certainly the N cuts defined by single vertices have at least K crossing edges, so therefore, the degree of a vertex is at least K. So our assumption that every single cut in the graph has at least K crossing edges because it's a lower bound on the number edges incident on each possible vertex. >>So, why is that usual? Well let's recall the following general facts about any graph; which is that if you sum up over the degrees of the nodes, so if you go node by node, look at how many edges are insident on that node, that's the degree of V, and then sum them up over all vertices. What will you get? You'll get exactly twice the number of edges, okay? So this is true for any undirected graph, that the sum of the degrees of the vertices is exactly double- the number of edges. To see this, you might think about taking a graph, starting with the empty set of edges, and then adding the edges of the graph one at a time. Each time you add a new edge to a graph, obviously the number of edges goes up by 1, and the degree of each of the endpoints of that edge also go up by 1, and there are, of course, two endpoints. So every time you add an edge, the number of edges goes up by 1, the sum of those degrees goes up by 2. Therefore, when you've added all the edges, the sum of the degrees is double the number of edges that you've added. That's why this is true. Now, in this graph, at that we have a hand here, every degree is at least K, and there's N nodes. So this left hand side, of course, is at least KN for us. So therefore if we just divide through by 2, and flip the inequality around, we have the number of edges has to be at least the size of the crossing cut, so the degrees of every vertex times the number of vertices divided by 2. So this is just the primitive inequality rearranging. Putting this together with your answer on the quiz, since the probability of S1 is exactly K/M, and M is at least KN/2, if we substitute, we get that the probability of S1 is at worst 2/N, 2 over the number of vertices, and the K cancels out. So that's, sort of, our first milestone. We've figured out the chance that we screw up in the first iteration, that we pick some edge from the crosses the cut (A, B). And things look good. This is a, this is a small number, right? So, in general, the number of vertices might be quite big. And this says that the probability we screw up is only 2 over the number of vertices. So, so far, so good. Of course, this was only the first iteration. Who knows what happens later? >>So now that we understand the chances of screwing up in the first iteration, let's take our next baby step, and understand the probability that we don't screw up in either of the first two iterations. That is, we're gonna be interested. And the following probability. The probability that we don't screw up in the first iteration nor in the second iteration. Now, as you go back to the definition of a conditional probability, to realize that we can rewrite an intersection like this in terms of conditional probabilities. Namely, as the probability that we don't screw up in the second iteration, given that we didn't do it already, times the probability that we didn't screw up in the first iteration. Okay? So the probability that we miss all of these K vulnerable edges and in the second iteration given that we didn't contract any of them in the first iteration. Now notice this, we already have a good understanding on the previous slide. We are given a nice lower bound of this. We say there's a good chance that we don't screw up, probably at least 1-2/N. And in some sense we also have a very good understanding of this probability. We know this is 1 minus the chance that we do screw up. And what's the chance that we do screw up? Well, these K edges are still hanging out in the graph. Remember we didn't contract any, in the first iteration that's what's given. So there are K ways to screw up, and we choose an edge to contract uniformly at random, so the total number of choices is the number of remaining edges. >>Now the problem is, what's nice is we have an exact understanding of this probability. This is an equality. The problem is we don't have a good understanding of this denominator. How many remaining edges are there? We have an upper bound on this. We know this is at most N-1, assuming we got rid of one edge in the previous iteration, but actually what, if you think about it, what we need of this quantity is a lower bound and that's a little unclear because in addition to contracting the one edge and getting that out of the graph, we might have created a bunch of self loops and deleted all events. So it's hard to understand exactly what this quantity is. So instead we're gonna rewrite this bound in terms of the numbers of the remaining vertices, and of course we know it's exactly N-1 vertices remaining. We took two of the last iterations and contracted down to 1. So how do we relate the number of edges to the number of vertices? Well we do it just in exactly the same way as in the first iteration. We'll make some more general observation. In the first iteration, we observed that every node in the original graph induces a cut. Okay, with that node was on one side, the other N-1 edges were on the other side. But the fact that's a more general statement, even after we've done a bunch of contractions, any single node in the contracted graph, even if it represents a union of a bunch of nodes in the original graph, we can still think of that as a cut in the original graph. Right? So if there's some super node in the contracted graph, which is the result of fusing twelve different things together, that corresponds to a cut where those twelve nodes in the original graph are on the one side A, and the other N-12 vertices are on the other side of the cut, B. So, even after contractions, as long as we have at least two nodes in our contracted graph, you can take any node and think of it as half of a cut, one side of a cut in the original graph. >>Now remember, K is the number of edges crossing our minimum cut (A, B), so any cuts in the original graph G has to have K crossing edges. So, since every node in the contracted graph naturally maps over to a cut in the original graph with at least K edges crossing it, that means, in the contracted graph, all of the degrees have to be at least K. If you ever had a node in the contracted graph that had only say K-1 incident edges, well then you'd have a cut in the original graph with only K-1 edges contradiction. So just like in the first iteration, now that we have a lower bound on the degree of every single vertex, we can derive a lower bound on the number of edges that remain in the graph. The number of remaining edges is at least 1/2, that's because when you sum over the degrees of the vertices, you double count the edges, times the degree of each vertex, that we just argued that that's at least K in this contracted graph, times the number of vertices, that we know there's exactly N-1 vertices left in the graph at this point. So now what we do is to plug this inequality, to plug this lower bound of the number of remaining edges, on, as we'll substitute that for this denominator, so in lower bounding the denominator, we upper bound this fraction, which gives us a lower bound on 1 minus that fraction, and that's what we want. So what we find is that the probability that we don't screw up in the second iteration given that we didn't screw up in the first iteration. Where again, by screwing up means picking one of these K edges crossing (A, B) to contract is at least 1-(2/(N-1)). So, that's pretty cool. We took the first iteration, we analyzed it, we showed the probability that we screw up is pretty low, we succeed with probability of at least 1-(2/N). In the second iteration, our success probability has dropped a little bit, but it's still looking pretty good for reasonable values of N, 1-(2/(N-1)). >>Now, as I hope you've picked up, we can generalize this pattern to any number of iterations, so that the degree of every node of the contracted graph remains at least K. The only thing which is changing is the number of vertices is dropping by 1. So, extending this pattern to its logical conclusion, we get the following lower bound on the probability that the contraction algorithm succeeds. The probability that the contraction algorithm outputs the cut (A, B), you recall we argued, is exactly the same thing as the probability that it doesn't contract anything, any of the K crossing edges, any of the set F in the first iteration, nor in the second iteration, nor in the third iteration, and then so on, all the way up to the final (N-2)th iteration. Using the definition of conditional probability, this is just the probability that we don't screw up in the first iteration, times the probability that we don't screw up in the second iteration given that we didn't screw up in the first iteration, and so on. In the previous two slides, we showed that, we don't screw up in the first iteration, with probability of at least 1-(2/N). In the second iteration, with probability at least 1-(2/(N-1)). And of course, you can guess what that pattern looks like. And that results in the following product. Now, because we stop when we get down to two nodes remaining, the last iteration in which we actually make a contraction, there are three nodes. And then, the second to last iteration in which we make a contraction, there are four nodes. So that's where these last two terms come from. Rewriting, this is just (N-2)/N times (N-3)/(N-1), and so on. And now something very cool happens, which is massive cancellation, and to this day, this is always just incredibly satisfying to be able to cross out so many terms. So you get N-2, cross it out here and now here, there's going to be a pair of N-3s that get crossed out, and N-4s, and so on. On the other side, there's going to be a pair of 4s that get crossed out, and a pair of 3s that get crossed out. And we'll be left with only the two largest terms on the denominator, and the two smallest terms in the numerator, which is exactly 2/N(N-1). And to keep things simple among friends, let's just be sloppy and lower bound this by 1/(N^2). So that's it. That's our analysis of the success probability of Karger's contraction algorithm. Pretty cool, pretty slick, huh? >>Okay, I'll concede, probably you're thinking "Hey, wait a minute. We're analyzing the probability that the algorithm succeeds, and we're thinking of the number of vertices N as being big, so we'll see here as a success probability of only 1/(N^2), and that kinda sucks." So that's a good point. Let me address that problem. This is a low success probability. So that's disappointing. So why are we talking about this algorithm, or this analysis? Well, here's something I want to point out. Maybe this is not so good, 1/(N^2) you're going to succeed, but this is still actually shockingly high for an brute-forth algorithm which honestly seems to be doing almost nothing. This is a nontrivial lower bound and non trivial success probability, because don't forget, there's an exponential number of cuts in the graph. So if you try to just pick a random cut i.e you put every vertex 50:50 left or right, you'll be doing way worse than this. You'll have a success probability of like 1/(2^N). So this is way, way better than that. And the fact that its inverse polynomial means is that using repeated trials, we can turn a success probability that's incredibly small into a failure probability that's incredibly small. So lemme show you how to do that next. >>So, we're gonna boost the success probability of the contraction algorithm in, if you think about it a totally obvious way. We're gonna run it a bunch of times, each one independently using a fresh batch of random coins. And we're just going to remember the smallest cut that we ever see, and that's what we're gonna return, the best of a bunch of repeated trials. Now the question is, how many trials are we gonna need before we're pretty confident that we actually find the meant cut that we're looking for? To answer this question vigorously, let's define some suitable events. So by Ti, I mean the event at the Ith trail succeeds, that is the Ith time we run the contraction algorithm which does output that desired meant cut (A, B). For those of you that watched the part II of the probability review, I said a rule of thumb for dealing with independents is that, you should maybe, as a working hypothesis, assume granted variables are dependent, unless they're explicitly constructed to be independent. So here's a case where we're just gonna define the random variables to be independent. We're just gonna say that we run [inaudible] the contraction algorithm over and over again with fresh randomness so that they're gonna be independent trials. Now, we know that the, probability that a single trial fails can be pretty big, could be as big as 1-1/(N^2). But, here, now, with repeated trials, we're only in trouble if every single trial fails. If even one succeeds, then we find the meant cut. So a different way of saying that is we're interested in the intersection of T1 and T2 and so on, that's the event that every single trial fails. And now we use the fact that the trials are independent. So, the probability that all of these things happen is just the product of the relevant probabilities. So, the product from I=1 to capital N of the probability of not TI. Recall that we argued that the success probability of a single trial was bounded below by 1/(N^2). So the failure probability is bounded above by 1-1/(N^2). So since that's true for each of the capital N terms, you get an overall failure probability for all capital N trials of 1 minus 1/(n^2) raised to the capital of N. Alright, so that's a little calculation. Don't lose sight of why we're doing the calculation. We want to answer this question, how many trials do we need? How big does capital N need to be before are confident that we get the answer that we want? >>Okay, so to answer that question I need a quick calculus fact, which is both very simple and very useful. So for all real numbers X, we have the following inequality, 1+x is bound above by e^x. So I'll just give you a quick proof via picture. So first think about the line 1+x. What does that cross through? Well, that crosses through the points when x is -1, y is 0, and when x is 0, y is 1. And it's a line, so this looks like this blue line here. What does e^x look like? Well, if you substitute x = 0, it's gonna be 1. So in fact two curves kiss each other at x = 0. But exponentials grow really quickly, so as you jack up x to higher positive numbers, it becomes very, very steep. And for x negative numbers it stays non-negative the whole way. So this sort of flattens out for the negative numbers. So, pictorially, and I encourage you to, you know, type this into your own favorite graphing program. You see the e^x bounds above everywhere, the line, the 1+x. For those of you who want something more rigorous, there's a bunch of ways to do it. For example, you can look at the [inaudible] expansion of e^x at the point 0. >>What's the point? The point is this allows us to do some very simple calculations on our previous upper bound on the failure probability by working with exponentials instead of working with these ugly one minus whatevers raised to the whatever term. So, let's combine our upper bound from the previous slide with the upper bound provided by the calculus fact. And to be concrete, let's substitute some particular number of capital N. So, let's use little n^2 trials, where little n is the number of vertices of the graph. In which case, the probability that every single trial fails to recover the cut (A, B) is bounded above by e to the -1/(N^2). That's using the calculus fact applied with X = -1/(N^2). And then we inherit the capital N and the exponent which we just substantiated to little n^2. So of course the N^2 are gonna cancel, this is gonna give us E^(-1), also known as 1/E. So if we're willing to do little n^2 trials, then our failure probability has gone from something very close to 1, to something which is more like, say, 30 some more percent. Now, once you get to a constant success probability, it's very easy to boost it further by just doing a few more trials. So if we just add a natural log factor, so instead of a little n^2 trials, we do little n^2 times the natural log of the little n. Now, the probability that everything fails is bound and above by the 1/e that we had last time, but still with the residual natural log of N up top. And this is now, merely 1/N. So I hope it's clear what happened. We took a very simple, very elegant algorithm, that almost always didn't do what we want. It almost always failed to output the cut (A, B). It did it with only probability 1/(n^2). But, 1/(n^2) is still big enough that we can boost it, so that it almost always succeeds just by doing repeated trials. And the number of repeated trials that we need is the reciprocal of its original success probability boosted by, for the logarithmic factor. So that transformed this almost always failing algorithm into an almost always succeeding algorithm. And that's a more general less, more general algorithm technique, which is certainly worth remembering. >>Let me conclude with a couple comments about the running time. This is probably the first algorithm of a course, of the course where we haven't obsessed over just what the running time is. And I said, it's simple enough. It's not hard to figure out what it is, but it's actually not that impressive. And that's why I haven't been obsessing over it. This is not almost linear. This is not a for free primitive as I've described it here. So it's certainly a polynomial-time algorithm; its running time is bounded above by some polynomial in n and m. So it's way better than the exponential time you get from brute-force search through all 2^n possible cuts. But it is certainly, the way I've described it, we gotta to n^2 trials, plus a log factor, which I'm not even going to bother writing down. And also, each trial, while at the very least, you look at all the edges, so that's going to be another factor of M. So this is a bigger polynomial than in any, almost any of the algorithms that we're going to see. Now, I don't wanna undersell this application of random sampling in computing cuts because I've just shown you the simplest, most elegant, most basic, but therefore also the slowest implementation of using contractions to compute cuts. There's been follow-up work with a lot of extra optimizations, in particular, doing stuff much more clever than just repeated trials, so basically using work that you did in previous trials to inform how you look for cuts in subsequent trials. And you can shave large factors off of the running time. So there are much better implementations of this randomized contraction algorithm than what I'm showing you here. Those are, however, outside the course, scope of this course.

# 9-5-Counting Minimum Cuts

So this is short optional video, really just for fun, I want to point out an interesting consequence tracking algorithm has about a problem that is in pure graph theory. So, to motivate the question, I want to remind you of something we discussed in passing, which is that a graph may have more than one minimum cut. So, there may be distinct cuts which are tied for the fewest number of crossing edges. For a concrete example, you could think about a tree. So, if you just look at a star graph, that is hubs and spokes, it's evident that if you isolate any leaf by itself, then you get a minimum cut with exactly one crossing edge. In fact, if you think about it for a little while, you'll see that in any tree you'll have N-1 different minimum cuts, each with exactly one crossing edge. >>The question concerns counting the number of minimum cuts. Namely, given that a graph may have more than one minimum cut, what is the largest number of minimum cuts that a graph with N vertices can have? We know the answer is at least N-1. We already discussed how trees have N-1 distinct minimum cuts. We know the answer at most something like 2^N, because a graph only has roughly 2^N cuts. In fact, the answer is both very nice and wedged in between. So the answer is exactly N choose 2, where N is the number of vertices. This is also known as N(N-1)/2. So it can be bigger than it is in trees, but not a lot bigger. In particular, graphs have only; undirected graphs have only polynomially many minimum cuts. And that's been a useful fact in a number of different applications. So, I'm going to prove this back to you. All I need is one short slide on the lower bound and then one slide for the upper bound, which follows from properties of the random contraction algorithm. >>So for the lower bound, we don't have to look much beyond our trees example. We're just gonna look at cycles. So for any value of N, consider the N cycle. So here, for example, is the N cycle with N = 8. That would be an octagon. And the key observation is that, just like in the tree, how moving each of the N-1 edges breaks the tree into two pieces and defines the cut. With a cycle, if you remove just one edge, you don't get a cut. The thing remains connected, but if you remove any pair of edges, then that induces a cut of the graph, corresponding to the two pieces that will remain. No matter which pair of edges you remove, you get a distinct pair of groups, distinct cuts. So ranging overall N choose 2 choices of pairs of edges, you generate N choose 2 different cuts. Each of these cuts has exactly two crossing edges, and it's easy to see that's the fewest possible. >>So that's the lower bound, which was simple enough. Let's now move on to the upper bound, which, a purely count-all fact will follow from an algorithm. So consider any graph that has N vertices, and let's think about the different minimum cuts of that graph. What we're going to use is that the analysis of the contraction algorithm proceeded in a fairly curious way. So remember how we define the success probability of a contraction algorithm. We fixed up front, some min cut (A, B). And we defined the contraction algorithm, the basic contraction algorithm, before the repeated trials. We defined the contraction algorithm as successful, if and only if it output the minimum cut (A, B) that we designated upfront. If it output some other minimum cut, we didn't count it. We said nope, that's a failure. So we actually analyzed a stronger property than what we were trying to solve, which is outputting a given min cut (A, B) rather than just any/all min cut. So how is that useful? Well, let's apply it here. For each of these T minimum cuts of this graph, we can think about the probability that the contraction algorithm outputs that particular min cut. So we're gonna instantiate the analysis with a particular minimum cut (Ai, Bi). And what we proved in the analysis is that the probability that the algorithm outputs the cut (Ai, Bi), not just any/all min cut. But, in fact, this exact cut (Ai, Bi) is bounded below by. We, in the end, we made a sloppy inequality. We said it's at least 1/(N^2). But if you go back to the analysis, you'll see that it was, in fact, 2/N(N-1), also known as 1/(N choose 2). So instantiating the contraction algorithm success probability analysis without all of the repeated trials business, we show that for each of these T cuts, for each fixed cut (Ai, Bi), the probability that this algorithm outputs that particular cut is at least 1/(N choose 2). >>Let's introduce a name for this event, the event that the contraction algorithm outputs the Ith min cut. Let's call this Si. The key observation is that the Sis are disjoint events. Remember an event is just a subset of stuff that could happen. So one thing that could happen is that the algorithm outputs the Ith main cuts, and by this joint, we just mean that there is no outcome that in a given pair of events. And that's because the contraction algorithm at N to the [inaudible], once it makes its conflicts, it outputs a single cut is a distinct cut. It can only output a dest one of them. >>Why is it important that these Sis are disjoint events? Well, with disjoint events, the probabilities <i>add</i> the probability of the union of a bunch of disjoint events is the sum of the probabilities of constituent events. If you want to think about this pictorially, and just draw a big box, denoting everything that could happen omega, and then these SIs just these [inaudible] that don't overlap. So S1, S2, S3, and so on. Now the sum or probabilities of this joint events can sum to, at most, 1, right? The probability of all of omega is 1, and these SIs have not overlap and are packed into omega, so the sum of their probabilities is gonna be smaller. >>We're adding up formally. We have that the sum of the probabilities. Which we can lower bound by the number of different events. And remember there are T different min cuts for some parameter T. For each min cut (Ai, Bi), a lower bound of the probability that, that could spit out as output is 1/(N choose 2). So a lower bound on the sum of all of these probabilities is the number of them, T times the probability lower bound, 1/(N choose 2), and this has got to be at most 1. Rearranging, what do we find? T, the number of different mid-cuts, is bounded above by N choose 2. Exactly the lower bound provided by the N cycle. The N cycle has N choose 2 distinct minimum cuts. No other graph has more. Every graph has only a polynomial number indeed, at most a quadratic number of minimum cuts.