The third step, So, there is a lot of Today we're going to not talk about sorting. This is an exciting new development. We know that what we're looking We're going to talk about another problem, a related problem, but a different problem. We're going to talk about another problem that we would like to solve in linear time. Last class we talked about we could do sorting in linear time. To do that we needed some additional assumptions. Today we're going to look at a problem that really only needs linear time, even though at first glance it might look like it requires sorting. So this is going to be an easier problem. The problem is I give you a bunch of numbers. Let's call them elements. And they are in some array, let's say. And they're in no particular order, so unsorted. I want to find the kth smallest element. This is called the element of rank k. In other words, I have this list of numbers which is unsorted. And, if I were to sort it, I would like to know what the kth element is. But I'm not allowed to sort it. One solution to this problem, this is the naÃ”ve algorithm, is you just sort and then return the kth element. This is another possible definition of the problem. And we would like to do better than that. So you could sort, what's called the array A, and then return A[k]. That is one thing we could do. And if we use heap sort or mergesort, this will take n lg n time. We would like to do better than n lg n. Ideally linear time. The problem is pretty natural, straightforward. It has various applications. Depending on how you choose k, k could be any number between 1 and n. For example, if we choose k=1 that element has a name. Any suggestions of what the name is? The minimum. That's easy. Any suggestions on how we could find the minimum element in an array in linear time? Right. Just scan through the array. Keep track of what the smallest number is that you've seen. The same thing with the maximum, k=n. These are rather trivial. But a more interesting version of the order statistic problem is to find the median. This is either k equals n plus 1 over 2 floor or ceiling. I will call both of those elements medians. Finding the median of an unsorted array in linear time is quite tricky. And that sort of is the main goal of this lecture, is to be able to find the medians. For free we're going to be able to find the arbitrary kth smallest element, but typically we're most interested in finding the median. And on Friday in recitation you'll see why that is so useful. There are all sorts of situations where you can use median for really effective divide-and-conquer without having to sort. You can solve a lot of problems in linear time as a result. And we're going to cover today two algorithms for finding order statistics. Both of them are linear time. The first one is randomized, so it's only linear expected time. And the second one is worst-case linear time, and it will build on the randomized version. Let's start with a randomize divide-and-conquer algorithm. This algorithm is called rand-select. And the parameters are a little bit more than what we're used to. The order statistics problem you're given an array A. And here I've changed notation and I'm looking for the ith smallest element, so i is the index I'm looking for. And I'm also going to change the problem a little bit. And instead of trying to find it in the whole array, I'm going to look in a particular interval of the array, A from p up to q. We're going to need that for a recursion. This better be a recursive algorithm because we're using divide-and-conquer. Here is the algorithm. With a base case. It's pretty simple. Then we're going to use part of the quicksort algorithm, randomized quicksort. We didn't actually define this subroutine two lectures ago, but you should know what it does, especially if you've read the textbook. This says in the array A[p...q] pick a random element, so pick a random index between p and q, swap it with the first element, then call partition. And partition uses that first element to split the rest of the array into less than or equal to that random partition and greater than or equal to that partition. This is just picking a random partition element between p and q, cutting the array in half, although the two sizes may not be equal. And it returns the index of that partition element, some number between p and q. And we're going to define k to be this particular value, r minus p plus 1. And the reason for that is that k is then the rank of the partition element. This is in A[p...q]. Let me draw a picture here. We have our array A. It starts at p and ends at q. There is other stuff, but for this recursive all we care about is p up to q. We pick a random partition element, say this one, and we partition things so that everything in here, let's call this r, is less than or equal to A[r] and everything up here is greater than or equal to A[r]. And A[r] is our partition element. After this call, that's what the array looks like. And we get r. We get the index of where partition element is stored. The number of elements that are less than or equal to A[r] and including r is r minus p plus 1. There will be r minus p elements here, and we're adding 1 to get this element. And, if you start counting at 1, if this is rank 1, rank 2, this element will have rank k. That's just from the construction in the partition. And now we get to recurse. And there are three cases -- -- depending on how i relates to k. Remember i is the rank that we're looking for, k is the rank that we happen to get out of this random partition. We don't have much control over k, but if we're lucky i=k. That's the element we want. Then we just return the partition element. More likely is that the element we're looking for is either to the left or to the right. And if it's to the left we're going to recurse in the left-hand portion of the array. And if it's to the right we're going to recurse in the right-hand portion. So, pretty straightforward at this point. I just have to get all the indices right. Either we're going to recurse on the part between p and r minus 1, that's this case. The rank we're looking for is to the left of the rank of element A[r]. Or, we're going to recurse on the right part between r plus 1 and q. Where we recurse on the left part the rank we're looking for remains the same, but when we recurse on the right part the rank we're looking for gets offset. Because we sort of got rid of the k elements over here. I should have written this length is k. We've sort of swept away k ranks of elements. And now within this array we're looking for the i minus kth smallest element. That's the recursion. We only recurse once. And random partition is not a recursion. That just takes linear time. And the total amount of work we're doing here should be linear time plus one recursion. And we'd next like to see what the total running time is in expectation, but let's first do a little example -- -- to make this algorithm perfectly clear. Let's suppose we're looking for the seventh smallest element in this array. And let's suppose, just for example, that the pivot we're using is just the first element. So, nothing fancy. I would have to flip a few coins in order to generate a random one, so let's just pick this one. If I partition at the element 6, this is actually an example we did two weeks ago, and I won't go through it again, but we get the same array, as we did two weeks ago, namely 2, 5, 3, 6, 8, 13, 10 and 11. If you run through the partitioning algorithm, that happens to be the order that it throws the elements into. And

If you run through the partitioning algorithm, that happens to be the order that it throws the elements into. And this is our position r. This is p here. It's just 1. And q is just the end. And I am looking for the seventh smallest element. And it happens when I run this partition that 6 falls into the fourth place. And we know that means, because all the elements here are less than 6 and all the elements here are greater than 6, if this array were sorted, 6 would be right here in position four. So, r here is 4. Yeah? The 12 turned into an 11? This was an 11, believe it or not. Let me be simple. Sorry. Sometimes my ones look like twos. Not a good feature. That's an easy way to cover. [LAUGHTER] Don't try that on exams. Oh, that one was just a two. No. Even though we're not sorting the array, we're only spending linear work here to partition by We know that if we had sorted the array 6 would fall here. We don't know about these other elements. They're not in sorted order, but from the properties of partition we know 6 went the right spot. We now know rank of 6 is 4. We happened to be looking for 7 and we happened to get this number 4. We want something over here. It turns out we're looking for 10, I guess. No, 11. There should be eight elements in this array, so it's the next to max. Max here is 13, I'm cheating here. The answer we're looking for is for is in the right-hand part because the rank we're looking for is 7, which is bigger than Now, what rank are we looking for in here? Well, we've gotten rid of four elements over here. It happened here that k is also 4 because p is 1 in this example. The rank of 6 was 4. We throw away those four elements. Now we're looking for rank 7 minus 4 which is 3. And, indeed, the rank 3 element here is still 11. So, you recursively find that. That's your answer. Now that algorithm should be pretty clear. The tricky part is to analyze it. And the analysis here is quite a bit like randomized quicksort, although not quite as hairy, so it will go faster. But it will be also sort of a nice review of the randomized quicksort analysis which was a bit tricky and always good to see a couple of times. We're going to follow the same kind of outline as before to look at the expected running time of this algorithm. And to start out we're going to, as before, look at some intuition just to feel good about ourselves. Also feel bad as you'll see. Let's think about two sort of extreme cases, a good case and the worst case. And I should mention that in all of the analyses today we assume the elements are distinct. It gets really messy if the elements are not distinct. And you may even have to change the algorithms a little bit because if all the elements are equal, if you pick a random element, the partition does not do so well. But let's assume they're all distinct, which is the really interesting case. A pretty luck case -- I mean the best cases we partition right in the middle. The number of elements to the left of our partition is equal to the number of elements to the right of our partition. But almost as good would be some kind of 1/10 to 9/10 split. Any constant fraction, we should feel that. Any constant fraction is as good as 1/2. Then the recurrence we get is, let's say at most, this bad. So, it depends. If we have let's say 1/10 on the left and 9/10 on the right every time we do a partition. It depends where our answer is. It could be if i is really small it's in the 1/10 part. If i is really big it's going to be in the 9/10 part, or most of the time it's going to be in the 9/10 part. We're doing worst-case analysis within the lucky case, so we're happy to have upper bounds. I will say t(n) is at most t of T(9/10n)+Theta(n). Clearly it's worse if we're in the bigger part. What is the solution to this recurrence? Oh, solving recurrence was so long ago. What method should we use for solving this recurrence? The master method. What case are we in? Three. Good. You still remember. This is Case 3. We're looking at nlog_b(a). b here is 10/9, although it doesn't really matter because a is 1. log base anything of 1 is 0. So, this is n^0 which is 1. And n is polynomially larger than 1. This is going to be O(n), which is good. That is what we want, linear time. If we're in the lucky case, great. Unfortunately this is only intuition. And we're not always going to get the lucky case. We could do the same kind of analysis as we did with randomized quicksort. If you alternate between lucky and unlucky, things will still be good, but let's just talk about the unlucky case to show how bad things can get. And this really would be a worst-case analysis. The unlucky case we get a split of 0:n-1. Because we're removing the partition element either way. And there could be nothing less than the partition element. We have 0 on the left-hand side and we have n-1 on the right-hand side. Now we get a recurrence like T(n)=T(n-1) plus linear cost. And what's the solution to that recurrence? n^2. Yes. This one you should just know. It's n^2 because it's an arithmetic series. And that's pretty bad. This is much, much worse than sorting and then picking the ith element. In the worst-case this algorithm really sucks, but most of the time it's going to do really well. And, unless you're really, really unlucky and every coin you flip gives the wrong answer, you won't get this case and you will get something more like the lucky case. At least that's what we'd like to prove. And we will prove that the expected running time here is linear. So, it's very rare to get anything quadratic. But later on we will see how to make the worst-case linear as well. This would really, really solve the problem. Let's get into the analysis. Now, you've seen an analysis much like this before. What do you suggest we do in order to analyze this expected time? It's a divide-and-conquer algorithm, so we kind of like to write down the recurrence on something resembling the running time. I don't need the answer, but what's the first step that we might do to analyze the expected running time of this algorithm? Sorry? Look at different cases, yeah. Exactly. We have all these possible ways that random partition could split. It could split 0 to the n-1. It could split in half. There are n choices where it could split. How can we break into those cases? Indicator random variables. Cool. Exactly. That's what we want to do. Indicator random variable suggests that what we're dealing with is not exactly just a function T(n) but it's a random variable. This is one subtlety. T(n) depends on the random choices, so it's really a random variable. And then we're going to use indicator random variables to get a recurrence on T(n). So, T(n) is the running time of rand-select on an input of size n. And I am also going to write down explicitly an assumption about the random numbers. That they should be chosen independently from each other. Every time I call random partition, it's generating a completely independent random number from all the other times I call random partition. That is important, of course, for this analysis to work. We will see why some point down the line. And now, to sort of write down an equation for T(n) we're going to define indicator random variables, as you suggested. And we will call it X_k. And this is for all k=0...n-1. Indicator random variables either 1 or 0. And it's going to be 1 if the partition comes out k on the left-hand side. So say the partition generates a k:n-k-1 split and it is 0 otherwise. We have n of these indicator random variables between 0...n-1. And in each case, no matter how the random choice comes out, exactly one of them will be 1. All the others will be 0. Now we can divide out the running time of this algorithm based on which case we're in. That will sort of unify this intuition that we did and get all the cases. And then we

based on which case we're in. What kind of clarify this intuition that we did and get all the cases. And then we can look at the expectation. T(n), if we just split out by cases, we have an upper bound like this. If we have 0 to n-1 split, the worst is we have n-1. Then we have to recurse in a problem of size n-1. In fact, it would be pretty hard to recurse in a problem of size 0. If we have a 1 to n-2 split then we take the max of the two sides. That's certainly going to give us an upper bound and so on. And at the bottom you get an n-1 to 0 split. This is now sort of conditioning on various events, but we have indicator random variables to tell us when these events happen. We can just multiply each of these values by the indicator random variable and it will come out 0 if that's not the case and will come out 1 and give us this value if that happens to be the split. So, if we add up all of those we'll get the same thing. This is equal to the sum over all k of the indicator random variable times the cost in that case, which is t of max k, and the other side, which is n-k-1, plus theta n. This is our recurrence, in some sense, for the random variable representing running time. Now, the value will depend on which case we come into. We know the probability of each of these events happening is the same because we're choosing the partition element uniformly at random, but we cannot really simplify much beyond this until we take expectations. We know this random variable could be as big as n^2. Hopefully it's usually linear. We will take expectations of both sides and get what we want. Let's look at the expectation of this random variable, which is just the expectation, I will copy over, summation we have here so I can work on this board. I want to compute the expectation of this summation. What property of expectation should I use? Linearity, good. We can bring the summation outside. Now I have a sum of expectation. Let's look at each expectation individually. It's a product of two random variables, if you will. This is an indicator random variable and this is some more complicated function, some more complicated random variable representing some running time, which depends on what random choices are made in that recursive call. Now what should I do? I have the expectation of the product of two random variables. Independence, exactly. If I know that these two random variables are independent then I know that the expectation of the product is the product of the expectations. Now we have to check are they independent? I hope so because otherwise there isn't much else I can do. Why are they independent? Sorry? Because we stated that they are, right. Because of this assumption. We assume that all the random numbers are chosen independently. We need to sort of interpolate that here. These X_k's, all the X_k's, X_0 up to X_n-1, so all the ones appearing in this summation are dependent upon a single random choice of this particular call to random partition. All of these are correlated, because if one of them is 1, all the others are forced to be correlation among the X_k's. But with respect to everything that is in here, and the only random part is this T(max(kn-k-1)). That is the reason that this random variable is independent from these. The same thing as quicksort, but I know some people got confused about it a couple lectures ago so I am reiterating. We get the product of expectations, E[X_k] E[T(max(kn-k-1))]. I mean the order n comes outside, but let's leave it inside for now. There is no expectation to compute there for order n. Order n is order n. What is the expectation of X_k? 1/n, because they're all chosen with equal probability. There is n of them, so the expectation is 1/n. The value is either 1 or 0. We start to be able to split this up. We have 1/n times this expected value of some recursive T call, and then we have plus 1 over n times order n, also known as a constant, but everything is summed up n times so let's expand this. I have the sum k=0 to n-1. I guess the 1/n can come outside. And we have expectation of [T(max(kn-k-1))]. Lots of nifty braces there. And then plus we have, on the other hand, the sum k=0 to n-1. Let me just write that out again. We have a 1/n in front and we have a Theta(n) inside. This summation is n^2. And then we're dividing by n, so this whole thing is, again, order n. Nothing fancy happened there. This is really just saying the expectation of order n is order n. Average value of order n is order n. What is interesting is this part. Now, what could we do with this summation? Here we start to differ from randomized quicksort because we have this max. Randomized quicksort we had the sum of T(k) plus T(n-k-1) because we were making both recursive calls. Here we're only making the biggest one. That max is really a pain for evaluating this recurrence. How could I get rid of the max? That's one way to think of it. Yeah? Exactly. I could only sum up to halfway and then double. In other words, terms are getting repeated twice here. When k=0 or when k=n-1, I get the same T(n-1). When k=1 or n-2, I get the same thing, 2 and n-3. What I will actually do is sum from halfway up. That's a little bit cleaner. And let me get the indices right. Floor of n/2 up to n-1 will be safe. And then I just have E[T(k)], except I forgot to multiply by 2, so I'm going to change this 1 to a 2. And order n is preserved. This is just because each term is appearing twice. I can factor it out. And if n is odd, I'm actually double-counting somewhat, but it's certain at most that. So, that's a safe upper bound. And upper bounds are all we care about because we're hoping to get linear. And the running time of this algorithm is definitely at least linear, so we just need an upper bounded linear. So, this is a recurrence. E[T(n)] is at most 2/n times the sum of half the numbers between 0 and n of E[T(k)]+Theta(n). It's a bit of hairy recurrence. We want to solve it, though. And it's actually a little bit easier than the randomized quicksort recurrence. We're going to solve it. What method should we use? Sorry? Master method? Master would be nice, except that each of the recursive calls is with a different value of k. The master method only works when all the calls are with the same value, same size. Alas, it would be nice if we could use the master method. What else do we have? Substitution. When it's hard, when in doubt, use substitution. I mean the good thing here is we know what we want. From the intuition at least, which is now erased, we really feel that this should be linear time. So, we know what we want to prove. And indeed we can prove it just directly with substitution. I want to claim there is some constant c greater than zero such that E[T(n)], according to this recurrence, is at most c times n. Let's prove that over here. As we guessed, the proof is by substitution. What that means is we're going to assume, by induction, that this inequality is true for all smaller m. I will just say 4 less than n. And we need to prove it for n. We get E[T(n)]. Now we are just going to expand using the recurrence that we have. It's at most this. I will copy that over. And then each of these recursive calls is with some value k that is strictly smaller than n. Sorry, I copied it wrong, floor of n over 2, not zero. And so I can apply the induction hypothesis to each of these. This is at most c times k by the induction hypothesis. And so I get this inequality. This c can come outside the summation because it's just a constant. And I will be slightly tedious in writing this down again, because what I care about is the summation here that is left over. This is a good old-fashioned summation. And if you remember back to your summation tricks or whatever, you should be able to evaluate this. If we started

And if you remember back to your summation tricks or whatever, you should be able to evaluate this. It started at zero and went up to n minus 1, that's just an arithmetic series, but here we have the tail end of an arithmetic series. And you should know, at least up to theta, what this is, right? n^2, yeah. It's definitely T(n^2). But we need here a slightly better upper bond, as we will see the constants really matter. What we're going to use is that this summation is at most 3/8 times n^2. And that will be critical, the fact that 3/8 is smaller than 1/2, I believe. So it's going to get rid of this 2. I am not going to prove this. This is an exercise. When you know that it is true, it's easy because you can just prove it by induction. Figuring out that number is a little bit more work, but not too much more. So you should prove that by induction. Now let me simplify. This is a bit messy, but what I want is c times n. Let's write it as our desired value minus the residual. And here we have some crazy fractions. This is 2 times 3 which is 6 over 8 which is 3/4, right? Here we have 1, so we have to subtract up 1/4 to get 3/4. And this should be, I guess, 1/4 times c times n. And then we have this theta n with double negation becomes a plus theta n. That should be clear. I am just rewriting that. So we have what we want over here. And then we hope that this is nonnegative because what we want is that this less than or equal to c times n. That will be true, provided this thing is nonnegative. And it looks pretty good because we're free to choose c however large we want. Whatever constant is imbedded in this beta notation is one fixed constant, whatever makes this recurrence true. We just set c to be bigger than 4 times that constant and then this will be nonnegative. So this is true for c sufficiently large to dwarf that theta constant. It's also the base case. I just have to make the cursory mention that we choose c large enough so that this claim is true, even in the base case where n is at most some constant. Here it's like 1 or so because then we're not making a recursive call. What we get -- This algorithm, randomize select, has expected running time order n, Theta(n). The annoying this is that in the worst-case, if you're really, really unlucky it's n^2. Any questions before we move on from this point? This finished off the proof of this fact that we have Theta(n) expected time. We already saw the n^2 worst-case. All perfectly clear? Good. You should go over these proofs. They're intrinsically related between randomized quicksort and randomized select. Know them in your heart. This is a great algorithm that works really well in practice because most of the time you're going to split, say, in the middle, somewhere between a 1/4 and 3/4 and everything is good. It's extremely unlikely that you get the n^2 worst-case. It would have to happen with like 1 over n^n probability or something really, really small. But I am a theoretician at least. And it would be really nice if you could get Theta(n) in the worst-case. That would be the cleanest result that you could hope for because that's optimal. You cannot do better than Theta(n). You've got to look at the elements. So, you might ask, can we get rid of this worst-case behavior and somehow avoid randomization and guarantee Theta(n) worst-case running time? And you can but it's a rather nontrivial algorithm. And this is going to be one of the most sophisticated that we've seen so far. It won't continue to be the most sophisticated algorithm we will see, but here it is. Worst-case linear time order statistics. And this is an algorithm by several, all very famous people, Blum, Floyd, Pratt, Rivest and Tarjan. I think I've only met the B and the R and the T. Oh, no, I've met Pratt as well. I'm getting close to all the authors. This is a somewhat old result, but at the time it was a major breakthrough and still is an amazing algorithm. Ron Rivest is a professor here. You should know him from the R in RSA. When I took my PhD comprehensives some time ago, on the cover sheet was a joke question. It asked of the authors of the worst-case linear time order statistics algorithm, which of them is the most rich? Sadly it was not a graded part of the comprehensive exam, but it was an amusing question. I won't answer it here because we're on tape, [LAUGHTER] but think about it. I may not be obvious. Several of them are rich. It's just the question of who is the most rich. Anyway, before they were rich they came up with this algorithm. They've come up with many algorithms since, even after getting rich, believe it or not. What we want is a good pivot, guaranteed good pivot. Random pivot is going to be really good. And so the simplest algorithm is just pick a random pivot. It's going to be good with high probability. We want to force a good pivot deterministically. And the new idea here is we're going to generate it recursively. What else could we do but recurse? Well, you should know from your recurrences that if we did two recursive calls on problems of half the size and we have a linear extra work that's the mergesort recurrence, T(n)=2[T(n/2)+Theta(n)]. You should recite in your sleep. That's n lg n. So we cannot recurse on two problems of half the size. We've got to do better. Somehow these recursions have to add up to strictly less than n. That's the magic of this algorithm. So this will just be called select instead of rand-select. And it really depends on an array, but I will focus on the i-th element that we want to select and the size of the array that we want to select in. And I am going to write this algorithm slightly less formally than randomize-select because it's a bit higher level of an algorithm. And let me draw over here the picture of the algorithm. The first step is sort of the weirdest and it's one of the key ideas. You take your elements, and they are in no particular order, so instead of drawing them on a line, I am going to draw them in a 5 by n over 5 grid. Why not? This, unfortunately, take a little while to draw, but it will take you equally long so I will take my time. It doesn't really matter what the width is, but it should be width n over 5 so make sure you draw your figure accordingly. Width n over 5, but the height should be exactly 5. I think I got it right. I can count that high. Here is 5. And this should be, well, you know, our number may not be divisible by 5, so maybe it ends off in sort of an odd way. But what I would like is that these chunks should be floor of n over 5. And then we will have, at most, four elements left over. So I am going to ignore those. They don't really matter. It's just an additive constant. Here is my array. I just happened to write it in this funny way. And I will call these vertical things groups. I would circle them, and I did that in my notes, but things get really messy if you start circling. This diagram is going to get really full, just to warn you. By the end it will be almost unintelligible, but there it is. If you are really feeling bored, you can draw this a few times. And you should draw how it grows. So there are the groups, vertical groups of five. Next step. The second step is to recurse. This is where things are a bit unusual, well, even more unusual. Oops, sorry. I really should have had a line between one and two so I am going to have to move this down and insert it here. I also, in step one, want to find the median of each group. What I would like to do is just imagine this figure, each of the five elements in each group gets reorganized so that the middle one is the median. So I am going to call these the medians of each group. I have five elements so the median is right in the middle. There are two elements less than the median, two elements greater than the median. Again, we're assuming all elements are distinct. So there they are. I compute

them. How long does that take me? N over five groups, each with five elements, compute the median of each one? Sorry? Yeah, 2 times n over 5. It's theta n, that's all I need to know. I mean, you're counting comparisons, which is good. It's definitely Theta(n). The point is within each group, I only have to do a constant number of comparisons because it's a constant number of elements. It doesn't matter. You could use randomize select for all I care. No matter what you do, it can only take a constant number of comparisons. As long as you don't make a comparison more than once. So this is easy. You could sort the five numbers and then look at the third one, it doesn't matter because there are only five of them. That's one nifty idea. Already we have some elements that are sort of vaguely in the middle but just of the group. And we've only done linear work. So doing well so far. Now we get to the second step, which I started to write before, where we recurse. So the next idea is, well, we have these floor over n over 5 medians. I am going to compute the median of those medians. I am imagining that I rearranged these. And, unfortunately, it's an even number, there are six of them, but I will rearrange so that this guy, I have drawn in a second box, is the median of these elements so that these two elements are strictly less than this guy, these three elements are strictly greater than this guy. Now, that doesn't directly tell me anything, it would seem, about any of the elements out here. We will come back to that. In fact, it does tell us about some of the elements. But right now this element is just the median of these guys. Each of these guys is a median of five elements. That's all we know. If we do that recursively, this is going to take T of n over 5 time. So far so good. We can afford a recursion on a problem of size n over 5 and linear work. We know that works out to linear time. But there is more. We're obviously not done yet. The next step is x is our partition element. We partition there. The rest of the algorithm is just like randomized partition, so we're going to define k to be the rank of x. And this can be done, I mean it's n minus r plus 1 or whatever, but I'm not going to write out how to do that because we're at a higher level here. But it can be done. And then we have the three-way branching. So if i happens to equal k we're happy. The pivot element is the element we're looking for, but more likely i is either less than k or it is bigger than k. And then we make the appropriate recursive call, so here we recursively select the i-th smallest element -- -- in the lower part of the array. Left of the partition element. Otherwise, we recursively select the i minus k-th smallest element in the upper part of the array. I am writing this at a high level because we've already seen it. All of this is the same as the last couple steps of randomized select. That is the algorithm. The real question is why does it work? Why is this linear time? The first question is what's the recurrence? We cannot quite write it down yet because we don't know how big these recursive subproblems could be. We're going to either recurse in the lower part or the upper part, that's just like before. If we're unlucky and we have a split of like zero to n minus one, this is going to be a quadratic time algorithm. The claim is that this partition element is guaranteed to be pretty good and good enough. The running time of this thing will be T of something times n, and we don't know what the something is yet. How big could it be? Well, I could ask you. But we're sort of indirect here so I will tell you. We have already a recursive call of T of n over 5. It better be that whatever constant, so it's going to be something times n, it better be that that constant is strictly less than 4/5. If it's equal to 4/5 then you're not splitting up the problem enough to get an n lg n running time. If it's strictly less than 4/5 then you're reducing the problem by at least a constant factor. In the sense if you add up all the recursive subproblems, n over 5 and something times n, you get something that is a constant strictly less than one times n. That forces the work to be geometric. If it's geometric you're going to get linear time. So this is intuition but it's the right intuition. Whenever you're aiming for linear time keep that in mind. If you're doing a divide-and-conquer, you've got to get the total subproblem size to be some constant less than one times n. That will work. OK, so we've got to work out this constant here. And we're going to use this figure, which so far looks surprisingly uncluttered. Now we will make it cluttered. What I would like to do is draw an arrow between two vertices, two points, elements, whatever you want to call them. Let's call them a and b. And I want to orient the arrow so it points to a larger value, so this means that a is less than b. This is notation just for the diagram. And so this element, I am going to write down what I know. This element is the median of these five elements. I will suppose that it is drawn so that these elements are larger than the median, these elements are smaller than the median. Therefore, I have arrows like this. Here is where I wish I had some colored chalk. This is just stating this guy is in the middle of those five elements. I know that in every single column. Here is where the diagram starts to get messy. I am not done yet. Now, we also know that this element is the median of the medians. Of all the squared elements, this guy is the middle. And I will draw it so that these are the ones smaller than the median, these are the ones larger than the median. I mean the algorithm cannot do this. It doesn't necessarily know how all this works. I guess it could, but this is just for analysis purposes. We know this guy is bigger than that one and bigger than that one. We don't directly know about the other elements. We just know that that one is bigger than both of those and this guy is smaller than these. Now, that is as messy as the figure will get. Now, the nice thing about less than is that it's a transitive relation. If I have a directed path in this graph, I know that this element is strictly less than that element because this is less than that one and this is less than that one. Even though directly I only know within a column and within this middle row, I actually know that this element -- This is x, by the way. This element is larger than all of these elements because it's larger than this one and this one and each of these is larger than all of those by these arrows. I also know that all of these elements in this rectangle here, and you don't have to do this but I will make the background even more cluttered. All of these elements in this rectangle are greater than or equal to this one and all of the elements in this rectangle are less than or equal to x. Now, how many are there? Well, this is roughly halfway along the set of groups and this is 3/5 of these columns. So what we get is that there are at least -- We have n over 5 groups and we have half of the groups that we're looking at here roughly, so let's call that floor of n over 2, and then within each group we have three elements. So we have at least 3 times floor of floor of n over 5 over 2 n floor elements that are less than or equal to x. And we have the same that are greater than or equal to x. Let me simplify this a little bit more. I can also give you some more justification, and we drew the picture, but just for why this is true. We have at least n over 5 over 2 group medians that are less than or equal to x. This is the argument we use. We have half of the group medians are less than or equal to x because x is the median of the group median, so that is no big surprise.

group medians are less than or equal to x because x is the median of the group medians, so that is no big surprise. This is almost an equality but we're making floors so it's greater than or equal to. And then, for each group median, we know that there are three elements there that are less than or equal to that group median. So, by transitivity, they're also less than or equal to x. We get this number times three. This is actually just floor of n over 10. I was being unnecessarily complicated there, but that is where it came from. What we know is that this thing is now at least 3 times n over 10, which is roughly 3/10 of elements are in one side. In fact, at least 3/10 of the elements are in each side. Therefore, each side has at most 7/10 elements roughly. So the number here will be 7/10. And, if I'm lucky, 7/10 plus 1/5 is strictly less than one. I believe it is, but I have trouble working with tenths. I can only handle powers of two. What we're going to use is a minor simplification, which just barely still works, is a little bit easier to think about. It's mainly to get rid of this floor because the floor is annoying. And we don't really have a sloppiness lemma that applies here. It turns out if n is sufficiently large, 3 times floor of n over 10 is greater than or equal to 1/4. Quarters I can handle. The claim is that each group has size at least 1/4, therefore each group has size at most 3/4 because there's a quarter on the side. This will be 3/4. And I can definitely tell that 1/5 is less than 1/4. This is going to add up to something strictly less than one and then it will work. How is my time? Good. At this point, the rest of the analysis is easy. How the heck you would come up with this algorithm, you realize that this is clearly a really good choice for finding a partition element, just barely good enough that both recursions add up to linear time. Well, that's why it took so many famous people. Especially in quizzes, but I think in general this class, you won't have to come up with an algorithm this clever because you can just use this algorithm to find the median. And the median is a really good partition element. Now that you know this algorithm, now that we're beyond 1973, you don't need to know how to do this. I mean you should know how this algorithm works, but you don't need to do this in another algorithm because you can just say run this algorithm, you will get the median in linear time, and then you can partition to the left and the right. And then the left and the right will have exactly equal size. Great. This is a really powerful subroutine. You could use this all over the place, and you will on Friday. Have I analyzed the running time pretty much? The first step is linear. The second step is T of n over I didn't write it, is linear. And then the last step is just a recursive call. And now we know that this is 3/4. I get this recurrence. T of n is, I'll say at most, T of n over 5 plus T of 3/4n. You could have also used 7/10. It would give the same answer, but you would also need a floor so we won't do that. I claim that this is linear. How should I prove it? Substitution. Claim that T of n is at most again c times n, that will be enough. Proof is by substitution. Again, we assume this is true for smaller n. And want to prove it for n. We have T of n is at most this thing. T of n over 5. And by induction, because n of 5 is smaller than n, we know that this is at most c. Let me write it as c over 5 times n. Sure, why not. Then we have here 3/4cn. And then we have a linear term. Now, unfortunately, I have to deal with things that are not powers of two. I will cheat and look at my notes. This is also known as 19/20 times c times n plus theta n. And the point is just that this is strictly less than one. Because it's strictly less than one, I can write this as one times c of n minus some constant, here it happens to be 1/20, as long as I have something left over here, 1/20 times c times n. Then I have this annoying theta n term which I want to get rid of because I want this to be nonnegative. But it is nonnegative, as long as I set c to be really, really large, at least 20 times whatever constant is here. So this is at most c times n for c sufficiently large. And, oh, by the way, if n is less than or equal to 50, which we used up here, then T of n is a constant, it doesn't really matter what you do, and T of n is at most c times n for c sufficiently large. That proves this claim. Of course, the constant here is pretty damn big. It depends exactly what the constants and the running times are, which depends on your machine, but practically this algorithm is not so hot because the constants are pretty big. Even though this element is guaranteed to be somewhere vaguely in the middle, and even though these recursions add up to strictly less than n and it's geometric, it's geometric because the problem is reducing by at least a factor of 19/20 each time. So it actually takes a while for the problem to get really small. Practically you probably don't want to use this algorithm unless you cannot somehow flip coins. The randomized algorithm works really, really fast. Theoretically this is your dream, the best you could hope for because it's linear time and you need linear time as guaranteed linear time. I will mention, before we end, an exercise. Why did we use groups of five? Why not groups of three? As you might guess, the answer is because it doesn't work with groups of three. But it's quite constructive to find out why. If you work through this math with groups of three instead of groups of five, you will find that you don't quite get the problem reduction that you need. Five is the smallest number for which this works. It would work with seven, but theoretically not any better than a constant factor. Any questions? All right. Then recitation Friday. Homework lab Sunday. Problem set due Monday. Quiz one in two weeks.