6.046J Introduction to Algorithms, Fall 2005
Transcript – Lecture 4

OK. Today we are going to talk about a very interesting algorithm called Quicksort -- -- which was invented by Tony Hoare in 1962. And it has ended up being a really interesting algorithm from many points of view. And because of that, it turns out today's lecture is going to be both hard and fast. If you see the person next to you sleeping, you will want to say let's get going. It's a divide-and-conquer algorithm.

And it sorts, as they say, in place, meaning that it just rearranged the elements where they are. That is like insertion sort rearranges elements where they are. Mergesort does not. Mergesort requires extra storage in order to do the merge operation. To merge in linear time and place, it doesn't merge in place in linear time. It doesn't do it just by rearranging. It is nice because it is in place, so that means that it is fairly efficient in its use of storage. And it also happens to be very practical if you tune it a bit. The basic algorithm turns out, if you just implement that, it's not necessarily that efficient.

But if what you do was then do the standard kinds of things you do to goose up the runtime of something, and we'll talk a little about what those things are, then it can be very, very practical. So, it uses divide-and-conquer paradigm. First step is divide. And to do this basically it does it by partitioning. So, it partitions the input array into two subarrays around an element we call the pivot --

-- such that elements in the lower subarray are less than or equal to x, are less than or equal to elements in the upper subarray. If we draw a picture of the input array, this partition step basically takes some element x and everything over here is less than or equal to x after the partition step and everything over here is greater than or equal to x. And so now the conquer step is pretty easy.

You just recursively sort the two subarrays. So, I recursively sort the elements less than or equal to x, I recursively sort the elements greater than or equal to x. And then combine is then just trivial. Because once I have sorted the things less than or equal to x, then sorted the things greater than or equal to x, the whole thing is sorted. So, there is nothing to do really for the combine. The key step in quicksort is this partition step. That is the thing that does all of the work. And so you can view quicksort of just as recursive partitioning. That's all it is.

Just as mergesort was recursive merging, quicksort sort of goes the other way around and does recursive partitioning. The key is the linear time, by which I mean theta n, partitioning subroutine. And here are some pseudocode for it. This is actually slightly different from the book. The book has one. In fact, there is a nice problem in the book that has even a different one, but they are all basically the same idea.

Partition (A, p, q). And what we are looking at, at this step of the recursion, is the subarray A from p to q. And basically we pick a pivot, which is we are going to just pick as the first element of the array A of p. And the book, just for your information,

uses A of q. I use A of p. It doesn't really matter. And then we set an index to p and then we have a loop. This is the code. Basically the structure of it is a for loop with an "if" statement in the middle. And so the structure of the algorithm of this partitioning step looks as follows.

We set the pivot to be the first element. Here is p and here is q. This is going to be our invariant for the loop. And, at any time during the execution of a loop, I essentially have some values up to i which are already less than or equal to x and then some values that end at j minus 1 that are greater than or equal to x. And then I don't know about the rest. And so we start out with i equal to p and j equal to p plus 1. It starts out at p plus 1 so that everything is unknown except for x here. And then the idea is that it is going to preserve this invariant.

And the way it does it is, as we go through the loop, it looks at a of j and says is it greater than or equal to x? Sorry, is it less than or equal to x? If it is greater than or equal to x it does nothing, because what can happen? If this is greater than or equal to x, essentially it just goes to the next iterational loop which moves this boundary and the invariant is satisfied. Does everybody see that? Yeah, OK.

But if it is less than or equal to x, I have got a problem if I want to maintain the invariant if this next element is less than or equal to x. And so what it does then is it says oh, let me just move this boundary and swap this element here, which is greater than or equal to x, with this one here that is less than or equal to x, thereby increasing the size of this subarray and then the invariant is satisfied again. It is a fairly simple algorithm.

And it is actually a very tight and easy algorithm. That is one reason that this is such a great piece of code because it is very efficient. Now, in principle, the running time for this on n elements is order n. Because I am basically just going through the n elements and just doing a constant amount of work and then just a constant amount of work outside. This is a clever piece of code. In fact, in principle partition is easy, right? If I weren't worrying about doing it in place, it is really a pretty easy thing to do. I take an element and just compare every other element with it. I throw one into one bin and one into the other. That is clearly linear time.

But often what you find is that just because you can do it that way theoretically doesn't mean that that is going to end up giving you good code. And this is a nice piece of code that allows you to do it in place. And that is one reason why this is a particularly good algorithm, because the constants are good. So, yes, when we do asymptotic analysis we tend to ignore the constants, but when you're actually building code you care about the constants.

But first you care much more than just about the constants, is whether overall it is going to be a fast algorithm. Let's go through an example of this, I guess I will do it over here, just so we get the gist. Here is a sample array that I have created out of hallcloth. And here we are going to set x, the pivot, to be 6. Let's look to see how this algorithm works. So, i starts out here and j starts out here if we initialize. And what we do is start scanning right, essentially that code is scanning right until it gets something which is less than or equal to the pivot. It keeps going here until it finds, j keeps incrementing until it finds something that is less than or equal to the pivot.

And, in that case, it is the number 5. Then it says we will swap these two things. And it does that and we get 6, 5, 13, 10, 8, 3, 2, 11. And meanwhile now i gets

incremented and j continues where it left off. And so now we keep scanning right until we get to something that is less than or equal to the pivot. In this case it is 3. We swap 3 and 5 and get 6, 3, etc. And now, at this step we increment i, we start j out here.

And in this case, right off the bat, we have something which is less than or equal to x, so we swap these two. I blew it, didn't I? Oops. What did I do? I swapped the wrong thing, didn't I, here? Ah-ha. That is why I am not a computer. Good. We should have swapped this guy, right? Swapped i plus 1, right? This was i. We swap i plus 1, good. So, that's all wrong. Let's swap the right things. Now we have 6, 5, 3, 10, 8, 13, 2, 11. That even corresponds to my notes for some strange reason. This is i and now this is j. And now when I look, I immediately have something that is less than or equal to the pivot.

We swap this and i plus 1, so now we have 6, 5, 3, 2, 8, 13, 10, 11. And we, at that point, increment i to here. And we have j now going here and j runs to the end. And the loop terminates. When the loop terminates there is one less swap that we do, which is to put our pivot element in the middle between the two subarrays. Here we swap this one and this one, and so that gives us then 2, 5, 3, 6, 8, 13, 10, 11. And this is the pivot.

And everything over here is less than or equal to the pivot. And everything over here is greater than or equal to the pivot. OK, so the quicksort routine. Once we have this partition routine, quicksort is a pretty easy piece of code to write. I should have said return here i, right? You have got to return with the pivot. Here I have got to return i because we want to know where the pivot element is.

Sorry. I will plug in my code. r gets partition of (A, p, q) and then we quicksort (A, p, r-1) and quicksort of (A, r+1, q). And that is it. That's the code. The initial call is quicksort of (A, 1, n). Because once we partitioned, we just have to quicksort the two portions, the left and right portions. Just the boundary case is probably worth mentioning for a second. If there are zero or one elements, that is basically what can possibly happen here, is that I get zero or one elements here. Then the point is there is nothing to do because the array is sorted, either because it is an empty array or because it only has one element.

One of the tricks to making quicksort go fast, as one tunes this, is to, in fact, look at having a special purpose sorting routine for small numbers of elements. For example, if you get down to five elements having some straight line piece of code that knows how to sort five elements sufficiently as opposed to continuing to go through recursion in order to accomplish that. And there are a variety of other things. This is a tail recursive code, and so you can use certain tail recursion optimizations.

And there are a variety of other kinds of optimizations that you can use to make this code go fast. So, yeah, you can tune it up a bit beyond what is there, but the core of it is this efficient partitioning routine. That is the algorithm. It turns out that looking at how fast it runs is actually a little bit challenging. In the analysis, we are going to assume that all elements are distinct. It turns out that this particular code does not work very well when you have repeated elements, but Hoare's original partitioning routine is actually more efficient in that case if there are duplicates in what you are sorting.

And I encourage you to look at that. It has a much more complicated invariant for partitioning routine, but it does a similar kind of thing. It's just a bit more complicated. If they weren't all distinct, there are things you can do to make them distinct or you can just use this code. The easiest thing to do is just use Hoare's original code because that works pretty well when they are nondistinct.

But this is a little bit easier to understand. Let's let T(n) be the worst-case running time on n elements. And so what is the worse-case? What is the worse-case going to be for quicksort? That's right. If you always pick the pivot and everything is greater than or everything is less than, you are not going to partition the array very well. And when does that happen? What does the original input look like that makes that happen? If it is already sorted or reverse sorted.

So, if the input is sorted or reverse sorted. That is actually kind of important to understand, because it turns out the most common thing to sort is something that is already sorted, surprisingly, or things that are nearly sorted. But often it is just sorted and somebody wants to make sure it is sorted. Well, let's just sort it again rather than checking to see if it is sorted. And, in those cases, one side of the partition of each partition has no elements. Then we can write out what the recursion is for that. We have T(n). If one side has no elements, we are going to have T(0) on that side.

And on the other side we are going to have T(n-1). We are just writing out the recursion for this. One side has no elements. The other side has n-1 elements. And then partitioning and all the bookkeeping and so forth is order n. What is T(0)? What is T(0)? What is that asymptotically? It's a constant, order 1. That is just order 1 + T(n-1) + order n. Well, the order 1 can be absorbed into the order n, so this is really just saying it is T(n-1) + order n.

And what is that equal to? That is order n^2. Why is that order n^2? It is an arithmetic series. Actually, just like we got for insertion sort. Just like for insertion sort it is an arithmetic series. Going through all that work and we have an algorithm called quicksort, and it is no faster than insertion sort. Nevertheless, I said it was a good algorithm. The reason it is a good algorithm is because its average case time, as we are going to see, is very good.

But let's try to understand this a little bit more just so that we understand the difference between what is going to happen in the average case and what is going to happen in the worse-case. Let's draw a recursion tree for this for T(n) = T(0) + T(n-1) + and I will make the constant explicit for cn. So, we get an intuition of what is going on. Some constant times n. And then we have T(n) is equal to, and we write it with the constant part here, cn, and then T(0) here, and then T(n-1) here. Now, I know that all you folks are really fast and want to jump immediately to the full-blown tree.

But, let me tell you, my advice is that you spend just a couple of minutes writing it out. Since the tree grows exponentially, it only costs you a constant overhead to write out the small cases and make sure that you have got the pattern that you are developing. So, I am going to go one more step. Here we have T(0) and now this becomes c(n-1) and now we have another T(0) over here and T(n-2). And we continue that, dot, dot, dot. That is all equal to cn with a T(0) here, c(n-1) with a T(0), c(n-2), T(0) here, and that goes all the way down until we end up with T(1) down here.

What is the height of this tree? What is the height of the tree here? Yeah, n. Good. Because every step we are just decrementing the argument by 1. So, the height is n. To analyze this, let's first add up everything that is here. Just so we understand where these things are coming from, this is just theta of the summation of k equals 1 to n of k, actually of ck. That is what is in there. And that is equal to order n^2. That is where our algorithmatic series is coming from. So, that is Theta(n^2). And then all of these things here are all Theta(1).

And how many of them are there? There are n Theta(1)'s. So, the total amount is T(n) = Theta(n) + Theta(n^2) = Theta(n^2). Just to see what the structure is in terms of the recursion tree, it is a highly unbalanced recursion tree. Now I am going to do something that I told you should never do, which is we are going to be do a best-case analysis. This is for intuition only. And, in general, we don't do best-case analyses.

It doesn't mean anything, unless we get some intuition for it maybe. But basically it means nothing mathematically because it's providing no guarantee. And so this is intuition only. If we are really lucky what happens for partition? What is going to be the lucky case? Yeah, it splits right in the middle. Which is essentially -- -- n/2 : n/2. It is really (n-1)/2 : (n-1)/2, but we're not going to worry about the details because we're only doing intuition for the best-case because best-case is not what we want.

If that happened, what is the recurrence I get? Imagine it split it exactly in the middle every time, then what happens? You get T(n) = 2T(n/2) + order n for partitioning and bookkeeping. And what is the solution of that recurrence? That is n log n. That is the same as the merge sort recurrence. It is which case of the master theorem? Case 2, right? Because n to the log base 2 of 2 is n to the 1, it is the same, so we tack on the extra log n.

Case 2 of the master theorem. That is pretty good. That says that in the best-case quicksort is going to do well. How about let's suppose the split is always let's say 1/10 : 9/10, 1/10n : 9/10n. In that case, are we lucky or are we unlucky? I mean, if the split is really skewed, we clearly are going to be unlucky, right, because then it's, say, 1 to n. If it is really in the middle it is n log n. What do you suppose it is if it is 1/10 : 9/10? Is that lucky or unlucky? We will have a little democracy here. Who thinks that that is a lucky case? It is going to be fast running time. And who thinks it is an unlucky case? OK, so we have some brave souls. And who didn't vote? Oh, come on.

Come on. It is always better, by the way, to say yes or no and be right or wrong, because then you have some emotional commitment to it and we will remember better, rather than just sitting and being quiet. You don't manipulate your own emotions well enough to remember things well. Those people who voted win over the people who don't vote, whether they are right or wrong. Well, let's take a look.

Here is the recurrence. T(n) = T(1/10n) + T(9/10n) + Theta(n). And we will assume that this part here is less than or equal to some cn in order to analyze it. We will just do a recursion tree for this and see. Here is a recursion tree. We have T(n) = cn, T(1/10n), T(9/10n). Now we have again cn at the top. This gets complicated, right? This is 1/10cn. Now, over here we have 1/10. And then we are plugging it into the recursion again, so we now get T(1/100n) and over here we get T(9/100n). And over here we have now 9/10cn. And that gives us T(9/100n) again.

And here we get T(81/100n). And we keep going on. That is equal to cn, 1/10cn here. Down this way we have 1/100cn. And that keeps going down until we get to order 1 down here. And over here we have 9/10cn. And here, let's see, this is 9/100cn and this is now 9/100cn and this is 81/100cn. And these things keep going down until they get down to order 1. But the leaves are not all at uniform depth here, right? This side is way further up than this side, right? Because here we are only going down by 9/10 each time. So, in fact, what is the length of this path here?

What is the length of this path down to this, if I take the left most spine? Somebody raise there hand. Yeah? Log base 10 of n. Because I am basically cutting down by a factor of 10 each time. And how long does it take me to reduce it to 1? That is the definition, if you will, of what a log is, log base 10. What is this one? What is this path going that way? Log of n. Log base 10/9 of n.

Because we're going down by 9/10 each time. Once again, essentially the definition of n. And everything in between there is somewhere between log base 10 of n and log base 10/9 of n. So, everything is in between there. Now what I can do is do the trick that we did for mergesort in looking at what the evaluation of this is by adding up what is the cost of the total level. That is just cn. What is the cost of the next level? cn. And what is the cost of the next level? cn. Every level we are still doing the same amount of work. And we take that all the way down. And the last levels --

Eventually we hit some point where it is not equal to cn where we start getting things that are less than or equal to cn because some of the leaves start dropping out starting at this level. Basically this part is going to be log base 10 of n, and then we start getting things that are less than or equal to cn, and so forth, until finally we get to add it all up. T(n) is going to be less than or equal to cn times, well, what is the longest that this could possibly be? Log base 10/9 of n. Plus we have all of the leaves that we have to add in, but all the leaves together add up to just order n.

All the leaves add up to order n, so we have + Theta(n). And so this is how much? If I add all of this together, what is this asymptotically? That is n log n. So, T(n) is actually bounded by n log n. We are lucky. Those people who guessed lucky were right. A 1/10 : 9/10 split is asymptotically as good as a 50 : 50 split. And, in fact, we can lower bound this by just looking at these things here and discover that, in fact, T(n) is lower bounded by cn log base 10 of n + order n. And so T(n) is lower bounded by also asymptotically n log n. So, T(n) is actually Theta(n lg n).

Now, this is not really proof. I generally recommend that you don't do this kind of thing to do a proof. This is a good intuition of a recursion tree. The way you prove this is what? Substitution method. Good. What you do is use this to get your guess and then use substitution method to prove that your guess is right. It is too easy to make mistakes with this method. It is very easy to make mistakes.

With the substitution method it is harder to make mistakes because there is just algebra there that you are cranking through. It is easier to verify rather than dot, dot, dots and trees that you drew improperly and wrote in wrong amounts and so forth. OK? So, this is n log n. That's pretty good. It is order n log n. And we are lucky. Now let's try another one. This is all for intuition because, I will tell you, by the time we get to the end of this class you folks are going to bolting for the door because we are going to do some good math today, actually. It is actually fun math,

I think, but it is challenging. If you are not awake, you can still sleep now, but I will tell you when to wake up. One more bit of intuition. Suppose that we alternate steps.

Suppose we do the partitioning thing. And it happens that we start out lucky and then we have a partitioning step that is unlucky and then we have a step that is lucky and a step that is unlucky and we do that all the way down the tree. Suppose we alternate. Are we lucky or unlucky if we do that? This time I want everybody voting. It doesn't matter what your answer is. Everybody has to have a stake in the game. It is sort of like horseracing. If ever you have watched horseracing, it is really boring, but if you put a little bit of money down, a little skin in the game suddenly it is interesting. The same thing here.

I want everybody to put some skin in the game. Who thinks that this is going to be lucky? Who thinks it is going to be unlucky? OK. Who didn't vote? [LAUGHTER] You guys. No skin in the game, ha? Let's analyze this so we can once again write a recurrence. On the lucky step, we will have L(n) be the running time on a lucky step of size n. And that is going to be twice. While the next step is going to be unlucky. It is two unluckies over 2 plus order n. That is our lucky step. And then for the unlucky step it is essentially going to be L of n minus 1, it is going to be lucky on the next step, plus order n. That is unlucky.

See how I have described this behavior with a system now of recurrences that are dependent where the boundary cases, once again which are unstated, is that the recurrences have a constant solution with constant input. Now we just do a little bit of algebra using substitution. L(n) is then equal to, well, I can just plug in, for U(n/2) plug in the value of U(n/2). And that gives me 2[L(n/2-1) + Theta(n) + Theta(n)]. See what I did here? I simply plugged in, for U(n/2), this recurrence. In fact, technically I guess I should have said Theta(n/2) just to make this substitution more straightforward.

It is the same thing, but just to not skip a step. That we can now crank through. And that is 2L(n/2 - 1) +, and now I have two T(n/2) plus another one, so all of that is just order n. And what is the solution to that recurrence? n log n. Theta(n lg n). Does everybody see that? OK? Theta(n lg n). This is basically just, once again, master theorem with a little bit of jiggering here. That minus one is only going to help us, actually, in the solution of the master theorem. So, it is order n lg n. We are lucky. If we alternate lucky and unlucky, we are lucky. How can we insure that we are usually lucky? If I have the input already sorted, I am going to be unlucky.

Excuse me? You could randomly arrange the elements, that is one way. What is another way? That is a perfectly good way, actually. In fact, it is a common thing to do. Randomly choose the pivot, OK. It turns out those are effectively equivalent, but we are going to do the randomly choose the pivot because it is a little bit easier to analyze. But they are effectively equivalent. That gives us the algorithm called randomized quicksort.

And the nice thing about randomized quicksort is that the running time is independent of the input ordering. Very much for the same reason that if I just scramble the input, it would be independent of the input ordering. If I randomly scramble the input then it doesn't matter what the order of the input was. Whereas, original quicksort has some slow cases, input sorted or reverse sorted, and some fast cases. In particular, it turns out that if it is random it is going to be pretty fast.

If I actually randomly scramble the input or pivot on a random element, it doesn't matter what the input was. One way of thinking about this is with an adversary. Imagine your adversary, you are saying I have a good sorting algorithm and he says I have a good sorting algorithm and you're trying to sell to a single customer. And the customer says OK, you guys come up with benchmarks for each of your algorithms. And you get to look at his algorithm.

Well, you look and you say oh, he is using quicksort. I will just give him something that is already sorted. That is what you could do to him. If you had quicksort, he would do the same thing to you. So, how can you defeat him? Well, one way is with randomization. Big idea in computer science, use random numbers. The idea here is if I permute the ordering at random, as one suggestion, or I pivot at random places, then the input ordering didn't matter.

And so there is no bad ordering that he can provide that is going to make my code run slowly. Now, I might get unlucky. But that is just unlucky in my use of my random-number generator. It is not unlucky with respect to what the input was. What the input was doesn't matter. Everybody follow that? OK. The nice thing about randomized quicksort is that it makes no assumptions about the input distribution.

You don't have to assume that all inputs are equally likely because either you can make it that way or you pivot in a way that makes that effectively whole. And, in particular, there is no specific input that can elicit the worst-case behavior. The worst-case is determined only by a random-number generator. And, therefore, since it is only determined by a random-number generator, we can essentially bound the unluckiness mathematically. We can say what are the odds? So, we are going to analyze this. And this is where you know if you belong in this course or not. If you have skipped 6.042 or whatever, this is a good place to do the comparison.

Since it is going to be a little bit, why don't people just stand up for a moment and take a stretch break. Since this is going to be a nice piece of mathematics we are going to do, you are going to want to feel fresh for it. Stretch break is over. Analysis. Good. I think we are going to make this. I am sort of racing. There is a lot of stuff to cover today. Good. Let's let T(n) now be the random variable for the running time assuming --

Wow. I didn't even write here what we did here. So, we are going to pivot on a random element. That is the basic scheme we are going to do. And the way I do that, by the way, is just in the code for partition, rather than partitioning on the first element, before I do the partition, I just swap the first element with some other element in the array chosen at random, perhaps itself. So, they are all equally likely to be pivoted on. And then just run the ordinary partition.

This is a random variable for running in time assuming, we have to make an assumption for doing probability, the random numbers are independent. So that when I pivot in one place, it is independent of how I pivoted in some other place as I am running this algorithm. Then, to analyze this, what I am going to do is I want to know where we pivoted. For k = 0, 1, ..., n-1, let's let, for a particular partition, the random variable X_k = 1 if partition generates a k : n-k-1 split, and 0 otherwise.

In the partition routine, I am picking a random element to pivot on. And X_k is going to be my random variable that is 1 if it generates a split that has k elements on the left side and n-k-1 elements on the right side of the pivot. Some of those, too, of

course are n-1 because I also have the pivot. And 0 otherwise. So, I now have n random variables that I have defined associated with a single partition where all of them are going to be zero except one of them, whichever one happens to occur is going to have the value 1. This is called, by the way. What is the name of this type of random variable?

Bernoulli. Well, Bernoulli has other assumptions. It is an indicator random variable. It turns out it is Bernoulli, but that's OK. It is an indicator random variable. It just takes on the value of 0, 1. And Bernoulli random variables are a particular type of indicator random variable. Which it turns out these are. That is an indicator random variable. Indicator random variables are a good way when you are trying to understand what the sum of a bunch of things is. It is a good way to break apart your big random variables into smaller ones that can be analyzed. Let's just take a look at this indicator random variable. What is the expectation of X_k equal to?

In other words, what is the probability that I generate a k : n-k-1 split? X_k is, let's just write out what that means, just to refresh people's memory. That is 0 times the probability that X_k equals 0 plus 1 times the probability that X_k equals 1, which is equal, well, that is all zero. That is just equal to the probability that X_k equals 1. And that is a general property of indicator random variables, is that their expectation is the probability that they are 1. The nice thing about indicator random variables is it directly connects the probability to the expectation without any other terms going on. What is the probability of X_k equals 1? 1/n.

So, all splits are equally likely. And I have n elements, so each ones has a 1/n chance of being picked as the pivot. And, once you pick the pivot, that determines what is on the left and the right and so forth. So, it is 1/n. Everybody with me so far? More or less? OK. As I say, this is going to test whether you're in the class. If you go home and you study this and you cannot get it, and you have a deficiency in your math background in trying to take the course, this is a good indication that probably you have taken something a little over your head. Let's write out what T(n) is equal to here.

T(n) is going to be equal to T(0) + T(n-1) + Theta(n) if we get a 0 : n-1 split and is equal to T(1) + T(n-2) + order n if we have a 1 : n-2 split. And now down here it is going to be T(n-1) + T(0) + Theta(n) if we end up with an n-1 : 0 split. So, this is our recurrence for T(n). And, unfortunately, the recurrence is kind of hairy because it has got n cases. And this is, once again, where the brilliance of being able to use indicator random variables comes in. Because we will be able to take this case analysis and reduce it to mathematics so we don't have cases using indicator random variables.

And the way we do that is using the following trick of converting the cases into a summation. Let's just take a look at why these two things are the same. The indicator random variable is zero, except if you get the particular split. Therefore, this summation is going to be zero, except for that k which actually appeared in which case it is the value that we say it is. See the trick using multiplication by 0, 1 variable to handle all the cases?

I think that is damn clever. I think that is damn clever. And this is like the classic thing that you do with indicator random variables. It's one of the reasons they are a very powerful method. Because now we actually have a mathematical expression, hairy although it may be, for our recurrence. Now, what we are going to analyze is

the expected value of T(n). That is what we want to do. What is the expected value of T(n)? To do that, I just write the expected value of T(n) is equal to the expected value of this big summation. And now we can go ahead and start to evaluate the expected value of that summation. Everybody with me?

Yes? Any questions at this point? I see a thumbs up. That's nice to see. But I generally believe that what I want to see is no thumbs down. It is good to see the thumbs up, but that means one person understands, or thinks he understands. [LAUGHTER] So, this is, I claim, equal to the following. Actually, I am going to need a little space here so I am going to move the equal sign over a little bit.

I claim that summation is equal to that. This expectation is equal to that summation of expectations. Why is that? What are the magic words that justify this step? Linearity of expectation. The expectation of a sum is the sum of the expectations. So, that is linearity of expectation. I don't need independence for that. That is just always true for expectation of any random variables. The sum of the expectations is the expectation of the sum and vice versa. Here we did the vice versa. That is equal to now the sum of k=0 to n-1 of expectation of X_k [T(k) + T(n-k-1) + Theta(n)].

Why is that true? What I have done is I've said the expectation of the product is the product of the expectations. That is because of independence. What is independent of what? The X_k here, random variable, are independent of any of the other partitionings in, if you will, the X_k that would exist for any of the other recursive calls. So, whatever happens in here is independent of what happened there. We are actually hiding. Since we have a recurrence, we are not partitioning the same wage time. We have a different one.

We actually have something going on underneath the mathematics you have to pay attention to that the mathematics alone isn't really showing, which is that in T(k) there is actually a set of random choices that are being made, if you will. And so you have to understand that those are independent of those, in which case we can multiple the probabilities of their expectations. Is everybody with me? That is a big one, independence of X_k from other random choices.

That is equal to now, well, first of all, this is nice. What is the expectation of X_k? 1/n. That actually doesn't even belong in the summation. We will just pop it outside. I get 1/n times the sum of k=0 to n-1 of expectation of T(k) + 1/n summation k=0 to n-1 of expectation of T(n-k-1) + 1/n summation k=0 to n-1 up to Theta(n). That is, again, using linearity of expectation there this time to split up these pieces and just factoring out the expectation of k as being 1/n. Everybody with me still? All of this is elementary. It is just one of these things that is hard just because there are so many steps.

And it takes that you have seen some of this before. Now the next observation is that these two summations are, in fact, identical. They are the same summation, just in a different order. This is going T(0), T(1), T(2), T(3) up to T(n-1). This one is going T(n-1), T(n-2), T(n-3) down to T(0). These are, in fact, equal. So, therefore, I have two of them. And then what is this term equal to?

What is that one equal to? Theta(n). Let's just see why. The summation of 0 : n of Theta(n) is Theta(n^2) divided by n. Or, if I want to bring the Theta(n) out, I have 1 times the summation of k equals1 to n of Theta(1) or of 1. So, once again, you get n, either way of doing it. This is, in some sense, because the summations have

identical terms, and this is just algebra. Now what we are going to do is do something for technical convenience.

Because we are going to absorb the k=0, 1 terms into the Theta(n) for technical convenience. We have a recurrence here where I have an order n. And, if I look at the cases where k=0 or k=1, I know what the expectation is. For 0, 1, the expected cost is the worst case cost, which is constant. Because I am only solving the problem for a constant size. And we know that for any of the boundary cases that our solution of recurrence, our assumption is that it is constant time. So, I basically can just take those two terms out. And all that does it accumulate some more constant here in the Theta(n). It is going to make the solution of the recurrence a little bit easier.

And, if I do that, I get expectation of T(n) = 2/n summation k=2 to n-1 of expectation of T(k) + Theta(n). So, all of that work was to derive the recurrence. And now we have to solve it. Just to review what we did, we started out with a recurrence which was for the random variable which involved a case statement. We converted that into some mathematics without the case statement, just with a product, and then we derived a recurrence for the expectation. And now we are in the process of trying to solve that recurrence. We have done some simplification of the recurrence so that we understand what it is that we are going to solve here. By the way, I don't give things like this on quizzes. I do expect you to understand it.

The elements of this you will find on a quiz. This is a lot of work to figure out. This took smart people to do. Even though it is all elementary, but working out something like this at the elementary level is still a bit of work even for somebody who is knowledgeable in this area. Now we are going to solve that last recurrence over there and we are going to prove that the expectation of T(n) is less than or equal to (an lg n) for some constant a greater than 0. That is going to be what we are going to do. And so what technique do you think we should use to prove this? Does this look like a master method?

It is nothing like the master method. So, when in doubt do substitution. It is the grand-daddy of all methods. What we will do is solve the base case by simply choosing a big enough so that (an lg n) is bigger than the expectation of T(n) for sufficiently large small n. So, I just pick a to be big enough. And this is, by the way, why I wanted to exclude 0 and 1 from the recurrence. Because, for example, when n=0, log of 0 is, it's like dividing by 0, right, you cannot do it. Log of 1 is 0. So here, even if I restricted it to 1, here I would have a 0, and I can't ever pick a big enough to dominate those cases. What I do is I just say look, I just absorb whatever the cost is into the T(n) for technical convenience.

And that lets me address it as (an lg n) to be big enough to handle the base case. So, that is why we made that technical assumption. We are going to use a fact which is that the summation of k=2 to n-1 of k lg k is less than or equal to 1/2n^2 lg n - 1/8n^2. I am going to leave that as an exercise for you to workout. I think it is an exercise in the book, too. I want you to go and evaluate this. There are two ways to evaluate it. One is by using purely summations and facts about summations by splitting the summation into two pieces and reconstituting it to prove this bound.

The other way is to use the integral method for solving summations. Either way you can prove. The integral method actually gets you a tighter bound than this. This is a basic fact, and you should go off and know how to do that. Now let's do substitution.

The expectation of T(n) is less than or equal to 2/n times the summation k=2 to n-1, now we do the substitution of ak lg k, the smaller values plus Theta(n).

I might mentioned, by the way, that the hard part of doing this, it is easy to get the bound without this term, it is easy to get this bound, 1/2n^2 lg n, it is harder to get the second order term. It turns out you need the second order term in order to do what we are going to do. You have to be able to subtract a quadratic amount of the n^2 lg n in order to make this proof work. And that is the trickier part in evaluating that summation. So, we get this. That is less than or equal to? Well, I happen to know how much this is by using that formula. I use my fact and get 2a/n (1/2n^2 lg n - 1/8n^2) + Theta(n).

Did I do something wrong? There we go. Very good. That is equal to - If I multiply this first part through that is an lg n. And now, so I don't make a mistake, I want to express this as my desired, this is what I want it to be, minus a residual. I am going to write the residual as this part. And so, the way to write that is, that is going to be minus. And then it is going to be this term here, which is going to be an/4 - Theta(n).

And that is going to be less than or equal to an lg n if this part is positive. And I can make that part positive by picking a big enough such that an/4 dominates the constant in the Theta(n) here. Whatever the constant is here, I can find an a that is big enough so that this term makes this part positive. If a is big enough so that an/4 dominates Theta(n). And so the running time of randomized quicksort is order n lg n. That is what we just proved, the expected running time is order n lg n. Now, in practice, quicksort is a great algorithm. It is typically three or more times faster than mergesort. It doesn't give you the strong guarantee necessarily of mergesort and being worst-case n lg n.

But in practice, if you use randomized quicksort, it is generally as much as three times faster. It does require code tuning in order to get it up to be that fast. You do have to go and coarsen the base cases and do some other tricks there, but most good sorting algorithms that you will find are based on quicksort. Also one of the other reasons it works well is because it tends to work well with caches in virtual memory.

We are not really talking much about caching models and so forth, big topic these days in algorithms, but it does work very well with caches in virtual memory. It is another reason that this is a good algorithm to use. Good recitation, by the way, on Friday. We are going to see another n log n time algorithm, a very important one in recitation on Friday.