So, we're going to talk today about binary search trees. It's something called randomly built binary search trees. And, I'll abbreviate binary search trees as BST's throughout the lecture. And, you of all seen binary search trees in one place or another, in particular, recitation on Friday. So, we're going to build up the basic ideas presented there, and talk about how to randomize them, and make them good. So, you know that there are good binary search trees, which are relatively balanced, something like this. The height is log n. We called unbalanced, and that's good. Anything order log n will be fine. In terms of searching, it will then cost order log n. And, there are bad binary search trees which have really large height, possibly as big as n. So, this is good, and this is bad. We'd sort of like to know, we'd like to build binary search trees in such a way that they are good all the time, or at least most of the time. There are lots of ways to do this, and in the next couple of weeks, we will see four of them, if you count the problem set, I believe. Today, we are going to use randomization to make them balanced most of the time in a certain sense. And then, in your problem set, you will make that in a broader sense. But, one way to motivate this topic, so I'm not going to define randomly built binary search trees for a little bit. One way to motivate the topic is through sorting, our good friend. So, there's a natural way to sort n numbers using binary search trees. So, if I give you an array, A, how would you sort that array using binary search tree operations as a black box? Build the binary search tree, and then traverse it in order. Exactly. So, let's say we have some initial tree, which is empty, and then for each element of the array, we insert it into the tree. That's what you meant by building the search tree. So, we insert AI into the tree. This is the binary search tree insertion, standard insertion. And then, we do an in order traversal, which in the book is called in order tree walk. OK, you should know these algorithms are, but just for very quick reminder, tree insert basically searches for that element AI until it finds the place where it should have been if it was in the tree already, and then adds a new leaf there to insert that value. Tree walk recursively walks the left subtree, then prints out the root, and then recursively walks the right subtree. And, by the binary search tree property, that will print the elements out in sorted order. So, let's do a quick example because this turns out to be related to another sorting algorithm we've seen already. So, while the example is probably pretty trivial, the connection is pretty surprising. At least, it was to me the first time I taught this class. So, my array is three, one, eight, two, six, seven, five. And, I'm going to visit these elements in order from left to right, and just build a tree. So, the first element I see is three. So, I insert three into an empty tree. That requires no comparisons. Then I insert one. I see, is one bigger or less than three? It's smaller. So, I put it over here. Then I insert eight. That's bigger than three, so it get's a new leaf over here. Then I insert two. That sits between one and three. And so, it would fall off this right child of one. So, I add two there. Six is bigger than three, and less than eight. So, it goes here. Seven is bigger than three, and less than eight, bigger than six. So, it goes here, and five fits in between three and five, three and six rather. And so, that's the binary search tree that again. Then I run an in order traversal, which will print one, two, three, five, six, seven, eight. OK, I can run I quickly in my head because I've got a big stack. I've got to be a little bit careful. Of course, you should check that they come out in sorted order: one, two, three, five, six, seven, eight. And, if you don't have a big stack, you can go and buy one. That's always useful. Memory costs are going up a bit these days, or going down. They should be because of politics, but price-fixing, or whatever. So, the question is, what's the running time of the algorithm? Here, this is one of those answers where it depends. The parts that are easy to analyze are, well, initialization. The in order tree walk, how long does that take? n, good. So, it's order n for the walk, and for the initialization, which is constant. The question is, how long does it take me to do n tree inserts? Anyone want to guess any kind of answer to that question, other than it depends? I've already stolen the thunder there. Yeah? Big Omega of n log n, that's good. It's at least n log n. Why? Right. So, you gave two reasons. The first one is because of the decision tree lower bound. That doesn't actually prove this. You have to be a little bit careful. This is a claim that it's omega n log n all the time. It's certainly omega n log n in the worst case. Every comparison-based sorting algorithm is omega n log n in the worst case. It's also n log n every single time, omega n log n because of the second reason you gave, which is the best thing that could happen is we have a perfectly balanced tree. So, this is the figure that I have drawn the most on a blackboard in my life, the perfect tree on 15 nodes, I guess. So, if we're lucky, we have this. And if you add up all the depths of the nodes here, which gives you the search tree cost, in particular, these n over two nodes in the bottom, each have depth log n. And, therefore, you're going to have to pay it least n log n for those. And, if you're less balanced, it's going to be even worse. That takes some proving, but it's true. So, it's actually omega n log n all the time. OK, there are some cases, like you do know that the elements are almost already in order, you can do it in linear number comparisons. But here, you can't. Any other guesses at an answer to this question? Yeah? Big O n^2? Good, why? Right. We are doing n things, and each node has depth, at most, n. So, the number of comparisons we're making per element we insert, is, at most, n. So that's, at most, n^2. Any other answers? Is it possible for this algorithm to take n^2 time? Are there instances where it takes theta n^2? If it's already sorted, that would be pretty bad. So, if it's already sorted or if it's reverse sorted, you are in bad shape because then you get a tree like this. This is the sorted case. And, you compute. So, the total cost, the time in general is going to be the sum of the depths of the nodes for each node, X, in the tree. And in this case, it's one plus two plus three plus four, this arithmetic series. There's n of them, so this is theta n squared. It's like n^2 over two. So, that's bad news. The worst-case running time of this algorithm is n^2. Does that sound familiar at all, and algorithms worst-case running time is n^2, in particular, in the already-sorted case? But if we're lucky, at the lucky case, as we said, it's a balanced tree. Wouldn't that be great? Anything with omega log n height would give us a sorting algorithm that runs in n log n. So, in the lucky case, we are n log n. But in the unlucky case, we are n^2 and unlucky use sorted. Does it remind you of any algorithm we've seen before? Quicksort. It turns out the running time of this algorithm is the same as the running time of quicksort in a very strong sense. It turns out the comparisons that this algorithm makes are exactly the same comparisons that quicksort makes. It makes them in a different order, but it's really the same algorithm in disguise. That's the surprise here. So, in particular, we've already analyzed quicksort. We should get something for free out of that analysis. So, the relation is, BST sort and quicksort make the same comparisons but in a different order. So, let me walk through the same example we did before: three, one, eight, two, six, seven, five. So, there is an array. We are going to run a particular version of quicksort. I have to be a little bit careful here. It's sort of the obvious version of quicksort. Remember, our standard, boring quicksort is you take the first element as the partition element. So, I'll take three here. And, I split into the elements less than three, which is one and two. And, the elements bigger than three, which is eight, six, seven, five. And, in this version of quicksort, I don't change the order of the elements, eight, six, seven, five. So, let's say the order is preserved because only then will this equivalence hold. So, this is sort of a stable partition algorithm. It's easy enough to do. It's a particular version of quicksort. And soon, we're going to randomize it. And after we randomize, this difference doesn't matter. OK, then on the left recursion, we split in the partition element. There is things less than one, which is nothing, things bigger than one, which is two. And then, that's our partition element. We are done. Over here, we partition on eight. Everything is less than eight. So, we get six, seven, five, nothing on the right. Then we partition at six. We get things less than six, mainly five, things bigger than six, mainly seven. And, those are sort of partition elements in a trivial way. Now, this tree that we get on the partition elements looks an awful lot like this tree. OK, it should be exactly the same tree. And, you can walk through, what comparisons does quicksort make? Well, first, it compares everything to three, OK, except three itself. Now, if you look over here, what happens when we are inserting elements? Well, each time we insert an element, the first thing we do is compare with three. If it's less than, we go to the left branch. If it's greater than, we go to the right branch. So, we are making all these comparisons with three in both cases. Then, if we have an element less than three, it's either one or two. If it's one, we're done. No comparisons happen here one to one. But, we compare two to one. And indeed, when we insert two over there after comparing it to three, we compare it to one. And then we figure out that it happens here. Same thing happens in quicksort. For elements greater than three, we compare everyone to eight here because we are partitioning with respect to eight, and here because that's the next node after three. As soon as eight is inserted, we compare everything with eight to see in fact that's less than eight, and so on: so, all of the same comparisons, just in a different order. So, we turn 90âˆž. Kind of cool. So, this has various consequences in the analysis. So, in particular, the worst-case running time is theta n^2, which is not so exciting. What we really care about is the randomized version because that's what performs well. So, randomized BST sort is just like randomized quicksort. So, the first thing you do is randomly permute the array uniformly, picking all permutations with equal probability. And then, we call BST sort. OK, this is basically what randomized quicksort could be formulated as. And then, randomized BST sort is going to make exactly the same comparisons as randomized quicksort. Here, we are picking the root essentially randomly. And here in quicksort, you are picking the partition elements randomly. It's the same difference. OK, so the time of this algorithm equals the time of randomized quicksort because we are making the same comparisons. So, the number of comparisons is equal. And this is true as random variables. The random variable, the running time, this algorithm is equal to the random variable of this algorithm. In particular, the expectations are the same. OK, and we know that the expected running time of randomized quicksort on n elements is? Oh boy. n log n. Good. I was a little worried there. OK, so in particular, the expected running time of BST sort is n log n. Obviously, this is not too exciting from a sorting point of view. Sorting was just sort of to see this connection. What we actually care about, and the reason I've introduced this BST sort is what the tree looks like. What we really want is that search tree. The search tree can do more than sort. n order traversals are a pretty boring thing to do with the search tree. You can search in a search tree. So, OK, that's still not so exciting. You could sort the elements and then put them in an array and do binary search. But, the point of binary search trees, instead of binary search arrays, is that you can update them dynamically. We won't be updating them dynamically in this lecture, and we will in Wednesday and on your problem set. For now, it's just sort of warm-up. Let's say that the elements aren't changing. We are building one tree from the beginning. We have all n elements ahead of time. We are going to build it randomly. We randomly permute that array. Then we throw all the elements into a binary search tree. That's what BST sort does. Then it calls n order traversal. I don't really care about n order traversal. What I want, because we've just analyzed it. It would be a short lecture if I were done. What we want is this randomly built BST, which is what we get out of this algorithm. So, this is the tree resulting from randomized BST sort, OK, resulting from randomly permute in the array of just inserting those elements using the simple tree insert algorithm. The question is, what does that tree look like? And in particular, is there anything we can conclude out of this fact? The expected running time of BST sort is n log n. OK, I've mentioned cursorily what the running time of BST sort is, several times. It was the sum. So, this is the time of BST sort on n elements. It's the sum over all nodes, X, of the depth of that node. OK, depth starts at zero and works its way down because the root element, you don't make any comparisons beyond that, you are making whatever the depth is comparisons. OK, so we know that this thing is, in expectation we know that this is n log n. What does that tell us about the tree? This is for all nodes, X, in the tree. Does it tell us anything about the height of the tree, for example? Yeah? Right, intuitively, it says that the height of the tree is theta log n, and not n. But, in fact, it doesn't show that. And that's why if you feel that that's just intuition, but it may not be quite right. Indeed it's not. Let me tell you what it does say. So, if we take expectation of both sides, here we get n log n. So, the expected value of that is n log n. So, over here, well, we get the expected total depth, which is not so exciting. Let's look at the expected average depth. So, if I look at one over n, the sum over all n nodes in the tree of the depth of X, that would be the average depth over all the nodes. And what I should get is theta n log n over n because I divided n on both sides. And, I'm using, here, linearity of expectation, which is log n. So, what this fact about the expected running time tells me is that the average depth in the tree is log n, which is not quite the height of the tree being log n. OK, remember the height of the tree is the maximum depth of any node. Here, we are just bounding the average depth. Let's look at an example of a tree. I'll draw my favorite picture. So, here we have a nice balanced tree, let's say, on half of the nodes or a little more. And then, I have one really long path hanging off one particular leaf. It doesn't matter which one. And, I'm going to say that this path has length, with a total height here, I want to make root n, which is a lot bigger than log n. This is roughly log n. It's going to be log of n minus root n, or so, roughly. So, most of the nodes have logarithmic height and, sorry, logarithmic depth. If you compute the average depth in this particular tree, for most of the nodes, let's say it's, at most, n of the nodes have height log n. And then, there are root n nodes, at most, down here, which have depth, at most, root n. So, it's, at most, root n times root n. In fact, it's like half that, but not a big deal. So, this is n. So, this is n log n, or, sorry, average depth: I have to divide everything by n. n log n would be rather large for an average height, average depth. So, the average depth here is log n, but the height of the tree is square root of n. So, this is not enough. Just to know that the average depth is log n doesn't mean that the height is log n. OK, but the claim is this theorem for today is that the expected height of a randomly built binary search tree is indeed log n. BST is order log n. This is what we like to know because that tells us, if we just build a binary search tree randomly, then we can search in it in log n time. OK, for sorting, it's not as big a deal. We just care about the expected running time of creating the thing. Here, now we know that once we prove this theorem, we know that we can search quickly in expectation, in fact, most of the time. So, the rest of today's lecture will be proving this theorem. It's quite tricky, as you might imagine. It's another big probability analysis along the lines of quicksort and everything. So, I'm going to start with an outline of the proof, unless there are any questions about the theorem. It should be pretty clear what we want to prove. This is even weirder than most of the analyses we've seen. It's going to use a fancy trick, which is exponentiating a random variable. And to do that we need a tool called Jenson's inequality. We are going to prove that tool. Usually, we don't prove probability tools. But this one we are going to prove. It's not too hard. It's also basic analysis. So, the lemma, says that if we have what's called to a convex function, f, and you should all know what that means, but I'll define it soon in case you have forgotten. If you have a convex function, f, and you have a random variable, X, you take f of the expectation. That's, at most, the expectation of f of that random variable. Think about it enough and draw a convex function that is fairly intuitive, I guess. But we will prove it. What that allows us to do is instead of analyzing the random variable that tells us the height of a tree, so, X_n I'll call the random variable, RV, of the height of a BST, randomly constructed BST on n nodes we will analyze. Well, instead of analyzing this desired random variable, X_n, sorry, this should have been in capital X. We can analyze any convex function of X_n. And, we're going to analyze the exponentiation. So, I'm going to define Y_n to be two to the power of X_n. OK, the big question here is why bother doing this? The answer is because it works and it wouldn't work if we analyze X_n. We will see some intuition of that later on, but it's not very intuitive. This is our analysis where you need this extra trick. So, we're going to bound the expectation of Y_n, and from that, and using Jensen's inequality, we're going to get a bound on the expectation of X_n, a pretty tight bound, actually, because if we can bound the exponent up to constant factors, the exponentiation up to constant factors, we can bound X_n even better because you take logs to get X_n. So, we will even figure out what the constant is. So, what we will prove, this is the heart of the proof, is that the expected value of Y_n is order n^3. Here, we won't really know what the constant is. We don't need to. And then, we put these pieces together. So, let's do that. What we really care about is the expectation of X_n, which is the height of our tree. What we find out about is this fact. So, leave some horizontal space here. We get the expectation of two to the X_n. That's the expectation of Y_n. So, we learned that that's order n^3. And, Jensen's inequality tells us that if we take this function, two to the X, we plug it in here, that on the left-hand side we get two to the E of X. So, we get two to the E of X_n is at most E of two to the X_n. So, that's where we use Jensen's inequality, because what we care about is E of X_n. So now, we have a bound. We say, well, two to the E of X_n is, at most, n^3. So, if we take the log of both sides, we get E of X_n is, at most, the log of n^3. OK, I will write it in this funny way, log of order n^3, which will actually tell us the constant. This is three log n plus order one. So, we will prove that the expected height of a randomly constructed binary search tree on n nodes is roughly three log n, at most. OK, I will say more about that later. So, you've now seen the end of the proof. That's the foreshadowing. And now, this is the top-down approach. So, you sort of see what the steps are. Now, we just have to do the steps. OK, step one: take a bit of work, but it's easy because it's pretty basic stuff. Step two is just a definition and we are done. Step three is probably the hardest part. Step four, we've already done. So, let's start with step one. So, the first thing I need to do is define a convex function because we are going to manipulate the definition a fair amount. So, this is a notion from real analysis. Analysis is a fancy word for calculus if you haven't taken the proper analysis class. You should have seen convexity in any calculus class. A convex function is one that looks like this. OK, good. One way to formalize that notion is to consider any two points on this curve. So, I'm only interested in functions from reals to reals. So, it looks like this. This is f of something. And, this is the something. If I take two points on this curve, and I draw a line segment connecting them, that line segment is always above the curve. That's the meaning of convexity. It has a geometric notion, which is basically the same. But for functions, this line segment should stay above the curve. The line does not stay above the curve. If I extended it farther, it goes beneath the curve, of course. But, that segment should. So, I'm going to formalize that a little bit. I'll call this x, and then this is f of x. And, I'll call this y, and this is f of y. So, the claim is that I take any number between x and y, and I look up, and I say, OK, here's the point on the curve. Here's the point on the line segment. The value of that point on the y value, here, should be greater than or equal to the y value here, OK? To figure out what the point is, we need some, I would call it geometry. I'm sure it's an analysis concept, too. But, I'm a geometer, so I get to call it geometry. If you have two points, p and q, and you want to parameterize this line segment between them, so, I want to parameterize some points here, the way to do it is to take a linear combination. And, if you should have taken some linear algebra, linear combination look something like this. And, in fact, we're going to take something called an affine combination where alpha plus beta equals one. It turns out, if you take all such points, some number, alpha, times the point, p, plus some number, beta times the point, q, where alpha plus beta equals one. If you take all those points, you get the entire line here, which is nifty. But, we don't want the entire line. If you also constrained alpha and beta to be nonnegative, you just get this line segment. So, this forces alpha and beta to be between zero and one because they have to sum to one, and they are nonnegative. So, what we are going to do here is take alpha times x plus beta times y. That's going to be our point between with these constraints: alpha plus beta equals one. Alpha and beta are greater than or equal to zero. Then, this point is f of that. This is f of alpha x plus beta, y. And, this point is the linear interpolation between f of x and f of y, the same one. So, it's alpha times f of x plus beta times f of y. OK, that's the intuition. If you didn't follow it, it's not too big a deal because all we care about are the symbolic answer for proving things. But, that's where this comes from. So, here's the definition. Its function is convex. If, for all x and y, and all alpha and beta are greater than or equal to zero, whose sum is one, we have f of alpha x plus beta y is less than or equal to alpha f of x plus beta f of y. So, that's just saying that this y coordinate here is less than or equal to this y coordinate. OK, but that's the symbolism behind that picture. OK, so now we want to prove Jensen's inequality. OK, we're not quite there yet. We are going to prove a simple lemma, from which it will be easy to derive Jenson's equality. So, this is the theorem we are proving. So, here's a lemma about convex functions. You may have seen it before. It will be crucial to Jensen's inequality. So, suppose, this is a statement about affine combinations of n things instead of two things. So, this will say that convexity can be generalized to taking n things. So, suppose we have n real numbers, and we have n values alpha i, alpha one up to alpha n. They are all nonnegative. And, their sum is one. So, the sum of alpha k, I guess, k equals one to n, is one. So, those are the assumptions. The conclusion is the same thing, but summing over all k. So, k equals one to n, alpha_k * x_k. Take f of that versus taking the sum of the alphas times the f's. k equals one to n. So, the definition of convexity is exactly that statement, but where n equals two. OK, alpha one and alpha two are alpha and beta. This is just a statement for general n. And, you can interpret this in some funnier way, which I won't get into. Oh, sure, why not? I'm a geometer. So, this is saying you take several points on this curve. You take the polygon that they define. So, these are straight-line segments. You take the interior. If you take an affine combination like that, you will get a point inside that polygon, or possibly on the boundary. The claim is that all those points are above the curve. Again, intuitively: true if you draw a nice, canonical convex curve, but in fact, it's true algebraically, too. It's always a good thing. Any suggestions on how we might prove this theorem, this lemma? It's pretty easy. So, what technique might we use to prove it? One word: induction. Always a good answer, yeah. Induction should shout out at you here because we already know that this is true by definition of convexity for n equals two. So, the base case is clear. In fact, there's an even simpler base case, which is when n equals one. If n equals one, then you have one number that sums to one. So, alpha one is one. And so, nothing is going on here. This is just saying that f of one times x_1 is, at most, one times f of x_1: so, not terribly exciting because that holds with the quality. OK, so we don't even need the n equals two base case. So, the interesting part, although still not terribly interesting, is the induction step. This is good practice in induction. So, what we care about is this f of this linear combination, f on combination, x_k times x_k summed over all k. Now, what I would like to do is apply induction. What I know about inductively, is say f of this sum, if it's summed only up to n minus one instead of all the way up to n. Any smaller sum I can deal with by induction. So, I'm going to try and get rid of the nth term. I want to separate it out. And, this is fairly natural if you've played with affine combinations before. But it's just some algebra. So, I want to separate out the alpha_n*x_n term. And, I'd also like to make it an affine combination. This is the trick. Sorry, no f here. If I just removed the last term, the alpha k's from one up to n minus one wouldn't sum to one anymore. They'd sum to something smaller. So, I can't just take out this term. I'm going to have to do some trickery here, x_k plus the f. Good. So, you should see why this is true, because the one minus alpha n's cancel. And then, I'm just getting the sum of alpha_k*x_k, k equals one to n minus one, plus the alpha_n*x_n term. So, I haven't done anything here. These are equal. But now, I have this nifty feature, that on the one hand, these two numbers, alpha n and one minus alpha n sum to one. And on the other hand, if I did it right, these numbers should sum up to one just going from one up to n minus one. Why do they sum up to one? Well, these numbers summed up to one minus alpha n. And so, I'm dividing everything by one minus alpha n. So, they will sum to one. So now, I have two affine combinations. I just apply the two things that I know. I know this affine combination will work because, well, why? Why can I say that this is alpha n f of x_n plus one minus alpha n f of this crazy sum? Shout it out. There are two possible answers. One is correct, and one is incorrect. So, which will it be? This should have been less than or equal to. That's important. It's on the board. It can't be too difficult. So, I'm treating this as just one big X value. So, I have some x_n, and I have some crazy X. I want f of the affine combination of those two X values is, at most, the affine combinations of the f's of those X values. This is? It is the inductive hypothesis where n equals two. Unfortunately, we didn't prove the n equals two case is a special base case. So, we can't use induction here the way that I've stated the base case. If you did n equals two base case, you can do that. Here, we can't. So, the other answer is by convexity, good. That's right here. So, f is convex. We know that this is true for any two X values, and provided these two sum to one. So, we know that this is true. Now is when we apply induction. So, now we are going to manipulate this right term by induction. See, before we didn't necessarily know that n was bigger than two. But, we know that n is bigger than n minus one. That much, I can be sure of. So, this is one minus alpha n times the sum, k equals one to n minus one of alpha k over one minus alpha n times f of x_k, if I got that right. This is by induction, the induction hypothesis, because these alpha k's over one minus alpha n sum to one. Now, these one minus alpha n's cancel, and we just get what we want. This is sum k equals one to n of alpha k, f of x_k. So, we get f of the sum is, at most, sum of the f's. That proves the lemma. OK, a bit tedious, but each step is pretty straightforward. Do you agree? Now, it turns out to be relatively straightforward to prove Jensen's inequality. That's the magic. And then, we get to do the expectation analysis. So, we use our good friends, indicator random variables. OK, but for now, we just want to prove this statement. If we have a convex function, f of the expectation is, at most, expectation of f of that random variable. OK, this is a random variable, right? If you want to sample from this random variable, you sample from X, and then you apply f to it. That's the meaning of this notation, f of X because X is a random variable. We get to use that f is convex. OK, it turns out this is not hard, if you remember the definition of expectation, oh, I want to make one more assumption here, which is that X is integral. So, it's an integer random variable, meaning it takes integer values. OK, that's all we care about because we're looking at running times. This statement is true for continuous random variables, too, but I would like to do the discrete case because then I get to write down what U of X is. So, what is the definition of E of X? X only takes on integer values. This is easy, but you have to remember it. It's a good drill. I don't really know much about X except that it takes on integer values. Any suggestions on how I should expand the expectation of X? How many people know this by heart? OK, it's not too easy then. Well, expectation has something to do with probability, right? So, I should be looking at something like the probability that X equals some value, x. That would seem like a good thing to do. What else goes here? A sum, yeah. The sum, well, X could be somewhere between minus infinity and infinity. That's certainly true. And, we have some more. There's something missing here. What is this sum? What does it come out to for any random variable, X, that takes on integer values? One, good. So, I need to add in something here, namely X. OK, that's the definition of the expectation. Now, f of a sum of things, where these coefficients sum to one looks an awful lot like the lemma that we just proved. OK, we proved it in the finite case. It turns out, it holds just as well if you take all integers. So, I'm just going to assume that. So, I have these probabilities, these alpha values sum to one. Therefore, I can use this inequality, that this is, at most, let me get this right, I have the alphas, so I have a sum, x equals minus infinity to infinity of the alphas, which are a probability; capital X equals little x times f of the value, f of little x. OK, so there it is. I've used the lemma. So, maybe now I'll erase the lemma. OK, I cheated by using the countable version of the lemma while only proving the finite case. It's all I can do in lecture. So, this is by a lemma. Now, what I'd like to prove and leave some blank space here is this is, at most, E of f of X, so that this summation is, at most, E of f of X. Actually, it's equal to E of f of X. And, it really looks kind of equal, right? You've got sum of some probabilities times f of X. It almost looks like the definition of E of f of X, but it isn't. You've got to be a little bit careful because E of f of X should talk about the probability that f of X equals a particular value. We can relate these as follows. It's not too hard. You can look at each value that f takes on, and then look at all the values, k, that map to that value, x. So all the k's where f of X equals x, the probability that X equals k, OK, this is another way of writing the probability that f of X equals x. OK, so, in other words, I'm grouping the terms in a particular way. I'm saying, well, f of X takes on various values. Clever me to switch. I used to use k's unannounced, so I better call this something else. Let's call this Y, sorry, switch notation here. It makes sense. I should look at the probability that X equals x. So, what I really care about is what this f of X value takes on. Let's just call it Y, look at all the values, Y, that f could take on. That's the range of f. And then, I'll look at all the different values of X where f of X equals Y. If I add up those probabilities, because these are different values of X. Those are sort of independent events. So, this summation will be the probability that f of X equals Y. This is capital X. This is little y. And then, if I multiply that by y, I'm getting the expectation of f of X. So, think about this, these two inequalities hold. This may be a bit bizarre here because these sums are potentially infinite. But, it's true. OK, this proves Jensen's inequality. So, it wasn't very hard, just a couple of boards, once we had this powerful convexity lemma. So, we just used convexity. We used the definition of E of X. We used convexity. That lets us put the f's inside. Then we do this regrouping of terms, and we figure out, oh, that's just E of f of X. So, the only inequality here is coming from convexity. All right, now comes the algorithms. So, this was just some basic probability stuff, which is good to practice. OK, we could see in the quiz, which is not surprising. This is the case for me, too. You have a lot of intuition with algorithms. Whenever it's algorithmic, it makes a lot of sense because you're sort of grounded in some things that you know because you are computer scientists, or something of that ilk. For the purposes of this class, you are computer scientists. But, with sort of the basic probability, unless you happen to be a mathematician, it's less intuitive, and therefore harder to get fast. And, in quiz one, speed is pretty important. On the final, speed will also be important. The take home certainly doesn't hurt. So, the take home is more interesting because it requires being clever. You have to actually be creative. And, that really tests algorithmic design. So far, we've mainly tested analysis, and just, can you work through probability? Can you figure out what the, can you remember what your running time of randomized quicksort is, and so on? Quiz two will actually test creativity because you have more time. It's hard to be creative in two hours. OK, so we want to analyze the expected height of a randomly constructed binary search tree. So, I've defined this before, but let me repeat it because it was a while ago almost at the beginning of lecture. I'm going to take the random variable of the height of a randomly built binary search tree on n nodes. So, that was randomized, the n values. Take a random permutation, insert them one by one from left to right with tree insert. What is the height of the tree that you get? What is the maximum depth of any node? I'm not going to look so much at X_n. I'm going to look at the exponentiation of X_n. And, still we have no intuition why. But, two to the X is a convex function. OK, it looks like that. It's very sharp. That's the best I can do for drawing, two to the X. You saw how I drew my histogram. So, we want to somehow write this random variable as something, OK, in some algebra. The main thing here is to split into cases. That's how we usually go because there's lots of different scenarios on what happens. So, I mean, how do we construct a tree from the beginning? First thing we do is we take the first node. We throw it in, make it the root. OK, so whatever the first value happens to be in the array, which we don't really know how that falls into sorted order, we put it at the root. And, it stays the root. We never change the root from then on. Now, of all the remaining elements, some of them are less than this value, and they go over here. So, let's call this r at the root. And, some of them are greater than r. So, they go over here. Maybe there's more over here. Maybe there's more over here. Who knows? Arbitrary partition, in fact, uniformly random partition, which should sound familiar, whether there are k elements over here, and n minus k minus one elements over here, for any value of k, that's equally likely because this is chosen uniformly. The root is chosen uniformly. It's the first element in a random permutation. So, what I'm going to do is parameterize by that. How many elements are over here, and how many elements are over here? Because this thing is, again, a randomly built binary search tree on however many nodes are in there because after I pick r, it's determined who is to the left and who is to the right. And so, I can just partition. It's like running quicksort. I partition the elements left of r, the elements right of r, and I'm sort of recursively constructing a randomly built binary search tree on those two sub-permutations because sub-permutations of uniform permutations are uniform. OK, so these are essentially recursive problems. And, we know how to analyze recursive problems. All we need to know is that there are k minus one elements over here, and n minus k elements over here. And, that would mean that r has rank k, remember, rank in the sense of the index in assorted order. So, where should I go? So, if the root, r, has rank, k, so if this is a statement about condition on this event, which is a random event, then what we have is X_n equals one plus the max of X_(k minus one), X_(n minus k) because the height of this tree is the max of the heights of the two subtrees plus one because we have one more level up top. OK, so that's the natural thing to do. What we are trying to analyze, though, is Y_n. So, for Y_n, we have to take two to this power. So, it's two times the max of two to the X_(k minus one), which is Y_(k minus one), and two to this, which is Y_(n minus k). And, now you start to see, maybe, why we are interested in Y's instead of X's in the sense that it's what we know how to do. When we solve a recursion, when we solve, like, the expected running time, we haven't taken expectations, yet, here. But, when we compute the expected running time of quicksort, we have something like two times, I mean, we have a couple of recursive subproblems, which are being added together. OK, here, we have a factor of two. Here, we have a max. But, intuitively, we know how to multiply random variables by a constant because that's, like, there's two recursive subproblems of the size is equal to the max of these two, which we don't happen to know here. But, there it is, whereas one plus, we don't know how to handle so well. And, indeed, our techniques are really good at solving recurrences, except up to the constant factors. And, this one plus really doesn't affect the constant factor too much, it would seem. OK, but it's a big deal. In exponentiation, it's a factor of two. So here, it's really hard to see what this one plus is doing. And, our analysis, if we tried it, and it's a good idea to try it at home and see what happens, if you tried to do what I'm about to do with X_n, the one plus will sort of get lost, and you won't get a bound. You just can't prove anything. With a factor of two, we're in good shape. We sort of know how to deal with that. We'll say more when we've actually done the proof about why we use Y_n instead of X_n. But for now, we're using Y_n. So, this is sort of a recursion, except it's conditioned on this event. So, how do I turn this into a statement that holds all the time? Sorry? Divide by the probability of the event? More or less. Indeed, these events are independent. Or, they're all equally likely, I should say. They're not independent. In fact, one determines all the others. So, how do I generally represent an event in algebra? Indicator random variables: good. Remember your friends, indicator random variables. All of these analyses use indicator random variables. So, they will just represent this event, and we'll call it Z_nk. It's going to be one if the root has rank, k, and zero otherwise. So, in particular, the probability of, these things are all equally likely for, a particular value of n if you try all the values of k. The probability that this equals one, which is also the expectation of that indicator random variable, which you should know, is it only takes values one or zero. The zero doesn't matter in the expectation. So, this is going to be, hopefully, one over n if I got right. So, there are n possibility of what the rank of the root could be. Each of them are equally likely because we have a uniform permutation. So, now, I can rewrite this condition statement as a summation where the Z_nk's will let me choose what case I'm in. So, we have Y_n is the sum, k equals one to n of Z_nk times two times the max of X, sorry, Y, k minus one, Y_n minus k. So, now we have our good friend, the recurrence. We need to solve it. OK, we can't really solve it because this is a random variable, and it's talking about recursive random variables. So, we first take the expectation of both sides. That's the only thing we can really bound. Y_n could be n^2 in an unlucky case, sorry, not n^2. It could be n^2. It could be two to the, boy, two to the n if you are unlucky because X_n could be as big as n, the height of the tree. And, Y_n is two to that. So, it could be two to the n. What we want to prove is that it's polynomial in n. If it's n to some constant, and we take logs, it'll be order log n. OK, so we'll take the expectation, and hopefully that will guarantee that this holds. OK, so we have expectation of this summation of random variables times recursive random variables. So, what is the first, woops, I forgot a bracket. What is the first thing that we do in this analysis? This should, yeah, linearity of expectation. That one's easy to remember. OK, we have a sum. So, let's put the E inside. OK, now we have the expectation of our product. What should we use? Independence. Hopefully, things are independent. And then, we could write this. Then, it would be the expectation of the product. And, heck, let's put the two outside, because it's not, no sense in keeping it in here. Y is there starting to look like X's? I can't even read them. Sorry about that. This should all be Y's. OK, very wise, random variables. So. Why are these independent? So, here we are looking at the choice of what the root is, what rank the root has in a problem of size n. In here, we're looking at what the root, I mean, there are various choices of what the search tree looks like in the stuff left of the root, and in the stuff right of the root. Those are independent choices because everything is uniform here. So, the choice of this guy was uniform. And then, that determines who partitions in the left and the right. Those are completely independent recursive choices of who's the root in the left subtree? Who's the root in the left of the left subtree, and so on? So, this is a little trickier than usual. Before, it was random choices in the algorithm. Now, it's in some construction where we choose the random numbers ahead of time. It's a bit funny, but this is still independent. So, we get this just like we did in quicksort, and so on. OK. Now, we continue. And, now it's time to be a bit sloppy. Well, one of these things we know. OK, E of ZNK, that, we wrote over here. It's one over n. So, that's cool. So, we get a two over n outside, and we get this sum of the expectation of a max of these two things. Normally, we would write, well, I think sometimes you write T of max, or Y of the max of the two things here. You've got to write it as the max of these two variables. And, the trick, I mean, it's not too much of a trick, is that the max is, at most, the sum. So, we have nonnegative things. So, we have two over n, sum k equals one to n of the expectation of the sum instead of the max. OK, this is, in some sense, the key step where we are losing something in our bound. So far, we've been exact. Now, we're being pretty sloppy. It's true the max is, at most, the sum. But, it's a pretty loose upper bound as things go. We'll keep that in mind for later. What else can we do with the summation? This should, again, look familiar. Now that we have a sum of a sum of two things, I'm trying to like it to be a sum of one thing. Sorry? You can use linearity of expectation, good. So, that's the first thing I should do. So, linearity of expectation lets me separate that. Now I have a sum of 2n things. Right, I could break that into the sum of these guys, and the sum of these guys. Do you know anything about those two sums? Do we know anything about those two sums? They're the same. In fact, every term here is appearing exactly twice. One says a k minus one. One says an n minus k, and that even works if it's odd, I think. So, in fact, we can just take one of the sums and multiply it by two.

So, this is four over n times the sum, and I'll rewrite it a little bit from zero to n minus one of E of Y_k. Just check the number of times each Y_k appears from zero up to n minus one is exactly two. So, now I have a recurrence. I have E of Y_n is, at most, this thing. Let's just write that for our memory. So, how's that? Cool. Now, I just have to solve the recurrence. How should I solve an ugly, hairy, recurrence like this? Substitution: yea! Not the master method. OK, it's a pretty nasty recurrence. So, I'm going to make a guess, and I've already told you the guess, that it's n^3. I think n^3 is pretty much exactly where this proof will be obtainable. So, substitution method, substitution method is just a proof by induction. And, there are two things every proof by induction should have, well, almost every proof by induction, unless you're being fancy. It should have a base case, and the base case here is n equals order one. I didn't write it, but, of course, if you have a constant size tree, it has constant height. So, this thing will be true as long as we set true if c is sufficiently large. OK, so, don't forget that. A lot of people forgot it on the quiz. We even mentioned the base case. Usually, we don't even mention the base case. And, you should assume that there's one there. And, you have to say this in any proof by substitution. OK, now, we have the induction step. So, I claim that E of Y_n is, at most, Ccof n^3, assuming that it's true for smaller n. You should write the induction hypothesis here, but I'm going to skip it because I'm running out of time. Now, we have this recurrence that E of Y_n is, at most, this thing. So, E of Y_n is, at most, four over n, sum k equals zero to n minus one of E of Y_k. Now, notice that k is always smaller than n. So, we can apply induction. So, this is, at most, four over n, sum k equals zero to n minus one of c times k^3. That's the induction hypothesis. Cool. Now, I need an upper bound on this sum, if you have a good memory, then you know a closed form for this sum. But, I don't have such a good memory as I used to. I never memorized this sum when I was a kid, so I don't remember everything when I memorize when I was less than 12 years old. I still remember all the digits of pi, whatever. But, anything I try to memorize now just doesn't quite stick the same way. So, I don't happen to know this sum. What's a good way to approximate this sum? Integral: good. So, in fact, I'm going to take the c outside. So, this is 4c over n. The sum is, at most, the integral. If you get the range right, so, you have to go one larger. Instead of n minus one, you go up to n. This is in the textbook. It's intuitive, too, as long as you have a monotone function. That's key. So, you have something that's like this. And, you know, the sum is taking each of these and weighting them with a value of one. The integral is computing the area under this curve. So, in particular, if you look at this approximation of the integral, then, I mean, this thing is certainly, this would be the sum if you go one larger at the end, and that's, at most, the integral. So, that's proof by picture. But, you can see this in the book. You should know it from 042 I guess. Now, integrals, hopefully, you can solve. Integral of x^3 is x^4 over four. I got it right. And then, we're valuing that at n. And, it's zero. Subtracting the zero doesn't matter because zero to the fourth power is zero. So, it's just n^4 over four. So, this is 4c over n times n^4 over four. And, conveniently, this four cancels with this four. The four turns into a three because of this, and we get n^3. We get cn^3. Damn convenient, because that's what we wanted to prove. OK, so this proof is just barely snaking by: no residual term. We've been sloppy all over the place, and yet we were really lucky. And, we were just sloppy in the right places. So, this is a very tricky proof. If you just tried to do it by hand, it's pretty easy to be too sloppy, and not get quite the right answer. But, this just barely works. So, let me say a couple of things about it in my remaining one minute. So, we can do the conclusion, again. I won't write it because I don't have time, but here it is. We just proved a bound on Y_n, which was two to the power X_n. What we cared about was X_n. So, we used Jensen's inequality. We get the two to the E of X_n is, at most, E of two to the X_n. This is what we know about because that's Y_n. So, we know E of Y_n is now order n^3. OK, we had to set this constant sufficiently large for the base case. We didn't really figure out what the constant was here. It didn't matter because now we're taking the logs of both sides. We get E of X_n is, at most, log of order n^3. This constant is a multiplicative constant. So, you take the logs. It becomes additive. This constant is an exponent. So, it would take logs. It becomes a multiple. Three log n plus order one. This is a pretty damn tight bound on the height of a randomly built binary search tree, the expected height, I should say. In fact, the expected height of X_n is equal to, well, roughly, I'll just say it's roughly, I don't want to be too precise here, 2.9882 times log n. This is the result by a friend of mine, Luke Devroy, if I spell it right, in 1986. He's a professor at McGill University in Montreal. So, we're pretty close, three to 2.98. And, I won't prove this here. The hard part here is actually the lower bound, but it's only that much. I should say a little bit more about why we use Y_n instead of X_n. And, it's all about the sloppiness. And, in particular, this step, where we said that the max of these two random variables is, at most, the sum. And, while that's true for X just as well as it is true for Y, it's more true for Y. OK, this is a bit weird because, remember, what we're analyzing here is all possible values of k. This has to work no matter what k is, in some sense. I mean, we're bounding all of those cases simultaneously, the sum of them all. So, here we're looking at k minus one versus n minus k. And, in fact, here, there's a polynomial version. But, so, if you take two values a and b, and you say, well, max of ab is, at most, a plus b. And, on the other hand you say, well, max of two to the a and two to the b is, at most, two to the a plus two to the b. Doesn't this feel better than that? Well, they are, of course, the same. But, if you look at a minus b, as that grows, this becomes a tighter bound faster than this becomes a tighter bound because here we're looking at absolute difference between a minus b. So, that's why this is pretty good and this is pretty bad. We're still really bad if a and b are almost the same. But, we're trying to solve this for all partitions into k minus one and n minus k. So, it's OK if we get a few of the cases wrong in the middle where it evenly partitions. But, as soon as we get some skew, this will be very close to this, whereas this will be still pretty far from this. You have to get pretty close to the edge before you're not losing much here, whereas pretty quickly you're not losing much here. That's the intuition. Try it, and see what happens with X_n, and it won't work. See you Wednesday.