PROFESSOR: Last lecture on sorting. Yay. And it's one of the coolest lectures on sorting, I would say. We're going to talk about linear-time sorting, when it's possible and when it's not possible, and this lecture sort of follows the tried and tested mathematical structure which is theorem, proof, counterexample. So we're going to start with a theorem which is that sorting requires n lg n time at least in the worst case and we're going to then prove that in fact, you can get away with linear time sometimes.

Both of these terms are correct, but they're slightly different models of computation. Remember models of computation from lecture two? So we're going to talk about a new model of computation, which we've sort of been using for most algorithms lately, called the comparison model. And it's a model of computations that's really useful for proving lower bounds which we haven't done much of yet.

We're going to prove two very simple lower bounds. One is that searching requires lg n time. This is basically binary search is optimal. And the other is that sorting requires n lg n time. This is that merge sort is optimal.

And then we're going to break outside of that comparison model, work in a different model of computation, our more usual RAM model, and show that in certain situations, we can get linear time. So that's the plan. Let's start with this comparison model.

So the idea in the comparison model is to restrict what kind of operations we can do to be comparisons. It's very straightforward. All input items are black boxes, you could say, in that you don't really know what they are. And a formal notion of black boxes is something we talked about last class at the end, abstract data type.

So it's a data structure, if you will. Every item that you're given is a data structure.

You want to sort them. And the data structure supports a single operation which is compared to another one. Only operation allowed-- I guess I should say plural actually-- are comparisons.

I'm going to be nice and I'll let you do less than, less than or equal to, greater than, whatever. I guess there's only one other. Well, there's two more, greater than or equal to and equals. So you can do all the usual comparisons. You get a binary answer, yes or no, and those the only operations you're given.

And basically the last four lectures have all been about algorithms in this model. So merge sort, it moves items around. It's changing pointers to items, but the only way it manipulates items or evaluates them is to compare one against the other. Heaps and heaps sort also only compare. Binary search trees only compare. Everything we've been seeing so far is about comparisons.

And so all the algorithms we've seen so far are in this model and we're going to prove that they are optimal in this model. That's the plan. I should also define the cost of an algorithm. Time cost is just going to be the number of comparisons. This is the weird part, I guess, of the model.

So in everything we've done so far, we've been in, I guess, pointer machine or RAM, either way. We've been showing binary search trees or AVL trees, you can do order lg n time, in the regular notion of time. But in particular they do order lg n comparisons. And what we're going to show on the-- this is only interesting from a lower bound perspective-- we're going to show that even if you just count comparisons, you can do whatever other crazy things you want. You need lg n time to search. You need n lg n time to sort. So that's our goal.

So to prove that we're going to introduce the notion of a decision tree. So the idea is the following: if we know that our algorithms are only comparing items, we can actually sort of draw all the possible things that an algorithm could do, so any comparison algorithm. So this focusing in on comparisons lets us take a tree perspective of what our algorithm does-- all possible comparisons and their outcomes and the resulting answer.

I think this would be a lot clearer if we look at an example-- binary search, how you search a simple algorithm. Look at the middle compared to the item you're searching for go left or go right. And our idea-- I didn't write it here-- is to look at a particular value of n, n being the size of your problem, so binary search, you're searching among n items for another item.

And I'm going to keep it simple, n equals three. I think I'm going to go a little wide, use the whole board. So n equals 3 we've got array, say index turning at zero-- pretty simple binary search-- look in the middle, go left or go right. But I'm going to write out this algorithm explicitly to say, all right, first thing I do is compare is A 1 less than x?

That's in all cases, no matter what the array is as long as n equals three, this is the first operation you do. The answer is either yes or no. If the answer is no, that means x is less than or equal to A 1, so it's to the left.

Then we compare with A 0. Is A 0 less than x? Answer is either yes or no. If the answer is no, we're kind of done. We know that x is over here or it might actually be equal to A 0. If you want to figure out whether it's equal or less than, there will be one more step. But I'll just stop it here.

We'll say, well in this case, x is less than or equal to A 0. I'm going to put it in a box, say that's done where the circles are the decisions. OK? If the answer is yes-- there's no to this question, yes to this question-- then you know that x falls in between here.

Probably need a wider box for this. So we have A 0 is less than x-- that was the yes to this-- and the note of this means that x is less than or equal to A 1, and so we basically identified where x fits. It's in between those two values, possibly equal to this one. Again, one more comparison, you could figure that out.

And then if x is to the right of A 1, so this is true, then we check x against A 2 and the answer is either no or yes. And in the no case-- well I've conveniently laid things out here, it's sequential-- in the yes case, x is bigger than A 2 so it's outside the

array. It's to the right. That's the answer. Set. Yeah. And in the other case, it's in between A 1 and A 2.

It's a tedious process to write out an algorithm like this because a binary search-- it's not so bad-- but if you tried to write a sorting algorithm out like this where the answers are down the bottom, here's the sorted order, and all the comparisons you do here, the tree will actually be of exponential size. So you don't actually want to represent an algorithm like this unless you're just trying to analyze it. But it's nice to think of an algorithm this way because you see all possible executions all at once.

Let me talk about some features of this tree versus the algorithm. So every internal node-- actually, I'm going to go over here. So we have a decision tree and we have an algorithm that decision tree represents. And so when we have an internal node in the decision tree, that corresponds to a binary decision in the algorithm. In this case, we're only looking at comparisons.

Slight technical detail, decision trees are a little more general than comparisons. Could be any binary decision here and everything I say will be true about any binary decision you make. Comparisons are the ones we kind of care about because all of our algorithms are doing that. And then a leaf in this tree corresponds to-- it stores or it represents that you've found the answer. Maybe I'll say found.

When the algorithm terminates, returns some answer to the problem, that's what we write down here. Here's where x is in this array and, yeah, we're done. What else do we have?

Here's some puzzles for you. If I just wanted to look at a single execution of the algorithm, what does that correspond to in the tree? Yeah.

AUDIENCE: Going from the root all the way down to the leaf.

PROFESSOR: Going from the root down to a leaf. This is what I normally call a root-to-leaf path, technically a downward root-to-leaf path. How about the running time of that execution? How long does it take? Keep going?

AUDIENCE: Lg n.

PROFESSOR: Lg n in binary search, but in general.

AUDIENCE: The length of that path.

PROFESSOR: The length of that path. Yeah. Got to make sure we get n cases right but I think it's correct, so if here is an execution of the algorithm, when x happens to be between A 0 and A 1, we do one comparison here, a second comparison here, and then we're done. So the cost was two and indeed the length of this path is two. So it works out, no off-by-one errors.

All right, now exciting one for us, what we care about all the time in this class is worst-case running time. This is a feature of the entire algorithm. What is the worst-case running time of a given decision tree? Yeah.

AUDIENCE: The height of the root, the height of the tree.

PROFESSOR: The height of the root also called the height of the tree. Yep. Or the depth of the deepest leaf, whatever. So in this case all the leaves have the same level, but in general we care about the overall height. How many levels in this tree are there? It's the number of levels minus one, technically. But the length of longest root-to-leaf path is the definition of height. Here it's two.

In general we know for binary search it's lg n, but given an arbitrary decision tree, we just have to figure out what the height of the tree is and we'll figure out the worst-case running time. So this is why decision trees are interesting. Not because it means they're pretty I guess, but the reason they're going to be useful is we have this kind of hard question which is how much time do you need to solve a problem? And we're translating it into how low can you make your tree, which is a problem we know a lot about.

Trees are pretty simple. These are binary trees. They're rooted, and so we know lots of good things.

So let's prove some lower bounds. So I claim that for searching-- maybe I should

define the problem a little more formally-- I want to claim a lg n lower bound. So let's say for searching I have n preprocessed items. Then finding a given item among them in the comparison model, so all you're allowed to do are compare items and other stuff, but the only things you're allowed to do with the items is compare them.

Requires omega lg n comparisons in the worst case. It's kind of tedious to write down these theorems, but for our first lower bounds, I thought I'd be super explicit. I mentioned here that the items are preprocessed to mean you could do whatever you want the items ahead of time, that's for free. So I can sort them in particular, which lets me do binary search. I could build them into an AVL tree, could do lots of things, but no matter what I do to find another item takes lg n time.

Can someone tell me why? Who doesn't have the lecture notes right in front them, that would make it easy. This is a little more interesting, but we had all the tools at our disposal now. We want to show that this at least lg n. Why? Yeah.

AUDIENCE: [INAUDIBLE] have a no or yes, right? So it's--

PROFESSOR: Right.

AUDIENCE: --omega lg n tree.

PROFESSOR: OK. At each step, we only have a no or yes. That's a binary tree. So that makes you think lg n.

AUDIENCE: That's possible, it could be lg n.

PROFESSOR: Maximum actually could be arbitrarily large. You could do a linear search and the height will be m. We care about the minimum of course. Why does it have to be-- why does the height of a tree have to be at least lg n? There's one more thing we need to say. Yeah.

AUDIENCE: The tree has to contain all possible--

PROFESSOR: Because the tree has to contain all possible-- answers, let's say. Yeah, exactly. I think that's worth a pillow. See if I can do this-- oh! Ouch. I knew it was only a matter

of time. Sorry. I'll pay you later. Damages. At least I didn't hit a laptop or something.

All right, so decision tree is binary-- that was the first thing-- and it must have at least n leaves, one for each answer. At least. Now, at the leaf you have to know what the answer is, but there may be many leaves that have the same answer. That's possible. And indeed that will happen not for binary search but typical algorithm.

There's multiple paths to get the same answer, so there may be more leaves than n. And in fact, if you want to know this kind of thing, where x fits in this perspective, there's n plus 1 answers. If you want to know is it equal or is it strictly between two things there's 2n plus 1 answers. But in all cases, there's at least n answers and that's all I need.

In particular there's-- say x exactly matches one of the given items-- there's n items-- so you need to have at least n leaves. Maybe have more, I don't care. But it I have a binary tree with at least n leaves, the height has to be at least lg n. We're done. The height is the worst-case running time.

Super, super easy proof. So easy, it's never been taught in 006 before. But I think it's a good warm up for the next one which is sorting. Sorting is really the same thing. It's not going to be any harder except that it's a little more math but really not much more.

So now we know-- we just proved two useful facts-- one is that binary search is optimal in a comparison model, the other is that binary search trees are actually a good way to solve a problem. If your goal is to solve search and all you're allowed to do is comparisons, then you need lg n time. And so the search or next larger or next smaller, predecessor, successor, in binary search trees need to take at least lg n time. No matter how you do it, even if you don't use a tree.

So this justifies why binary search trees are interesting, because again the comparison model, that's the best you can hope to do. So that's comforting. That's why I like lower bounds and theoretical computer science in general because you

know when you're done, at least in a given model. Whenever-- we're never actually done, because we can always change the model. At least we understand the limitations of comparisons.

So for sorting, we claim a lower bound of n lg n. You've heard n lg n a zillion times. You probably know this is true, but now we actually get to prove that it's true. So we just follow the same strategy. Decision tree is binary. The question is how many leaves does it have to have?

So for sorting-- I didn't draw up an example-- I'm not going to draw an example of sorting because the trees get ginormous. Right? Because of the depth is n log n, the height is n log n, then there's binary branching everywhere. That's a lot of nodes. Two to the n lg n is big. More than two to the n even, so it's hard to draw a picture even for n equals 3. You can do it. People have done it. I don't want to. I'm lazy.

But the internal nodes look just the same. You're comparing two items, A I verses A J. I'll just draw the generic version. You have A I less than A J, question mark. And then you'll have a no and a yes.

So that's what a typical comparison looks like. Swaps don't appear here, because we're just looking at the comparisons. And then when you get down to a leaf, a leaf-- this is the interesting part-- the leaf will look like this. Well I took the original A 5 and that turned out to be the smallest element. Then-- maybe I'll write it this way-- then I have A 7, that turned out to be the next smallest element, then A 1 then A 0, whatever.

Hey, right at the end, somehow you know the sorted order and you can just write it down. We're not charging for this. We're only charging for comparisons. So however, maybe you've done swaps, in the end you know what the final order is and so you just write it down. And your goal is to make enough comparisons that you figure out what the sorted order is. We claim the number of comparisons here has to be at least n lg n.

OK, why? Because the decision tree is binary and the number of leaves has to be at least the number of possible answers. Could be more because each answer could appear in several leaves and it probably will in a typical sorting algorithm. And how many possible answers are there? Batter?

AUDIENCE: n factorial.

PROFESSOR: n factorial, number of permutations. This is a permutation of the input sequence and if all the items you're given are distinct, there will be n factorial permutations of them. So that's the worst case. So n factorial.

Now the tricky part is the algebra. Say, oh, well then the height is at least lg base 2 of n factorial-- lg base 2 because it's a binary tree. You can put a parentheses here if you want, they're not necessary. So now I want to claim that this is n lg n. How do I do that?

Maybe you just know? Yeah.

AUDIENCE: We can either use Stirling's approximation or we could write it out as a sum [INAUDIBLE]

PROFESSOR: Wow, cool. All right, you could either use Stirling's approximation or write it out as a sum. I've never done it with it sum. Let's do that, that sounds like fun. So, right? I like that because you know Stirling's-- it's like you've got to know Stirling and that's kind of annoying. What if you don't know Sterling?

But we all know the definition of factorial. I mean, we learned in grade school just because it's fun, right? Oh, I guess we-- I mean we did because we're geeks.

And then we know the lg of our product is the sum of the lg's. So this is lg n plus lg n minus 1 plus lg 2 plus lg 1. I think at this point it's easier to use summation notation, so sum of lg i. OK now we've got to do sum, this is 1 to n I guess. Now we need to know something about lg's, so it's not so easy.

It's easy to show-- I mean, certainly this is at most n lg n, but we need show that it's at least n lg n. That's a little trickier. I happen to know it's true. But I'd know it even in

the summation form because I know that lg-- lg looks like this basically, and so if you're adding up, you're taking the area under this curve right? Oh, look at these integrals. Oh, integrals. Brings back memories. This is discrete math class, though, continuous stuff.

So you're adding up all these numbers, right? This is lg i over all the i's and basically all of them have the same length. Like if you look at the last half, that would be one way to prove it. Ah, it's fun, haven't done summations in so long. Good stuff. [? IS042 ?] material but applied to algorithms and in algorithms it's fun because you could throw away constant factors and life is good.

We don't need exact answers really. You can find an exact answer, but let's say you look at the last half. Those are all going to be basically lg n. You can prove that.

So this is going to be at least the sum where i equals n over 2 n of lg i. Here I just throw away the first out of our two terms. And this is going to be at least sum i equals n over 2 to n of lg n over 2. Each of these terms is bigger than lg n over 2 so if I just say, well, they're all lg n over 2 that's going to give me something even smaller. Now the lg n over 2, that's just lg n minus 1.

I love this. It's going to give the right answer even. So that's an equals and so this equals n lg n minus n. That summation I can do. All the terms are the same, sorry n over 2. Not quite what I wanted. Close enough.

Sorry there is only n over 2 terms here, ignoring floors and ceilings. So I get n lg n divided by 2. This is omega n lg n because this n term is smaller than n lg n. So this one dominates. Doesn't matter if this one's negative, because it's smaller. This is omega n lg n. We're done. Sorting is omega n lg n. Very easy.

Who said summations? All right. Why don't you come collect a pillow, I'm not going to throw that far. Afterwards. OK.

That's one way to do it. Another way to do it, if you happen to know Stirling's formula for n factorial-- n factorial is about n over e to the n times square root of 2 pi

n. Right? If you do Taylor series approximation of n factorial, the first term, which is the most important term for us because as the asymptotically dominating term is square root of 2 pi n times n over e to the n. Hope I got that right. Yeah, clearly I've been studying.

You take lg's of that and you do the same thing of lg of a product is sum of the lg's and you end up with-- the right answer is actually n lg n minus order n. So I was off by a factor of 2 here. The linear term-- it does appear, but it's smaller than this and this is also omega n lg n. If you don't care about constants, it doesn't matter. If you care about constants, the constant is 1.

Kind of nice. Easy to prove a one half. And if you look at the lecture notes it works through that. But I think we've seen enough of that lower bound. And that's the end of our lower bound topic. Any questions on that?

So it's really easy. Once you set up this framework of comparison trees and now it becomes just a question of the height of a comparison tree. Comparison trees are binary. Just count how many leaves do you have to have, take lg of that and you get a lower bound of that.

AUDIENCE: What is meant by n preprocessed items?

PROFESSOR: Oh, yeah. For searching I was trying to be careful and say, well, if I have n preprocessed items.

AUDIENCE: [INAUDIBLE]

PROFESSOR: It means you can do whatever the heck you want. So here's the model. I give you n items. You can do all pairwise comparisons between those items for free and then I give you a new item and then I start charging for comparisons. So another way to say it is I only charge between for comparisons between x and the other items. And even then you need lg n.

AUDIENCE: [INAUDIBLE] case for sorting, right?

PROFESSOR: With sorting they were not preprocessed. Yeah, I didn't write the theorem. It's just

sorting and given items, no preprocessing.

AUDIENCE: What if there were preprocessing?

PROFESSOR: If they were preprocessed, you'd be done in zero comparisons. Yeah, exactly. This theorem is also true if I remove preprocessed, but in fact then you need n time. Unfortunately this proof technique will only prove a lower bound of log n, because even if these items were not preprocessed, then you have to do linear search basically.

So if you don't know anything about the items, you need linear time. But this proof will only prove a lower bound of log n. So this technique, while cool and simple, does not always give you the right answer. It just gives you a lower bound. May not be the right lower bound, may not be tight.

So searching always requires lg n time and what's interesting is it requires it even when you preprocess the items. Sorting, if you haven't preprocessed the items, then it takes n lg n. Clear? Good.

Now we get to the algorithms part of the lecture, always the most fun. The moments you've been waiting for. Let me erase comparison trees. Henceforth and I mean not only this lecture, but also the next three lectures which are about hashing, we will not be in the comparison model because for comparison model, we're done. We solved search, we solved sorting, n lg n three ways. I mean, how much more can we do?

So it's time to bump it up a notch, increase our model power. We've talked about the RAM in particular, Random Access Machine, where memory is in array, you can access anything in the array in constant time. We're going to use that power of the RAM to sort in linear time, sometimes.

A more appropriate title for this section of this lecture would be integer sorting. OK, so far we've been talking about comparison sorting where the items you're given--the only thing you know about them is that you can compare them in constant time.

But now we're going to think about the situation where the things that you're sorting are integers. That's a big assumption but it's a practical assumption a lot of the time. If you're not sorting integers you can map whatever the heck you're sorting into integers.

And usually it's already been done because you're representing it on a computer. You've already represented your thing is an integer of sorts. Bad pun. This is an assumption. So we assume-- going to be a little more precise. The keys you're sorting are integers.

There's still-- I'm going to put a little n here, remember there's n keys. I'm also going to assume that they're in some range. And for convenience, I'm going to assume that they're all non-negative-- it's not hard to deal with negative numbers, but it's just convenient to think about non-negative numbers. So if you start at zero, there's some maximum value, say k minus 1. So there's k different values they could be.

K could be anything. It's a parameter. We've always had n as a parameter, now we're going to also have k as a parameter. And just for completeness-- and each fits in a word. Remember the machine word of your RAM machine? Words were the things that we could manipulate in constant time.

Now this is a very reasonable assumption because we've been assuming so far you can compare two keys to items in constant time. To get that for integers, you need to assume that your integers are fitting in words. We usually don't state this assumption, but I thought I'd throw it in just for kicks. So we've got a bunch of integers, each one fits in a word. I could compare them, that takes constant time, or I could add them or subtract them or multiply them or divide them or do whatever the heck I want.

It turns out you can do a lot more than comparisons and it turns out this will help us. I don't know if I want to tell you the answer here. For k-- not too big-- you can sort in linear time. Believe it or not, this topic, integer sorting, is still a major area of research. People are still trying to solve this problem.

One conjecture is that even in all cases, you can sort in linear time given any integers that fit in words. This is not yet solved. Best algorithm is n times square root lg lg n with high probability. So it's almost-- almost n. It's a lot better than n lg n.

I'll just write that for fun case you can't parse in words. This is the best algorithms to date. I would conjecture you can do linear time in all situations. We're not going to cover this algorithm. That's a little beyond us. It's in advanced algorithms if you're interested, 6854.

But we're going to show that for a lot of cases of interest when k is not ginormous, it's really easy to sort in linear time. All right? And our first algorithm to achieve this is counting sort. Counting sort does not make any comparisons. It only does other stuff. And it's going to depend on n. It's going to depend on k. We'll get some running times not bad as long as k is not giant.

So as the name might suggest, what you do is count all the items. So imagine I give you a bunch of keys like 3, 5 7, 5, 5, 3, 6, whatever. I'd like to run through this array and say, ah, I see there are two 3's and there are three 5's and there's one 6, and one 7, so how do I sort it? I'd like to say, well 3 is smallest key and there's two of them, so I'll write two 3's, then there's three 5's so I'll write three 5's, and then there's a 6 and then there's a 7. That's the intuition.

Now how do I-- how I do that with an algorithm? Suggestions? Yeah?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah. Allocate an array of memory, which is my counters-- I'm going to count each k. I need an array of size k because there are k possible keys. Convenient those two terms start with the same letter. And then I'll just-- whenever I see an-- I'm going to run through the items in order, when I see an item, I say, OK, well that's key 3. I will look at index 3 of this array, increment that counter.

Then I see 5, increment that counter. I see 7, I see 5, I see 5, and by the end, I'll know that there are three 5's and two 3's and so on. That's how I count. And then how do output the items? You want to keep going?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah, just traverse the array of counters and the array is already written in order by key, so it's really easy. I mean I could draw this array for you if you like at 0, 1, 2 3. Here's the 3 position, it ends up with the value 2. And if I just go through, a lot of these will have 0's in them, just skip those. When I find a non-zero entry, just write-- oh, that means there's two 3's there, so I write 3, 3.

OK, that algorithm would work but I'm not going to even write it down, not even going to dignify it, because all it does is sort integers. But there's a subtlety here which we're going to need in a moment, which is why I stress it, that really we have n items. Each of them has a key, but it might have other stuff too that we'd like to bring along for the ride. We'll see why we care about that in a moment.

But it's also a typical situation, like you have a spreadsheet and you click sort by this column. Well every row has a whole bunch of data but you're only trying to sort by one of those fields, that one column. And that one field may be an integer, but there's all this other stuff you'd like to bring along.

And when you say, oh, there are two 3's, you know there's this 3, which you know maybe has a cloud around it. There's this 3 which maybe has a heart around it. That's about the limit of my drawing abilities. And now say, oh, there are two 3's but which 3? Should the cloud go first, should the heart go first?

I mean I don't care which goes first, maybe-- I do, actually-- but I will in a moment. That's another topic. But I'd like to bring that cloud somewhere. I want to put the cloud somewhere, want to put the heart somewhere.

All right, so here's a way to do all that. Basically the same algorithm but I'm just going to use lists. Still have an array of k things but no longer counters, now lists. They could be linked lists, they could be python lists. It won't matter for my purposes.

And then I'll say for j and range of n-- that'd be super pythonic here-- I want to look

at the list who's at index a of j and append a of j. And then the output is going to be an empty list initially. And then I iterate through the array their k values for that, and I just say output, extend, list i.

OK. This is counting sort or a version of counting sort. In your textbook, you'll find a different version which does not use lists, So it's probably more practical because it uses no data structures whatsoever except three arrays. But it runs in the same amount of time and this is a lot easier I think to think about. This is more modern perspective, if you well.

For every item, if you'd look at them in the given order of your array, you see what it's key value is. Maybe that's not exactly the same as the item, so it could be key of x is just x. But you know in python sort, for example, you're given a key function. So you take that key value. The key is guaranteed to be an integer between 0 and k minus 1, so you look at the list, that numbered list, and you just add this item to the list.

OK. But the item is not just the key, it's everything-- whatever that data structure is-- and then you just go through the list and you concatenate them. OK. How long does this take? How long does this step take? N? Nope. Constant? Nope. OK. Look at all the actions. It's order k time. To create an empty list takes constant time. They're k of them. OK?

How long does this step take, just the append? Constant? Good. Remember, append is constant time from the Python model or your favorite model, anything. We're assuming the key takes constant time because that's the word, so that's an assumption, but in the normal assumption.

So total time here is order n. And this thing, well this takes basically the length of Li time. And so when you add it up, maybe plus 1-- because to look at an empty list you still need to look at it-- so you add it up and you get order sum of all the Li's is all the items. And then you get plus 1 for each of them, so you get n plus k. n plus k is the running time of this algorithm.

Add those up. OK, so counting sort is order n plus k. So if k happens to be order n, this is linear time. But as soon as it's a little bit bigger, you're in trouble. So counting sort's a good warm up, but it's not ultimately what we want.

And a much cooler algorithm is called radix sort. It's going to use counting sort as the subroutine, which is why spent all this time on a mediocre algorithm. And it's going to get a much larger range of k and it will still be linear time. I'll tell you the answer.

K can be polynomial in n. So like if all your integers are between 0 and n to the 100, you can sort them in n lg n time. That's a lot bigger. It's not just like 10 n. I mean you could do 10 n here as well with counting sort.

And it's not just like n lg n, but they can go all the way to n to the 100, still be linear time. So that's what we're going to achieve. The idea of radix sort is simple. It's actually kind of the Excel spreadsheet approach. We're going to imagine we want to break each integer into a bunch of columns. How do we do that? Well, the way we normally write down numbers, except not necessarily in decimal, in some arbitrary base b. So I say, oh, an integer in base b. Well then there's the least significant digit and then the next one and the next one and the next one, some sequence of digits. And if I know that the maximum value is k, I know that the number of digits, which I'm going to call-- for each number which I'm going to call d, is just lg base b of k plus one, whatever.

We've got to be super precise here because if I'm in base b then that's what lg is, right? So normally we think of lg base 2 because we're writing things in binary. Computer scientists normally think that way. And fine, so now we decomposed our integer.

I'm not going to actually compute this base b representation, because it would take a long time. I'd have to spend n times lg k time to do that. I don't want to do that.

OK, but just imagine it that way. And then the algorithm as follows, sort the integers, all of them, by the least significant digit. Sort by the next least significant digit. Dot,

dot, dot, sort by the most significant digit. So there are d iterations here, for d digits. Sort all the integers by the least significant, all the integers by the next, and so on.

It's like in your-- this is a useful technique in Excel, if you want to sort by several columns-- or your favorite spreadsheet, doesn't have to be Excel, sorry-- you click on the least significant column first, and then click on all the other columns in increasing order, you will sort by all of them, it turns out. It's kind of magical that this works.

I don't have a ton of time for an example. Let me first analyze the algorithm. We'll see if we have time for an example. So there are d digits-- oh, and I'm going to sort each of these sorts of using counting sort. This is I guess sort by digit using counting sort.

So how long does it take to sort using counting sort in this setting? Normally, it's n plus k. Here it is, n plus b. Good. Because all of our digits are between 0 and b minus 1. So we're just sorting by digit.

Now here is where we're using this idea of a key. When we say key, I wanted our integers. What we do is compute the digit we care about. So if we're in this step, the key function will be compute the least significant digit, which is like taking it mod b to compute the most significant digits like dividing by b to the power of d minus 1 or so.

OK but it's a constant. You do one divide and one mod, the constant number of operations. You can extract the digit in constant time. So the key function is constant time and so this works. We don't have to actually write them all down, just compute them as we need them.

Cool I guess we could compute them ahead of time. It's not a big deal. Fine. So that's each digit. So the total time is just that times d because we have d steps. So it's n plus b times d. Now d was that lg thing, lg base b of n. I have this b. What should be b? You gotta love the English language.

What should I choose b to be, or not to be? That's the question. Any suggestions? I want to minimize this, right? I want minimum running time. So I'd like b to kind of

large to make this base large. Sorry, this is not n, this is k. I copied that wrong out of excitement. Just copying this over.

OK, I'd like b to be large, but I don't want it to be so large because I don't want it to be bigger than n so what should I set b to be?

AUDIENCE: N.

PROFESSOR: N, good choice. It's a good trick whenever you have a sum of things you want to minimize, usually it's when they're equal. Occasionally it's the extreme like when b is 0 or something. B as 0 not a good plan. Base 0 is pretty slow.

So if I set-- I'll write it here-- you can prove it with a derivative or whatever. This is going to be minimized when b is-- I'll be vague-- theta n. So then it's going to come out to n times lg base n of k. And lo and behold, when k is polynomial in n, it's k to some constant, then that will be linear.

So let me write that. If k equals n to the c, or say is at most n to the c, then this is going to be order n times c. So if your integers are reasonably small, you get a linear time sorting algorithm. And reasonably small means polynomial in n, in value. That's kind of cool. That's radix sort. And we're out of time. There's an example in the textbook or in the notes how this works. You could prove it by a simple induction.