[SQUEAKING]

[RUSTLING]

[CLICKING]

JASON KU: Good morning, everybody. How's everybody doing? Nice long weekend we just came from-- I'm doing well. I'm actually getting over a little cold. Aw-- yeah, unfortunately. But after this, I don't have anything else this week, so that's good. OK, so last time, last week, we talked about how-- we looked at the search problem that we talked about earlier that week and showed that, in a certain model of computation, where I could only compare two objects that I'm storing in my-- that I'm storing and get some constant number of outputs on what I could-- how I could identify these things, like equal, or less than, or something like that, then we drew a decision tree and we got this bound that, if I had n outputs, I would require my decision tree to be at least log n height.

And so in this model, I can't find the things faster than log n time. But luckily, we are in a model of computation which has a stronger operation-- namely, random accessing. And if we stored the things that we're looking for, we have unique keys, and those keys are integers. Then, if I have an item with key K, if I store it at index K in my array, then I can find it and manipulate it in constant time. That's pretty cool.

That's what we called a direct access array. A direct access array-- really not different than a regular array, except how are you using it when we were talking about sequences is we are giving extrinsic semantics to the slots where we are storing these things. Basically, I could put any item in any slot. Where it was in my array had nothing to do with what those things were.

Here we are imposing intrinsic semantics on my array that, if I have an item with key K, it must be at index K. That's the thing that we're taking advantage of here. And then we can use this nice, powerful linear branching random access operation to find that thing in constant time, because that's our model of computation.

OK, then what was the problem with this direct access array? Anyone shout it out. Space-- right. So we had to instantiate a direct access array that was the size of the space of our keys. In general, my index location is-- could go from 0 to some positive number. If I a very large positive numbers, if I was sorting-- if I was searching among your MIT IDs, I'd have to have a direct access array that was that spanned that space of possible keys you could have.

And that could be much larger than n. And so the rest of the time we talked about how to fix that space problem. We can reduce the space by taking that larger key space from 0 to u, which could be very large, and map it down to a small space. Now, in general, if I give you a fixed hash function there, that's not going to be good in-- for all inputs.

If your inputs are very well distributed over the key space, then it is good, but in general, there would be hash functions with some inputs that will be bad. That's what we argued. And so for the rest of the time there, we talked about hash families, choosing a hash function randomly from among a large set of hash functions, which had a property that, if I chose this thing randomly and you, generating your input, didn't know which random numbers I was picking, the expectation over my random choice-- me-- I'm the one running the algorithm, not you giving me the input-- that random choice-- my algorithm actually behaves really well in expectation.

In particular, I got constant time for finding, inserting, and deleting into this data structure, in expectation. We did a little proof of-- that the chain links where we stored collisions in our hash function-- in our hash table-- sorry-- those wouldn't be very long, and so if they were constant, then I don't have to search more than a constant number of things when I go to an-- a hashed index location.

Does everyone remember what we talked about last week? I didn't show you this chart at the end, but I'm showing it to you now. Essentially, what we had was we have a bunch of different ways to deal with this set interface. And last week, we talked about the sorted array, and then we talked about this direct access array and this hash table, which do better for these dictionary-- the find, and insert, and delete operations-- or at least better in an expected sense.

What's the worst case performance of a hash table? If I have to look up something in a hash table, and I happen to choose a bad hash table-- hash function, what's the worst case here? What? n, right? It's worse than a sorted array, because potentially, I hashed everything that I was storing to the same index in my hash table, and to be able to distinguish between them, I can't do anything more than a linear search.

I could store another set's data structure as my chain and do better that way. That's actually how Java does it. They store a data structure we're going to be talking about next week as the chains so that they can get, worst case, log n. But in general, that hash table is only good if we're allowing-- OK, I want this to be expected good, but in the worst case, if I really need that operation to be worst case-- I really can't afford linear time ever for an operation of that kind-- then I don't want to use a hash table.

And so on your p set 2, everything we ask you for is worst case, so probably, you don't want to be using hash tables. OK? Yes?

AUDIENCE: What does the subscript e mean?

JASON KU: What does the subscript e mean? That's great. In this chart, I put a subscript on this is an expected runtime, or an A meaning this is an amortized runtime. At the end, we talked about how, if we had too many things in our hash table, then, as long as we didn't do it too often-- this is a little hand wavey argument, but the same kinds of ideas as the dynamic array-- if, whenever we got a linear-- we are more than a linear factor away from where we are trying-- basically, the fill factor we were trying to be, then we could just completely rebuild the hash table with the new hash function randomly chosen from our hash table with a new size, and we could get amortized bounds.

And so that's what Python-- how Python implements dictionaries, or sets, or even objects when it's trying to map keys to different things. So that's hash tables. That's great. The key thing here is, well, actually, if your range of keys is small, or if you as a programmer have the ability to choose the keys that you identify your objects with, you can actually choose that range to be small, to be linear, to be small with respect to your items.

And you don't need a hash table. You can just use a direct access array, because if you know your key space is small, that's great. So a lot of C programmers probably would like to do something like that, because they don't have access to-- maybe C++ programmers would have access to their hash table. Any questions on this stuff before we move on? Yeah?

AUDIENCE: So why is [INAUDIBLE]?

JASON KU: Why is it expected? When I'm building, I could insert-- I'm inserting these things from x 1 by 1 into my hash table. Each of those insert operations-- I'm looking up to see whether that-- an item with that key already exists in my hash table. And so I have to look down the chain to see where it is.

However, if I happen to know that all of my keys are unique in my input, all the items I'm trying to store are unique, then I don't have to do that check and I can get worst case linear time. Does that make sense? All right. It's a subtlety, but that's a great question.

OK, so today, instead of talking about searching, we're talking about sorting. Last week, we saw a few ways to do sort. Some of them were quadratic-- insertion sort and selection sort-- and then we had one that was n log n. And this thing, n log n, seemed pretty good, but can I do better? Can I do better?

Well, what we're going to show at the beginning of this class is, in this comparison model, no. n log n is optimal. And we're going to go through the exact same line of reasoning that we had last week. So in the comparison model, what did we use when we were trying to make this argument that any comparison model algorithm was going to take at least log n time?

What we did was we said, OK, I can think of any model in the comparison model-- any algorithm in the comparison model as kind of this-- some comparisons happen. They branch in a binary sense, but you could have it generalized to any constant branching factor. But for our purposes, binary's fine.

And what we said was that there were at least n outputs-- really n plus 1, but-- at least order n outputs. And we showed that-- or we argued to you that the height of this tree had to be at least log n-- log the number of leaves. It had to be at least log the number of leaves. That was the height of the decision tree.

And if this decision tree represented a search algorithm, I had to walk down and perform these comparisons in order, reach a leaf where I would output something. If the minimum height of any binary tree on a linear number of leaves is log n, then any algorithm in the comparison model also has to take log n time, because it has to do that many comparisons to differentiate between all possible outputs. Does that make sense?

All right. So in the sort problem, how many possible outputs are there? What is the output of a sorting algorithm?

AUDIENCE: [INAUDIBLE]

JASON KU: What? What's up? A list-- in particular, given my input-- some set of items A that has size n-- what I'm going to give you is some permutation of that list. So for each index, say, I could tell you where it goes.

Another way I could say is, where does the first item go to, where does the second item go to, where does the third item go to-- blah, blah, blah-- like that. So how many different choices of a permutation are there? Well, how many choices do I have for the first thing of where it could be in the final sorted array? It could be in any of the places, so it's n.

How about this one, the second one? Well, it can't go to where this one went, right but it can go anywhere else. So it's n minus 1. And since these are independent choices I'm making, if I multiply them all together, I get 9 factorial permutations that are the number of possible outputs that I have to my sorting algorithm. So for me, to have an output to my sorting algorithm be correct, I need at least n factorial leaves. Does that make sense? OK.

The nice thing about doing this last week is this is really just the number of leaves and this is really the number of leaves. So what's the number of leaves is theta n factorial. Here it's actually n factorial, but I'm just going to put it there. And here we get an n factorial.

I see. So it's at least omega n factorial. Does that make you happier? Theta here-- thank you-- has to be at least. So this was right. OK, so at least this many-- there are algorithms that, if it got-- it could take two different routes to get to the same output. So this is a lower bound on the number of leaves. OK?

So what this argument is saying is that, if I just replace the number of leaves n here with n factorial, I get a similar comparison sort lower bound now. So what is log of n factorial? This is familiar from p set 1 maybe. So one thing I could do is I could put in Sterling formula, right? And that'll give me something of the form n log n.

But what's another way I could lower bound n factorial? Well, I have a bunch of things here. That's n factorial. Half of these things-- these half, n/2 things-- are bigger than or equal to n/2. That make sense? So I can certainly lower bound this thing by n/2 to the n/2. That's a little easier thing to take a log of. If you take a log of that, that's asymptotically n log n.

So what we're getting here is any sorting algorithm here takes at least n log n comparisons, and so a merge sort's the best we can do. That make sense to everybody? We're just piggybacking on the analysis we had about decision trees, connecting leaves with the minimum height of any binary tree on that number of leaves, and just replacing n with n factorial-- nothing super interesting here. Yeah?

AUDIENCE: [INAUDIBLE] the n over 2.

JASON KU: Yeah, sure. You can just plug in Sterling formula, but I did this, so I might as well clarify. There are n terms here in the product. Half of them are at least n/2. Does that make sense? I can lower bound this product by something smaller than half of the terms-- a product of that, and that'll be fine.

So I'm taking n/2 of them and I'm multiplying n/2 altogether, n/2 times. Does that make sense? It's just providing a lower bound. I just need something that's smaller than all of these terms. And multiply them all together, and that'll give me a lower bound.

OK, so we can't do better than n log n in the comparison model, but what we did last week was use random access and a direct access array to do better. OK? Can anyone think of how to use that idea to sort faster? And I'm going to give you a caveat here. I'm going to let you assume that the keys of the things you're trying to sort out are unique.

And say they're in a bound-- in a small range. So how could I use a direct access array to sort faster? Any ideas? Yeah?

AUDIENCE: Could you just literally insert [INAUDIBLE] into a direct access array?

JASON KU: Uh-huh.

AUDIENCE: And then you look at that array and how to sort it.

JASON KU: OK. So what your colleague is saying is exactly correct. It's something that I like to call direct access array sort. We won't really call it that, because there's something more general that we'll talk about in just a second. But what your colleague was saying is, instantiate a big direct access array-- direct access array sort.

I'm instantiating this big direct access array of the space of my keys, and what your colleague was saying was I take each one of the items in my-- the things that I'm trying to sort, I look at each one of their keys, and I stick it in the direct accessory exactly where it needs to go, in constant time. That's great. Now, I gave you this caveat that all the keys were unique, so I don't have to deal with collisions here.

But then, after I'm done with this, all of these things are now in sorted order, and what I can do is I can just walk down this list. A lot of these cells are empty, potentially. Some of the keys might not be there, but what I can do is just walk down this list, pick off every item that does exist, stick them in an array-- I'm done.

Stick a key into here and then-- all right. Make direct access array. Store items-- item x in index x.key. Walk down direct access array, and return items seen in order. Does that make sense to everybody?

All right, how long does this step take? Building a direct access array order u-- OK, so this is order u-- how long does this take? How many items you have to insert? Order n, or just n-- and how long does it take to insert each one of these things into my direct access array? Worst case constant time-- so this is n times worst case constant time-- great.

How long does this last one take? Anyone? O of u also-- right, because I'm walking down the entire length of u. So this algorithm takes, in total, n plus u time. This is great. u is bigger than n, because we assumed distinct keys. But if u is on the order of n, then we now have linear time sorting algorithm. Yes? What's up?

AUDIENCE: [INAUDIBLE]

JASON KU: I'm sorry. You have to speak up.

AUDIENCE: How do you attach keys to the [INAUDIBLE]?

JASON KU: How do I attach keys to my inputs in my-- for a set data structure that we've been talking about, all of my items have keys. That's just something that we impose on our input.

AUDIENCE: [INAUDIBLE]

JASON KU: Each of the keys is-- in this case, it has to be a number. That's a nice point. We do this to talk about sorting items generally so that we don't have to deal with potentially if these keys have values associated with-- or other stuff associated-- put them on that item, and they'll still be there.

But in general, if you just wanted to sort integers, you could say that .key is-- points back to the object itself, if you want to just sort some integers. Does that make sense? It's a good question, though. OK, so that gives us a linear time algorithm when u is small, and under this condition that I have unique keys when I want to sort.

Those are fairly restrictive, so we might want to generalize this a little bit. OK? So that's direct access array sort. What if we had a set of keys that was a little larger? So let's say u is theta n implies linear time sorting. That's great.

So now, what happens if we expand that range a little bit? Say u is less than or equal to n squared-- maybe just less than. OK, this is a bigger range And if we instantiated a direct access array of quadratics size, we'd have a quadratic time algorithm. This is not helpful.

Anyone have a way in which we could sort integers that are between 0 and n squared? Maybe using the stuff that we had above-- Yeah?

AUDIENCE: [INAUDIBLE] sort by the first n, kind of like the first digit.

JASON KU: Your colleague is saying exactly the thing that I'm looking for, which is great, which is maybe we could break this larger number into two smaller numbers. Any integer that is between 0 n squared can be written as-- key can be some a and b, where a is essentially the higher n and b is the lower n. This is kind of weird.

OK, so what do I actually mean by this? I mean that let's let a be K, when I divide it by n-- integer, the floor-- key integer to divide by n. And b equals K mod n. So this is a number that's less than n and this is a number that's less than n. Does that make sense?

And actually, I can recover K at any time by saying K equals an plus b. I've essentially decomposed this into a base n representation of this number. And I have two digits in that number. This is the n-th-- n digit, and this is the ones digit. Does that make sense?

All right, so now let's say I have this list of numbers-- 17, 3, 24, 22, 12. Here I have five numbers. So what's n in this case? 5-- OK, not so interesting. n is 5 here. And I'm going to represent this as five pairs of numbers that are each within the bounds of 0 to 4. Does that makes sense?

So what is my a, b representation of 17? 3, 2-- OK. Yeah, so there are 3 times 5 plus 2. That's good. That's 17. Yeah? I think your colleague did that, right? I have all of these written down, so I'm just going to write it out. And I hope I did it correctly.

OK-- 3, 2; 0, 3; 4, 4; 4, 2; 2, 2-- OK. So now I have a bunch of things that I want to sort based on this function that I have. These are no longer just integers that I need to sort. I need to sort by this transformation of this thing into a number. Does that make sense?

So anyone have any ideas on how we could-- by the way, these are both constant time operations on your computer, as long as it's an integer division and this is mod. Python also has a nice thing, I think, in its standard operations, which is divmod of K, n. Is that right? Yeah. So if you want to use that, you can.

OK, so how do we sort these tuples? These are tuples, right? You guys are, I'm sure, very familiar with tuples by now. How do I sort these tuples? What's the most important digit of this thing? If I had to sort one of the digits and get something that's close to sorted, what's more important-- the 1's digit or the n's digit?

OK, we have discrepancy here. Who says 1? Who says n? Someone who said n tell me why. Oh, you all think that way for no reason.

AUDIENCE: [INAUDIBLE]

JASON KU: Yeah. Sorry. This is a little confusing. This is the 1's digit. This is the n's digit. This is the n's digit. This is the 1's digit in how I'm writing this. Does that makes sense? Yeah?

AUDIENCE: [INAUDIBLE] have a different ones digit inside of it. So you could have [INAUDIBLE] but that only tells you where they are with regard to the specific n category they're in. So it's more of a [INAUDIBLE].

JASON KU: Yeah. So what your colleague is saying is exactly correct. I could vary b all I want right with the same a. If I change a by 1, it doesn't matter what b is-- it's going to be bigger. Does that make sense? The K is much more sensitive to a than it is to b, so a is more important than b. Does that make sense?

So if I just wanted to get some linear time algorithm, I could just sort by their bigger digits and hope they don't differ very much on the smaller things. I've kind of sorted these things. Does that make sense? OK. What if I actually want to sort these things? Any hints? Yeah?

I need to sort on both, in some sense. What I'm going to tell you right now is an algorithm that I like to call tuple sort, but you can also think of it as Excel spreadsheets sort. I have an Excel spreadsheet of a bunch of data. I have a prioritization on how important the keys are to me-- the columns.

And if I have a very important column and an order of the columns of how important they are to me, I can repeatedly sought on the columns until they're sorted based on my preference. That's something that you may have done. Now, if I have an ordering on the preferences of my columns, do I start by sorting all of them on the most important thing or the least important thing? What?

Who says most? Who says least? There's discrepancy here. All right, let's try it out. All right, tuple sort-- let's start by sorting these things by least significant first, and then-- no, most significant first and then least significant. That was the first thing I asked you, right?

All right, so these are the most significant things, the first ones. And these are the less significant things. All right, instead of writing it as tuples, I'm going to write them as 32, 03, 44, 42, 22. Is everyone cool that? This is just base five representation.

All right, so let's start by sorting all of these things by the most significant thing, which is by this guy, this guy, this guy, this guy, and this guy. So how do I do it? The first one is 03, second one is 22, the next one is 32, 42, and then 44-- maybe 44? I don't know. Does it matter, the order in which I put these things? I don't know. I'm just going to keep it the same order for now.

All right, so I've sorted it by the least significant-- or the most significant-- sorry-- the leading term. And now I'm going to sort by the least significant. So what's the least significant here? 22-- then 2 is also-- this is also 2. This is also 2. This is 3. And sorted list-- voila. Why did that not work? Yeah?

AUDIENCE: [INAUDIBLE]

JASON KU: Yeah. So what happened is I did take into account the significant digit sort, but when I did the less significant thing, it erased all of my work from up here. Does that make sense? In the case of ties, we want the more significant thing to take precedence, so we want to do that thing last. Does that makes sense?

So the right way to do this-- this is the most significant first [INAUDIBLE] not good. All right, at least significant first-- let's try that. So least significant here is 2. OK, so I see 32, 42, 22, 03, and then 44. OK? Sound good? Least significant first-- now I do most significant. I sort the most significant thing.

OK, so what's the most significant thing? 03, 22, 32-- most significant four-- 44, and 42-- cool. We're sorted, right? I did what you told me to do. I sorted by the most significant thing. What's the problem here? What did I do wrong?

You wanted me to put 42 here and 44 here, right? Because 42 came first in the input and 44 came second, right? OK, if a sorting algorithm maintains this property that, if they are the same thing, then the output maintains their order from the input to the output-- their relative order-- that's what we call a stable sorting algorithm.

And so if we have a stable sorting algorithm when we're doing tuple sort, when we're sorting on different keys or columns of a set, we really want to be using a stable sorting algorithm. Does that makes sense? Because otherwise, we may mess up work we did before in a previous sort of the less significant things. And so yes, we want a stable sorting algorithm here, because then we will end up sorting our thing. Does that make sense? Yes?

AUDIENCE: [INAUDIBLE]

JASON KU: So what your colleague is saying-- let's sort by most significant, then look at all of the things with one of those that are the same, and now sort that. That's something we could do. How long would that take? Well, let's say I didn't use half of my more significant set of digits. Say I'm only using n/2 or-- that's not quite going to get what I want.

AUDIENCE: [INAUDIBLE]

JASON KU: Say again.

AUDIENCE: We'll take n squared [INAUDIBLE].

JASON KU: Yeah. So what we're going to do, if we have direct access array sort-- if I then go into each one of these digits and try to sort the things that are in there, that's going to take time. It's going to take time for each of those digits. There might be a ton of collisions into one of the things, and so I might take more time to sort that than linear. Does that make sense?

So I would prefer to do this tuple sort kind of behavior, sorting the smaller thing, sorting the bigger thing. And because I only have a constant number of things in my tuples, this is important, because I only have two things I'm worried about here. I only have to do two passes of a sorting algorithm to be able to sort these numbers.

However, can I use direct access array sort here? What was the initial stipulation I had on direct access array? That the keys were unique-- that's exactly the opposite of what we have here. We have things that could be the same. So we give up-- can't do it. What do we do instead? Yeah?

AUDIENCE: [INAUDIBLE]

JASON KU: You've already said the thing that I'm looking for, so that's great. Your colleague said, why can't we just put more things at a key? Why can't we put a list there? That's exactly what we do. This is called counting sort. And what we do here is we still have this direct access array of space u minus 0 to u minus 1, but instead of storing one thing here at each key K, we store a pointer to a chain.

This sounds like hashing, right? But the important thing is that I need to make sure, as I'm inserting things in here, that I'm maintaining the order in which they came in. I can't just throw them willy nilly, or else we have this problem up here that we had before.

So I need what I would say is sequence data structure, something that will maintain the order that I-- the extrinsic order that I had when I'm putting these things in. So as I have multiple things with K, I'm going to put them in the order. I can put-- have a pointer to a dynamic array or a linked list, where I just add things to the end.

And then, at the end of my algorithm, when I read off the things, I can just look at anyone that has a non-empty data structure under here and read them off in the order that they came. Does that makes sense? So for this example, I'm just going to do this last step here from the first row to the second row.

I'm going to have this direct access array with 0, 1, 2, 3, 4 on the slots. So how am I going to do this counting sort now? I have 32, 42, 22, 03, and 44. I can take the first one, 32. I'm sorting by the most significant thing. I stick it here-- 32, and then 44-- 42-- sorry-- 42, 22.

This is not so much different yet then dynamic array-- direct access array sort. But when we get to this duplicate, 44 here, we now have two things in this thing. And because we are keeping them in order in this sequence, I'm appending to the end. Then, when I go and read off the different things, then I'm returning them in a stable way in the way that I want them to be. Does that makes sense. And it's not overwriting the work I did on the lower significant digits.

So how long does this take? This also only takes order n plus u, because I'm instantiating this thing of size u. And then, how big are these data structures? Well, maybe I'm storing one, a constant amount for each index. So that's a u overhead. And then I'm paying 1 for every item I'm storing.

These things are only the lengths. The sum total of their lengths is n, because I'm only storing n things in there. So the total amount of space, the total amount of work I have to do is order-- I need to be able to spend in constant time and I need to be able to cycle through these things, iterate over them in linear time. But if I have that, I get n plus u. Yeah?

AUDIENCE: How do you ensure that, within your linked list or your dynamic-- those elements, like four equals four-- how do you make sure that those are sorted?

JASON KU: So your colleague is saying, how do I ensure that the things in these lists, where they collide, how do you ensure that they're sorted? I don't. I just ensure that they came in the order that they came. But as long as I sorted the lower order digits correctly in the previous things, then I'm assuming that their order as they come in will be sorted, if they collide.

That's the assumption. That's the reason why I'm doing these building up from the least significant to the most significant is so that I know that, when they collide, the underlying stuff there is sorted already in the input. Does that make sense? Great-- yeah?

AUDIENCE: So this array isn't as big as u. It's as big as n.

JASON KU: I'm using a direct access array on the keys-- oh, this is n. So counting sort is general for any u. I just happened to pick u being n in this case when I broke this thing up into n squared. But this general concept is-- doesn't matter what I choose for u. Does that make sense?

OK. But we will use that right now to sort larger ranges of numbers. This was exactly the idea. We're going to combine tuple sort, use counting sort as its auxiliary sorting-- stable sorting algorithm to do all its work on these digits. And so to sort of on n squared size numbers, I get linear time, which is great, because u is n in this case.

But can I extend that? What if I had n cubed? What if I had up to size u equals n cubed, or less than n cubed? How many digits would I have there? How many size n digits what I need to represent a number of size n cubed? Any ideas?

What did we do here? We divided off an n. We took it and stored it. We're left with something of size n. If I had a number of size n cubed, I could divide off an n. I'm left with something of n squared. I don't know how to deal with something of n squared. Actually, I do. I can split it up into two size n numbers.

So if I had numbers bound-- upper bounded by a cubic-- n cubed-- I could split it up into three digits. Three is still constant. And so I could split it up into three digits, tuple sort them in their increasing priority, and sort those. Again, I'm doing linear work per digit. I have a constant number of digits, so I get a linear time algorithm. Yeah?

AUDIENCE: When it comes to sorting [INAUDIBLE]--

JASON KU: Uh-huh.

AUDIENCE: Are you ensuring that that runtime is also big O of n plus u?

JASON KU: Yeah. So it's always going to be big O of n plus u, but because I'm bounding my digit size to be n, u is n there, and so I'm getting linear time. Does that make sense? Yeah. So the idea here-- this is what we call radix sort. Radix sort-- break up integers, max size u, into a base and tuple.

So basically, each one of my digits can range from 0 to n. How many base n digits do if I have a number of size u? Yeah, log n of u-- number of digits is log n of u-- log base n of u. And then tuple sort on digits using counting sort, from least to most significant-- that's the algorithm.

How long does that take? How long does it take to sort on a digit that spans the key 0 to n? Linear time, right? Order n time-- how many times do I have to do this tuple sort? The number of digits times, right? So the running time of this algorithm-- first, I have to do this stuff, break up each of the integers. That takes n time-- n times the number of digits.

I had to create each one of these tuples-- so n plus n times the number of digits-- log base n of u. So here I had to loop through all the things. And then here, for each thing, I broke it up into log base n of u digits, and that's how long the first thing took. And then, how long did it take me to tuple sort? n time per digit-- so I also get this factor. Does that make sense?

How long is that? Is that good? Is that bad? For what values of u is this linear time? If u is less than n to the c for some constant c, then the c comes out of the logarithm, log n of n is 1, and we get a linear time algorithm. Does that makes sense? OK. So that's how we can sort in linear time, if our things are only polynomially large.

So in counting sort, we get n plus u. In radix sort, we get also a stable sorting algorithm where the running time is n plus n times log base n of u. Does that makes sense? And then, in the situations where-- there's a typo there in counting sort-- that should be when u is order n-- counting short runs in linear time.

And it's linear time also in the case of rating sort, if our things are bounded by a polynomial in n, by n to the c for some constant c. Does that make sense? All right, so that's how to sort in linear time, with the caveat that your numbers aren't too big. OK, see you next week.