VICTOR COSTAN: Any questions about the sorting methods that you want me to go over in that while I revise? OK. All right, sorting. What sorting methods have we learned? Let's start from dumbest to smartest.

AUDIENCE: Merge sorting.

VICTOR COSTAN: OK, somewhere in the middle. Merge sort isn't very bad. What's the easiest method to sort?

AUDIENCE: Insertion.

VICTOR COSTAN: Insertion sort. Excellent. All right. What else? Heapsort. And? I gave two away now.

AUDIENCE: Counting.

VICTOR COSTAN: Counting sort. Very good. And? Oh, wow. If you don't even have the name of it. So the last one is radix sort. What are the running times for these three that you guys remember?

AUDIENCE: Insertion sort is linearly one more. It's bad.

VICTOR COSTAN: I want to see our pseudocode for insertion sorts.

AUDIENCE: n squared.

AUDIENCE: Now that's really bad.

VICTOR COSTAN: So linear is as good as you could possibly get. So sorting takes an array of random stuff and outputs an array of things in a sorted order. The array is size n, so it has to output an array of size n. If you can do an algorithm that runs in order n time, then that's the best you could possibly accomplish, because you have output n elements. So the best possible time you could get for sorting is theta of n.

All right. How about merge sort?

AUDIENCE: [INAUDIBLE].

VICTOR COSTAN: Thank you. Heapsort.

AUDIENCE: Order h. Order h is log n.

VICTOR COSTAN: Order h where h is log n. OK. And you're missing a factor. So a heap operation takes order h, which is log n. So if I have to insert a numbering in a heap or extract a number from a heap, that's log n. In order to start an array, how many insertions do I do?

AUDIENCE: I think-- now I don't know.

VICTOR COSTAN: OK. Wild guess.

AUDIENCE: n.

VICTOR COSTAN: Very good. See, there you go. So you need to insert all your numbers in a heap and then extract them one by one. And you will get them in the correct order that gives you the sorted results. So n log n.

Does anyone remember what's special about these three sorting methods that does not apply to the other two?

AUDIENCE: They're in place.

VICTOR COSTAN: Merge sort isn't quite in place. If it would be in place, it would be perfect. There is actually a way of making in place merge sort, but it requires a PhD degree to understand that. So we will not cover it in 6006, because I do not understand it. So I couldn't explain it.

So merge sort is not quite in place. Which one is in place?

AUDIENCE: Heapsort.

VICTOR COSTAN: Good. So heapsort is in place. Merge sort is not in place. And insertion sort is really slow, so we don't care that much about it. So what's special about these three that does not apply to these two?

AUDIENCE: You don't have to use integers.

VICTOR COSTAN: OK. You don't have to use integers. What do they want to know instead about the things you use? So we'll call them keys.

AUDIENCE: You need to be able to compare them.

VICTOR COSTAN: All right.

AUDIENCE: You don't need to have a minimum and a maximum integer.

VICTOR COSTAN: So turns out, if you have a comparison operator, you will have a minimum and a maximum. But that's complex abstract algebra that we don't need to worry about. So you gave me the good answer, which is we use something called a comparison model. And in that model, you do not need to know too much about your keys. So the elements in the area that you're sorting.

Your keys are blobs. And all they have to be able to do is know-- if you have two of them-- you have to know which one's greater. That's it. Nothing else. What's the problem with the comparison model?

AUDIENCE: It takes time to compare things. It's like with everything.

VICTOR COSTAN: Yeah. So we learned in lecture that there is a lower bound for the comparison model. And if you want to sort using nothing but this information, that will take you at least n log n time. You cannot do better than n log n if all you're using is comparisons.

So in that respect, merge sort and heap sort are optimal. If you want to stay within this model, this is the best time you're going to get. Does anyone know how you can implement this comparison model in Python?

So numbers respond to these operators, right? Actually, in Python this is equals equals. What if I have a random object and I want to make it respond to these operators?

So for example, I write merge sort. We wrote merge sort. And now I have my own objects, my own keys which are not necessarily integers, because that's why we like this. And we want to make them respond to these operators. So I can call merge sort on an array of them and it will crash. What do I have to do?

AUDIENCE: I mean, you have to give the keys values that can be compared.

VICTOR COSTAN: So suppose this is my key class.

AUDIENCE: This is lad, the lt, and gt.

VICTOR COSTAN: All right. There's a magical method in Python. So there is the old school model, which you might see in legacy code, which only works in Python 2.x, which is you define the method called cmp that takes self and other. And it has to return a number that's either smaller than zero, equal to zero, or greater than zero. And this maps to this.

So you'll see this in old code. But you shouldn't use it in new code. On this, you have a very good reason to.

Instead, the new model says that you define special methods called lt, which stands for less than. So it's this guy. le, which is less or equal. gt, which is greater than. And ge, which is greater or equal.

And if you look at our code for pieces two and three, we have some objects that pretend they're keys. And we have to define these methods. Also, when you define these, it's a good idea to define eq for equality comparison. And ne, which is this guy.

So these also take self and other key that you're comparing with. And they return true or false. So this will help you understand the code better.

All right, so with relatively little work, you can have any wild object you want act as a key. And then you have insertion sort, merge sort, heapsort, heaps, binary trees, AVLs. Everything works, because everything uses the comparison model.

The problem is this n log n bound. It's not as fast as the best possible sorting algorithm you could come up with. This is slower than this.

So that's why we have to break out of the comparison model. And we have to look into these boxes and get more information, so that we can write faster sorting algorithms. Does anyone remember the running time for counting sort?

AUDIENCE: [INAUDIBLE] again?

VICTOR COSTAN: OK.

AUDIENCE: n plus e.

VICTOR COSTAN: OK. Let's remember how counting sort looks like. Let's get this array that-- that should be enough-- four, one, three, two, three. How do we sort it using counting sort?

AUDIENCE: We initialize an array of all the possible values.

VICTOR COSTAN: Very good. Very good. So counting sort needs to know something about your values, right? It makes an assumption.

And the assumption is that these values are integers from 0 to, say, k minus 1. So you have k possible values. And they don't really have to be these as long as you can map them to these numbers.

So we are going to initialize an array. Let's say this is an array. And zero, one. So zero, one, two, three, four, five. So we're going to initialize it with--

AUDIENCE: Oh, zeroes.

VICTOR COSTAN: All right. And then?

AUDIENCE: Iterative over our list sort incrementing the corresponding value to each key in your--

VICTOR COSTAN: So which one am I incrementing here?

AUDIENCE: Pardon?

VICTOR COSTAN: Which one am I incrementing here?

AUDIENCE: Zero ne through four. One.

VICTOR COSTAN: Three, two. And then?

AUDIENCE: Three n.

So this becomes a two. And what do I do now?

AUDIENCE: Reiterate over that-- I don't know. I don't know what to call that identity [INAUDIBLE] almost? OK, an array. Printing into your output array one one, one two, two threes, one four.

VICTOR COSTAN: All right. So there's no zeroes and now fives. So one one, one two, one three, and one four. OK, so far so good. This is great.

There's one thing that's missing. For counting sort and for other sorting algorithms, we care about the property called stability. And stability means that if you have two equal keys, or at least two keys that look equal to the sorting algorithm, they might be different objects, because they might be implementing that.

The one that shows up first in the input should also show up first in the output. And that requires particular care, because you can't just look at the keys from your sorting perspective and know which one's supposed to go where. You have to remember where they were in the input.

So if this guy is 3a, and this guy is 3b, I can't use this approach anymore, right?

Because when I'm outputting here, all I know is I have to output a three. I don't have any other information associated with the key. So instead, I have to do something smarter.

AUDIENCE: Either replace your array with a 2-D array. Or I think better would be to replace each value with a length list.

VICTOR COSTAN: OK. So we can replace each value with a length list, which would have the keys that map to it, right. So here I would have a one. Here I would have a two. Here I would have 3a, and then 3b. and here I would have a four. So then I can go through these and output them the right way.

OK, now suppose I'm writing this in C. Suppose I'm in a low level language. And I'm in a low level language because I'm hired by one of these startups that are doing NoSQL databases. And they're writing everything in C to make their things really fast.

So I'm writing an index that uses counting sort. I don't have length lists, because if I'm writing in C, I have to write my own. And that's hard.

So I want to implement this in another way. Length lists are hard. What would I do instead? Can anyone think of another way?

AUDIENCE: I think you can decrement the values for the C in the array that you have, where you have to type the culture of each anyway.

VICTOR COSTAN: OK, so you have the right idea. You're missing one step. So I'll give everyone else a hint so that everyone can catch up. So what I want to do is I want to take this and transform it into something that allows me to go through the keys. So I know I have five keys here. I'm going to make an output array of five elements.

And I want to be able to see four and know that it belongs here. See one, know that it belongs here. See 3a, know that it belongs here. Then probably update the value associated with three. See two, know that it belongs here.

And then when I see 3b, know that it belongs here. So I want to look, when I get to

3a, I want to look inside here. And I want this to tell me that 3 belongs here, 3a belongs here.

So what would the position of 3a be? That's not good, right? Let's call this c instead so that I can say 3a be. So how would I define the position using the sorted property? 3a should go in the index that is how many keys smaller than 3 there are.

So if I can look through here and see how many keys do I have that are smaller than 3, this is where 3a needs to go. If I look at four, there are four keys that are smaller than four. So it needs to go in position four.

AUDIENCE: Well, that almost seems more like a compare. I'm guessing that makes it-- I think it's kind of a comparison model. But you're saying is it greater than. So it's not really counting sort anymore as much.

VICTOR COSTAN: Well, I'm telling you I can compute that using this. So I can use the counting sort algorithm and change this array a little bit so that I can do this trick and know what goes where.

AUDIENCE: You already mentioned using a 2-D array.

VICTOR COSTAN: But a 2-D array would be too much. In the end, I will be changing this in place. So no extra space except for this array of size k. But let's not worry about changing it in place right now. Let's say we're going to make another array of size k.

So I want it to tell me that-- I guess I don't care about this-- but I want it to tell me that one, the first one should go here, the first two should go here, the first three should go here, the first four should go here. How do I do that?

AUDIENCE: Well, you could make that array, right.

VICTOR COSTAN: But how do I compute it?

AUDIENCE: While you're making this one, you can start filling that one in. But while you're making the top one.

VICTOR COSTAN: Can I?

AUDIENCE: It would be like insertion sort though, kind of. So you come across the four. You put it in there, because you know how many there are. But that doesn't make a lot of sense.

VICTOR COSTAN: Yeah, OK. So let's abandon that route. Let's think of something else.

AUDIENCE: Could you populate the array with the number of elements that are less than that [INAUDIBLE]?

VICTOR COSTAN: So intuitively, I want this to tell me how many elements there are that are smaller than two. This should tell me the number of elements there are that are smaller than three, so on and so forth. OK, how would I compute that? Let's see what it's supposed to be. Let's fill it out with real values.

AUDIENCE: Zero.

VICTOR COSTAN: Zero. How many elements smaller than one?

AUDIENCE: Zero.

VICTOR COSTAN: How many elements smaller than two?

AUDIENCE: One.

VICTOR COSTAN: How many elements smaller than three?

AUDIENCE: Two. It's a cumulative sum.

VICTOR COSTAN: OK.

AUDIENCE: On the array above.

VICTOR COSTAN: So this is how many elements smaller than four? Or how many elements smaller than 5 4? OK. what's the difference between these two guys?

AUDIENCE: One.

VICTOR COSTAN: What's the difference between these two guys?

AUDIENCE: One.

VICTOR COSTAN: Yeah, you're right. Sorry. Thank you. What's the difference between these two guys?

AUDIENCE: Two. One.

VICTOR COSTAN: And what's the difference between these two guys?

AUDIENCE: Zero.

VICTOR COSTAN: OK, What did I just write here?

AUDIENCE: Same series up there.

AUDIENCE: Array.

VICTOR COSTAN: All right. So this guy is zero, right, because there's no element that-- there's nothing that's smaller to the smallest key. And then this guy is whatever was here plus this almost. So the difference between this guy and this guy is this.

AUDIENCE: So why go through an array? I mean, why did you bother? Why do we make a new array? Because we could just get that information.

VICTOR COSTAN: Making a new array so that we can see how to compute it. So now we're going to try to right pseudocode that does this in place. So suppose this array is a and this array is pass for position. And suppose-- sorry, not this array.

This array is a. This array is pass. And I start with this. And I want to end up with this. So let's try to write the pseudocode for counting sort.

Counting sort with an array a. I'm not going to write the first two lines that produce this. Let's transform this to this. How would I do that?

AUDIENCE: Initialize an array of the same size.

VICTOR COSTAN: OK. Can we try to do it in place?

AUDIENCE: Sure.

VICTOR COSTAN: How do we do it in place?

AUDIENCE: You could, well for four, you get the four. You're like, oh, I haven't encountered anything below me. So you put it in zero initially for four.

And then you get a one. And you're like, oh, I haven't gotten anything below me. But I forget to keep track of the fact that you have to iterate a whole list ever single time you get a new input.

VICTOR COSTAN: So I don't want to do that, because that's n squared.

AUDIENCE: What you need to do is keep a running sum. Is it a register? Is that what you do call it?

VICTOR COSTAN: Running sum. I like running sum.

AUDIENCE: OK. Keep a running sum of--

VICTOR COSTAN: Sums always start at zero, right?

AUDIENCE: Right. So you keep zero at-- you take the value in each index of that array and add it to sum.

VICTOR COSTAN: OK. So for i iterating from zero to-- so you want each value in this array, right?

AUDIENCE: Yes.

VICTOR COSTAN: So it's going to iterate from zero to what? How many elements do I have there?

AUDIENCE: Length k.

VICTOR COSTAN: OK, almost. So we're using Python numbering, which is zero base indexing. The indices look like this. So it's zero to--

AUDIENCE: [INAUDIBLE].

VICTOR COSTAN: Very good. Thank you. And you said I'm going to add the elements to a sum. So sum is sum plus position of i. OK. And then?

AUDIENCE: The replace is the [INAUDIBLE]. So zero should be zero still. One should be the sum after evaluating zero. You'll need a temp variable.

VICTOR COSTAN: OK.

AUDIENCE: You'll need to graph position i when in temp.

VICTOR COSTAN: Temp is position i.

AUDIENCE: Then say position i is sum before incremental sums. No. That's not it at all.

VICTOR COSTAN: Really?

AUDIENCE: We'll have to say that sum is sum plus temp. That is going to work.

VICTOR COSTAN: OK. How does everyone else feel about this? Does it make sense?

AUDIENCE: Not really.

AUDIENCE: [INAUDIBLE] temporary blast [INAUDIBLE] previous adjuration, because-- so when you first started, it's the very initial case that doesn't work. So like, if you're in the first column, everything's fine. Then you go to column one. You're looking at everything to the left of it. It's still going to be zero.

Then you go to the second column, but you already overwrote the previous column. So you need to store somehow the-- I don't know. It's just the initial case from when it first goes from zero to an actual qualified number. Because otherwise, you're just going to get like zero, zero, zero. And you just overwrite.

AUDIENCE: Can you start [INAUDIBLE]? Was that before you changed?

VICTOR COSTAN: OK. Sorry, I'm getting confused. This is getting hard. I will show you a trick to make life easier.

I'm going to put-- how many elements do I have here? Five, right? So I'm going to put a five here after the array. And then I'm going to ask you, what's this difference.

AUDIENCE: Zero.

VICTOR COSTAN: OK. So now we have this whole array. Can people see what's going on here.? So instead of starting at the beginning, I'm going to start at the end. And I'm going to know-- I know for sure there are n elements. Therefore, the index of this guy is n minus-- so the index of the last key is n minus how many keys I have with this value. Does this make sense?

AUDIENCE: But you're iterating over an order, right? So we can't just take the whole thing and say we're going to shift it over to the right.

VICTOR COSTAN: How about--

AUDIENCE: And you're going through left to right. You'll only know what you see thus far.

VICTOR COSTAN: How about going it for ai from n minus 1 to 0. Will it work then? So what would I write?

AUDIENCE: But isn't that super inefficient? Because then you're starting looking at the whole list. And then you're sort of, rather than just looking at the previous sum that you just-- the cumulative. So your first adjuration, you have to add up everything that you see. Like adjuration, you have to add everything up.

VICTOR COSTAN: So if I add everything up here, what's the result going to be?

AUDIENCE: Five.

VICTOR COSTAN: OK. What's five? So this counts how many zero keys I've seen, how many one keys I've seen, how many two keys I've seen, so on and so forth. So in total--

AUDIENCE: So you're subtracting

VICTOR COSTAN: It's how many keys I've seen. All this, the sum of all these, is how many keys I've sent. How many keys do I have?

AUDIENCE: Five. For each one you see, you can just--

VICTOR COSTAN: So who's five? It's the length of this guy, right? And we usually call that n. So when we're doing sorting, this is n. So maybe it's less confusing. Oh, I already used n in two places. So I guess that's it. I could say the length of a, but there you go.

So I could do the thing that we're going through before. I could figure out my temp variables. And I could make it work. Or I could do this.

AUDIENCE: I think it's the same though, isn't it?

VICTOR COSTAN: Yup. It's the same thing, except I think this is easier to write. Does anyone want to help me write this?

AUDIENCE: Maybe doing once you're starting with the top array, and then finding the bottom one.

VICTOR COSTAN: Yeah.

AUDIENCE: Oh, OK. Well, you just-- you start with the first one and the one ahead of it. And oh, I mean starting with the top right. Sorry.

VICTOR COSTAN: OK, so I have this. And then what do I do?

AUDIENCE: [INAUDIBLE]? Oh, so you're starting from the back.

VICTOR COSTAN: Yep.

AUDIENCE: Well, then you just compare that to-- I mean, you're going to start with zero difference. If you have-- well you don't have any of those last keys, so you'd be able to start with a zero.

VICTOR COSTAN: So what's the difference between five here, which is n, and this guy? What is this?

AUDIENCE: It's going to be zero.

VICTOR COSTAN: But what is it? Why is it zero? So this one's zero, this one's one, this one's two.

What is this? It's the last guy here, right?

AUDIENCE: Yeah, yeah.

VICTOR COSTAN: So this is pass of n minus 1. And this is pass of n minus 2, so on and so forth. So to get from n to the value here, I have to subtract this guy.

AUDIENCE: Pass of i.

VICTOR COSTAN: Pass of i.

AUDIENCE: [INAUDIBLE].

VICTOR COSTAN: OK. Very good.

AUDIENCE: And then update sum. Sum equals a pos value.

VICTOR COSTAN: Sweet. No temp variables, aside from this, I guess. How does this look like? Do people get it?

AUDIENCE: You're subtracting positive i, or you're subtracting a of i.

AUDIENCE: It's all one array.

AUDIENCE: It's the same thing. That's right.

VICTOR COSTAN: So a is this array. a is the input array. And pass is this guy. And this is pass before the four loop. And this is pass after the four loop. So I guess this is pass zero. And this is pass one. And here, we start with pass zero. This, we end up with pass one. OK

So we're able to compute this. There are many ways of doing this, but in the end, you want an array that looks like that. This is counting sort. This is the hard part of counting sort, coming up with that array. Once you come up with that array, you're golden. So let's see that we're golden and produce an output array with the keys in the right order.

So say we have an array called output. And this is going to have these keys in the right order. What's the pseudocode for that?

First, I'm going to create a new array. And I'm going to initialize it with n NIL values. Then what do I do?

AUDIENCE: Iterate over a.

VICTOR COSTAN: Very good. For-- nah, it's too low. Let's do it here.

AUDIENCE: i of a. From zero to n minus 1.

VICTOR COSTAN: OK. What do I do?

AUDIENCE: Out of [INAUDIBLE] has to be-- oh, can we modify pass one as we go?

VICTOR COSTAN: Yeah.

AUDIENCE: So you could say, out of pos one--

VICTOR COSTAN: So by the way, this is pos. The reason I label them with zero and one, so we're doing the change in place.

AUDIENCE: Right.

VICTOR COSTAN: The reason I labeled them is to say that this is what pos is before we going into the loop. This is what pos is afterwards. But it's a single array. So let's call it pos. So out of pos of--

AUDIENCE: Pos of i equals a to the i. Positive i plus pos squared.

VICTOR COSTAN: Yup. And I'm going to use the CLRS, the way which makes me write more. So how this work? I have the survey here. I start at four. What's pos of four?

AUDIENCE: Four.

VICTOR COSTAN: All right, so I'm going to write this as position four. I should probably make this a proper array. One two, three, four, five. So at four, I write four. And then I increment this guy to become five.

Then I get to one. So I look at pos of--

AUDIENCE: One.

VICTOR COSTAN: And that is zero. So I'm going to write one at position zero. And I'm going to increment it.

Then I get to 3a. I look at positive 3. It says 2. So I'm going to write 3a here and increment this. Then I get to two. Pos of two is--

AUDIENCE: One.

VICTOR COSTAN: One. So I write two here. Pos of two becomes two. Then I have 3c, which is pos of 3 is now 3. It's not two anymore. So yay, I'm not overwriting 3a. That's good. And this becomes four. Are people getting what just happened here?

AUDIENCE: Wait, why didn't [INAUDIBLE] to just basically train the next array into an index binder?

VICTOR COSTAN: Yep. So this guy tells me if I have a key, where do I write it in here? So these start out with pointers to the first element that would store that key value. And when I store a key, say when I start 3a, when I get to 3c, I don't want to store it in the same place.

So I have to increment that. I have to say, yo, I wrote 3a at position two. So next time, write it-- next time you see a three, right it at the position following that. And that's what this guy does.

So this is the relatively easy part. And this is the hard magic in counting sort. So how are people feeling about it now? Any nods, or is still confusing as hell?

AUDIENCE: It's a lot. I'm confused.

VICTOR COSTAN: OK. Well what should we do? Do you guys want to ask more questions? Do you want to run through another example? Do you want to try to see how this becomes useful in radix sort, so that you're motivated to figure it out on your own? What would make more sense?

All right. Who wants to do more count sort? Who wants to do some radix sort. All right. Radix sort it is.

Next time you want to move on, tell me understood and I'll believe you. And it'll look good on tape. Two, three--

AUDIENCE: You're not supposed to tell us that there's a camera in here.

VICTOR COSTAN: One, four. I think you're supposed to know, because otherwise you don't know that we're violating your rights. Two, four--

AUDIENCE: This is out the door.

VICTOR COSTAN: One, two, four, three, two, one, four, three. And one more. One, two, three, four. So this is to refresh your memory.

What do keys look like in merge and radix sort? So in concert, the keys have to be numbers from 0 to k minus 1. How about merge sort? What do keys look like?

So radix sort says that a key is a sequence of digits. Say you have d digits in a key. But then each digit isn't necessarily a base 10 digit like we're used to. Each digit is in base k. So each digit can be from 0 to k minus 1. And we're using base k.

How many keys can I represent this way? So if you have numbers of n digits in base k, what's the biggest number that we can represent, or how many numbers can we represent with that?

AUDIENCE: n to the k. No, d to the k. Right?

VICTOR COSTAN: Almost.

AUDIENCE: [INAUDIBLE] the d.

VICTOR COSTAN: All right. So if our base is two, like if we're using bits, then our base is two. And if I have eight bits, then two to the eight. Cool. So if I add one more digit, I get to multiply the number of keys I represent by k.

How do I radix sort? Does anyone remember?

AUDIENCE: We checked the log base k of everything. I guess log base d. Oh, k. It's based in--

VICTOR COSTAN: No. That would be hard math. We don't do hard math. In sorting, if you have integers going into your sort, you only do integer operations. You don't do anything math beyond them.

So what we do is we've broken up the keys into d digits for a reason. We're going to have d rounds in the sort. And in each round, we're going to take all the keys that we have. And we're going to sort them according to one of the digits.

So in one round, we'll sort them according to this digit. In one round, we'll sort them according to this digit, this digit, this digit. Which digit do we start with? What do you guys think?

AUDIENCE: To least significant digit, right?

AUDIENCE: And most significant on the left.

VICTOR COSTAN: So this or this?

AUDIENCE: The right side.

AUDIENCE: 100 is bigger than 1, even though the 1 is greater than the 0 in 100.

VICTOR COSTAN: You're helping me out. So the point I'm trying to make here is radix sort is unintuitive. If we ask you on a quiz what do you start with, your intuition will tell you to start with the most significant digit.

Go against it. In radix sort, you start with the least significant digit and then move your way out. So radix sort goes like this.

AUDIENCE: I mean, it does make sense, because you don't have very much information unless you're looking at bits. You can get a bunch of twos, but that doesn't give you much information. The most information is the smallest bit. And then you move up from there.

VICTOR COSTAN: It depends what information you're trying to get. But maybe you know the algorithm, so you're thinking, oh, by knowing the algorithm, I know that I'll have the most information by looking at it this way. All right, so let's sort these by the last digit. Sweet. Let's sort them by the digit, by the digit before the last digit.

What do I have to do in my sorting? What do I have to pay attention to? So the sorting method that I use has to have a property. It can't be any kind of sorting. Stable.

So the reason we went through all this pain in counting sort is because we want to have a stable sort here. Now, let's try to sort these in a stable manner. This is the first one, two, four, one, three. Then I have two threes, so one, four, three, two, one, two, three, four.

And then I have three fours. Two, three, four, one. Two, four, one, three. Two, one, four, three. Way this isn't good. Two, three, four, one. One, two, four, three.

AUDIENCE: You should cross them off if you write them down.

VICTOR COSTAN: I should. I was hoping you guys would help me if I mess up. So now these are sorted stably. Let's look at these last three that have the same digit here. So they have the same four.

If you look at the last digit, because I used a stable sorting, they're also sorted according to this last digit. So they're sorted according to these last two digits, because the sorting that I used is stable. So now if I sort according to this digit, then if my sorting is stable, they're going to be sorted according to the last three digits.

So as I go from my last digit to my first digit, the keys are going to be sorted according to the last digit, the last two digits, the last three digits, and then all the way up to everything. This is why I need a stable sort. And also, this is why I need to start from the end.

Does this make some sense? What stable sort did we just learn?

AUDIENCE: Counting.

VICTOR COSTAN: Counting sort. All right. So we're going to use counting sort for this. What's the running time for one round? So for one sorting. One counting sort takes how much time?

AUDIENCE: This is a radix sort.

VICTOR COSTAN: Yes. So radix sort is d rounds of counting sort. Count sort this, count sort this, count sort this, count sort this. So one round, one counting sort, what's the running time?

AUDIENCE: [INAUDIBLE].

VICTOR COSTAN: Thank you. Now how about d of these plus the running time?

AUDIENCE: dn plus b.

VICTOR COSTAN: OK, but I want to come back here. And I want to be able to say that radix sort is optimal. I want to be able to say that it is order n. So what do I have to do in order to be able to say that?

AUDIENCE: [INAUDIBLE] k equal to m.

VICTOR COSTAN: So you're going from-- you know the answer. You're going from the fact that you know the answer.

AUDIENCE: [INAUDIBLE].

VICTOR COSTAN: OK, very good. What if we wouldn't know the answer? What do I need to do?

AUDIENCE: Well, we know the first part is order n. So--

VICTOR COSTAN: So d has to be--

AUDIENCE: We want dn to be greater than dk, right?

VICTOR COSTAN: Well, so dn. dn has to be, at most, o of n, right. Because otherwise, the whole thing would go above. So that wouldn't work. So then what can I say about d?

AUDIENCE: Constant.

VICTOR COSTAN: Very good. And how do you write constants in math mode?

AUDIENCE: Order one.

VICTOR COSTAN: Very good. So d has to be order one. Otherwise, it's not going to come out to that. Now, what else do we know? We have this that's order n plus k.

If I said this to be a lot smaller than k, if I set it to be log n, it's going to be order n. If I set it k to be a constant, if I use bits, if I use base 2-- so I said k equal 2-- this is still going to be order n.

So if k goes way below n, this step is still going to be order n. So I might as well set k as high as possible. So k is order n, because that's the highest thing I could set it to. Now why do I want to do that? Yes, you have a ques--

AUDIENCE: [INAUDIBLE] represent in counting sort again? The length of what?

VICTOR COSTAN: So in counting sort, n is your input, how many keys you have. And k is the size of this array.

AUDIENCE: Oh, OK.

VICTOR COSTAN: So you have to be able to map your keys from 0 to k minus 1.

AUDIENCE: It's set by n, basically. Or it's set by the elements.

VICTOR COSTAN: Yeah. It's set by the nature of the keys.

AUDIENCE: OK. Got it.

VICTOR COSTAN: So in real life, we're thinking maybe we have some huge numbers that we want to sort. And we're going to chunk them up into-- when we're writing on the board, we always have to chunk them up in base 10 digits, because that's the only way we know how to write.

But in a computer memory, we can chunk them up into, say, base 10,000 digits. And the fewer digits you have, the faster this is going to run. So we have to figure out what's the base.

And it turns out that if you want to have radix sort run in order and time, well, the number of digits has to be sort of constant. I know that k should be order n, because I have no interest in making it lower than that. So these two bounds together tell me that the keys that I can sort are from zero up to order n of order one.

And this looks terrible, but what it comes up to is that you can sort keys that look like n to some constant for any constant. So you can sort huge keys, as long as huge still means finite. And as long as you can figure out how to map them to numbers.

Does this make some sense? Would we ever want to use merge sort instead of counting sort? Suppose we had a stable merge sort. Would we want to use that instead of counting sort here? What would happen?

So suppose it's stable. So it's correct. The algorithm isn't going to blow up. What's the running time for merge sort? So if I use a merge sort. So if I use the merge sort, it's going to be d times n log n.

So no matter how small d is, I'm still not running in linear time. So merge sort does not go well with radix sort.

So from my end, we're pretty much done. We started with n log n. And we got to a sorting algorithm that's order n.

We started at the beginning of [INAUDIBLE], saying that the best thing we can do is omega-- is that omega-- omega of n. We got to that limit. We're happy. We're going to be done with sorting. Any questions from you guys? That means everyone's confused, right? Yes, thank you.

AUDIENCE: Can you explain what the stability criteria is again?

VICTOR COSTAN: The--

AUDIENCE: Stability for these sorting algorithms. Which ones are stable and what makes it unstable?

VICTOR COSTAN: All right, very good. Thank you. So I like especially the last part, with which ones are stable. I'd like to go through that. So a stable sorting algorithm means that if you have two keys that are equal, the key that shows up first in the input is the key that is produced to the output.

So in this model, your keys are not necessarily integers. Your keys might be those weird classes that implement some method that maps them to integers. So say there is a method there, __int__, that gives you the integer for that.

So the sorting algorithm would only see a three here. But in fact, this is a complex object. And this is another complex object, but the sorting only sees the three. If this guy shows up before this guy in the input, they have to show up in the same order in the output.

AUDIENCE: Why would that be bad if they're switched?

VICTOR COSTAN: It's not stable. If they're switched, then when we're using a stable sorting algorithm here. So here, the key is this complicated object. But say we're in the second round. We're in this round, which we played with. Even though the key is this whole complicated object, the only thing that the counting sort sees is this number.

So this guy looks like three. This guy looks like three. And these three guys, although they're different, they look like four.

If I don't output them in the right order-- say I output this one all the way at the end-- then I'm going to get two, three, four, one to be down here. And now my numbers aren't sorted by the last two digits anymore. So it breaks any algorithm that assumes stability.

So stability is something that you get from a sort, because it's convenient to assume it in some other algorithm that builds up on that sort. If you don't need it, you don't care about it. But in some cases, you need it.

And for the second part, which algorithms are stable. Is insertion sort stable?

AUDIENCE: I assume so. I mean, stable is being correct, right?

VICTOR COSTAN: No. We mean that property there.

AUDIENCE: Oh, I see. You mean in order.

VICTOR COSTAN: Yep.

AUDIENCE: Oh, OK. Insertion sort goes in order. But I guess it could push other things out of order.

VICTOR COSTAN: So insertion sort, you're doing swapping to move things to the left. But if you find two things that are equal, you're never going to swap them. So insertion sort is in order, is stable.

Merge sort, the one we gave you in that list is not stable. But there is the one character change that makes it stable. And you should look at today's lecture notes to find out what that is. So merge sort can be stable.

Heapsort, stable or unstable? Unstable. And there's a really small example that you should look at. Counting sort, stable or unstable?

AUDIENCE: Stable.

VICTOR COSTAN: Thank you. It would have broken my heart if this would have come out wrong. And radix sort?

AUDIENCE: Probably. Yes.

VICTOR COSTAN: Probably stable, right? All right. Any more questions? I like that question by the way, because you made me do this. I like that. Any more questions? All right, thank you guys.