ProgrammingAbstractions-Lecture15

Instructor (Julie Zelenski):Hey. Welcome to Wednesday. Things that have happened and we're talking about sorting. We've got lots of sorting to talk about today. Some more sorting we'll talk about on Friday. That will be covered in Chapter 7 there. And then things that are happening. Boggle is coming in this Friday, so hopefully you're making good progress on Boggle. How many people just totally done with Boggle? All done, all behind you? Oh, that's a very healthy number. Like to see that. I did put a note here about Boggle Wednesday. Boggle is due Friday and given our late day policy we actually count days class meets and since Monday's a holiday that technically means if you were to handle Boggle late it is due on Wednesday.

Here's something I also want to say while I say that. Don't do it. It's a bad idea. You need to prep for the mid-term. I'm not gonna hand out assignments on Friday. I'm gonna just try to clear your plate for getting your head together for the mid-term. I think if you were finishing a late Boggle in that period you're really short changing yourself. The mid-term counts a lot more, kind of covers more ground, is more important in the big scheme of things than working on a late assignment. So do make your plans to get it done on Friday so that you can actually not have it hanging over your head when you're trying to get ready for that mid-term next Tuesday evening. Terman Auditorium, which is a little hike away from here over there in the Terman Building, 7:00 to 9:00 p.m. will be the big party zone.

Today I put up the solution to the problems I gave you on Monday, so you have both of those to work on. I should encourage you not to look at them kind of in parallel. I think that's one of the bad ideas to say. Look at the problem, look at the solution, and say yeah, that looks right. They are right, actually. I'll tell you that. You don't need to do that. What you need to do is say good, I have generated that and the best way to check whether you've generated it is not to say yeah, that looks like what I would have done is to do it, right? Do it on the paper without the solution around and then look at your answer relative to the solution and see if there's something you can learn about how our solution, your solution aren't exactly the same. There's a lot of partial credits. It's not like you need to be able to reproduce and it's not like there's only one correct answer, but, sort of, that practice is the best practice for the exam.

Getting ready to think about how to do it on paper. The other handout also very worthwhile reading. It's kind of long, but it has a lot of information that the section leaders and students over time have kind of gathered their wisdom and tried to write down just about how to make sure you do well on the exam. I think it's one of the more challenging aspects is to try to come up with the fair way to give an exam in a class that is largely experience and skill based like [inaudible] is that we can capture an assessment of how you're doing, right? But in this funky format of being on paper and stuff. So definitely do read that handout for, sort of, our advice and thoughts on how to make sure that you do succeed in that endeavor. Anything administratively you want to ask or talk about? Question?

Student: So if you have a conflict with the mid-term and have all ready filled [inaudible] when will we hear back on that?

Instructor (Julie Zelenski):So we’re gathering them all by Thursday and then we’re gonna try to do this big intersection. So either late Thursday or, sort of, Friday is when we’re kind of sort out all the possibilities and try to come up with a schedule that gets everybody taken care of. So you should hear by the end of this week you will know what’s going on. All right.

So I’m going to pick up on where I left off on Monday. And this was the code for the selection sort algorithm. And, in fact, I’m actually not gonna – I’m gonna be showing the code for each of the algorithms that I’m using here, but I’m gonna encourage you to not actually get really focused on the details of the code. This is actually one of those places where the code is an embodiment of the idea and the idea is the one that actually we’re interested in studying, which is the approach it takes to sorting, how it decides to get that data in sorted order, and then how that algorithm performs and the kind of details of where it’s a plus one and minus one less than – it’s actually a little bit less important for this particular topic.

So what I’m gonna show you actually is a little demo of this code in action that’s done using our graphics library just to show you what’s going on. And so this is the selection sort code that the outer loop of which selects the smallest of what remains from the position I to the end of the array and then swaps it into position after it’s done that finding. So this inner loop is basically just a min finder finding the minimum index element from I to the end. So I’m gonna watch this code step through. So you can see as it moves J down, which goes bopping by in green it’s actually updating this kind of min, the blue pointer there, that says, well, what’s the min I’m seeing so far?

Early on, it was 92, then it bumped down to 45, then it bumped down to 41, then 28, then 8 and finally got to the end and found nothing smaller so it says, okay, that must be where the min is. Let me move that guy to the very front. So exchanges those two guys, flopping another one back in there. Now, we know we have the very smallest one out of the way. It’s in the right spot. We have nothing more to do with that. There’s no changes that’ll need to be made to anything to the left of I and we do the same operation again. Select from the remaining n minus one elements, the minimum element. So watch the green, sort of, finger move down while the blue finger kind of stays behind on the smallest it’s seen so far. So it’s okay, so, well, then 15 is it. Swap that guy up into the second slot.

Do it again. Keep doing this, right? Working it’s way down and then, as you see, the array’s kind of going from the left to the right, the smallest most elements, right? Being picked off and selected and moved out and then leaving the larger ones kind of over here in a little bit of a jumbled mess to get sorted out in the later iterations of those loops. So there is select and sort in action, right? Doing its thing. Okay. Let me do another little demo with you because I have – there’s lots of ways to look at these things. I’m gonna do one that actually can do slightly bigger ones and so I’m gonna set this guy up and it’s gonna show, in this case, the red moving, right? And it holding onto the smallest it's seen and updating as it finds smaller ones. Move it to the front and then I'm actually gonna speed it up a little bit which is, let's see, go a little faster.

So you'll note that it took in this case and it does a little bit of counting. I'm gonna look at the analysis in a minute about a lot of comparisons and not so many moves. That's actually kind of the model behind selection sort is to do a lot of testing. That first iteration looks at every one of the n elements to find the smallest and then it does one move to pull it into position. Then it does the same thing again. Does another task looking at this time only n minus one elements remain, but doing another full set of comparisons to decide who is the smallest of what remains and then swapping that to the front. So it's taking a strategy that seems to indicate that comparison, right? Which it's gonna do a lot of, a very few number of moves as a result, because it identifies where something goes.

It pulls it out, puts it to the front, and doesn't do a lot of shuffling around. If I've put kind of a higher number of things in here and let it go it appears to kind of move very slowly at first, right? Because it's doing a lot of work to find those small ones and then move them down to the front, but as it gets further along it will tend to kind of speed up toward the end and that's because in subsequent iterations, right? There's fewer of the elements remaining to look at and so kind of a tiny little portion of it's been sort of so far the first four. If I – I think if I let this go it will go way too fast. Yeah. Way too fast to see it.

There's a lot of things I want to show you because I'm gonna add in the option for sound. So one thing to learn a little bit about how to visualize sorts, right? I think that this kind of technique sometimes it's easier to look at than the code is to see how it's putting things in order. Several years ago I had a student who actually was blind and these visualizations do squat for someone who's blind it turns out. So she helped me work out some ideas about how to do a different, sort of, interface. One that visualizes it using it when you're different senses would just do sound and so the change I've made to the code here will play a tone each time it's moving an element and the frequency of that tone is related to the height of the bar. So the smaller height bars are lower frequency, so lower tones.

And then the taller bars have a higher frequency, higher pitch to them. And so you'll hear the movement of the bars being expressed with this tone loop so – and I'm not hearing any sound. Is my – are we getting sound? There we go. I'm not getting sound now, how about you? No, we're not getting any sound at all. Here we go. I'm getting sound now. How about you?

Well, and that's true. It's only when they move. That's a very good point. What about – I'm running a little bit slower than I'm thinking. All right. I can get it to – all right. Let's try. Now, we're – I feel like we're hearing something. We're still. Definitely getting sound now. Okay. Now it's gonna be a little loud. There we go.

The march of the low order tones can be moved into place doing a lot of comparison and then you'll hear kind of the speed up I talked about. Toward the end is the later iterations.

Finishing it up. So giving you a kind of a little memory for, sort of, what's the action selection sort. The small amount of tones and the kind of sparse distance meeting shows that it's doing a lot of work in between each of those moves, right? And then the fact that the tones kind of advance from low to high tells you that it's working on the smaller values up to the larger values. Kind of working it's way to the end and that speeding up it closing in on the remaining ones as it gets to the end.

Left that around because even if you are can see it you can also hear it and that may help something. What we're gonna look at is how much work does it do? So if I go back and look at the code just to refresh what's going on, right? This inner loop is doing the comparisons all the way to find the min elements and so this is gonna iterate n minus one times on the first iteration, then n minus two, then n minus three, and all the way down those final iterations, three to look at, two to look at, one to look at and then one swap outside that. So what I actually have here is the sum of the values n minus one compares, plus one swap, but as two comparisons one swap, so one just swaps but actually the terms I'm looking at here are the number of comparisons the sum of the numbers one to n minus one is what we're trying to compute here.

Then if you've seen this some are the [inaudible], some are the Gaussian, and some you may all ready know how to solve it, but I just kind of showed you how to do the math to work it out, which is the term you're looking for is this sum. If you add it to itself, but rearrange the sequence of the terms so that they cancel each other out you'll see that the n minus one plus one gives you an n and that what you end up with is n minus one n's being added together is what the sum of this sequence against itself is and so we can divide by two to get the answer we're looking for, so we have a one-half n squared minus n term. Which in the big O world, right? Just comes down to n squared.

So tell this – it's a quadratic sort, right? That we would expect that if it took a certain amount of time to do a hundred, three seconds, then if we double that input we expect it to take four times as long. So if it was three seconds before it's 12 seconds now. So growing as a parabola is a fairly sharp steep curve to get through things. All right. So let me give you an alternative algorithm to this. Just to kind of think, that's gonna be the theme is, well, that's one way to do it and that has certain properties. Let's look at another way and then maybe we'll have some opportunity to compare and contrast these two different approaches to see what kind of tradeoffs they make in terms of algorithm choices.

So the way you might handle a deck of a cards – sorting a deck of cards. So if I'm handing out cards to people you're getting each card in turn that one way people do that is they pick up the first card and they kind of – you assume it's trivially sorted, right? It's in the right place. You pick up the next card and you decide, well, where does it go relative to the one you just had. Maybe you're just sorting by number order and you say, well, okay, it's greater than it and goes on this side. You pick up your next one and it's like it goes in between them. And so you're inserting each new element into the position of the ones that are all ready sorted.

So if you imagine applying that same thing in terms of computer talk in a vector sense is you could imagine kind of taking the vector, assuming that the first element is sorted, then looking at the second one and deciding, well, where does it go relative to the ones all ready sorted? The ones to my left and kind of moving it into position, shuffling over to open up that space that's gonna go into and then just extending that as you further and further down the vector. Taking each subsequent element and inserting it into the ones to its left to make a sorted array kind of grow from the left side. So it grows from the left somewhat similar to selection sort, but actually it will look a little bit different based on how it's doing its strategy here.

So let me give you the insertion sort code. So my outer loop is looking at the element at index one to the element of the final index of the raise side minus one and it copies that into this variable current to work with and then this inner loop is doing a down to loop, so it's actually backing up from the position J over starting from where you are to say, well, where does this – it keeps sliding this one down until it's fallen into the right place. So we'll move over the 92, in this case, to make space for the 45 and then that's kind of the whole iteration on that first time. The next time we have to look at 67. Well, 67's definitely gonna go past 92, but not past 45. 41 is gonna need to go three full iterations to slide all the way down to the front. 74 is just gonna go over one number. Just needs to slide past one.

So on different iterations, right? A little different amount of work is being done, right? This loop terminates when the number has been slotted into position. We won't know in advance how far it needs to go, but we go all the way to the end if necessary, but then kind of sliding each one up by one as we go down. So that one moved all the way to the front, 87 just has one to go, eights got a long way to go. All the way down to the very front there. Sixty-seven goes and stops right there, 15 almost to the bottom, and then 59 moving down this way. So kind of immediately you get the sense it's actually doing something different than selection sort, just visually, right? You're seeing a lot more movement for a start that elements are getting kind of shuffled over and making that space so it's definitely making a different algorithmic choice in terms of comparison versus moving than selection sort did, which is kind of a lot of looking and then a little bit of moving.

It's doing kind of the moving and the looking in tandem here. If I want to hear how it sounds, because nothing's complete without knowing how it sounds, I can go back over here and let me turn it down a little bit. So you hear a lot more work in terms of move, right? Because of the signal I'll speed it up just a little bit. So, as you can see, a kind of big [inaudible] a lot of work [inaudible] as it slides that thing down into position and then finding it's home. Some will have come a long distance to travel like that last one. Other ones are very quick where they don't have so far to go in turn. I think that's it. I'll crank it up to like a, sort of, a bigger number, let's say, and turn off the sound and just let it go and you can kind of get a sense of what it looks like.

It seems to move very, very quickly at the beginning and then kind of starts to slow down towards the end. If I compare that to my friend the selection sort, but it looks like insertion sort is way out in front and its gonna just win this race hands down, but, in fact, selection sort kind of makes a really quick end runaround and at the end it actually came in just a few fractions of a second faster by kind of speeding up towards the end. So it's like the tortoise and the hare. It looks like insertion sort's way out in front, but then actually selection sort manages to catch up. I'm gonna come back and look at these numbers in a second, but I want to go back and do a little analysis on this first.

If you take a look at what the code is doing and talk about kind of what's happening, right? We've got a number of iterations of the outside, which is sliding me to this position. This inner loop is potentially looking at all of the elements to the left. So on the first iteration it looks at one. At the second iteration it looks at two. The third one three and so on until the final one could potentially have n minus one things to examine and move down. If that element was the smallest of the ones remaining it would have a long way to travel. But this inner loop, right? Unlike selection sort that has kind of a known factor it's actually a little bit variable because we don't know for sure how it's gonna work.

Potentially in the worst case, right? It will do one comparison move, two and three and four all the way up through those iterations, which gives us exactly the same Gaussian sum that selection sort did. One plus two plus three all the way up to n minus one tells us its gonna be n squared minus n over two, which comes down to O of n squared. That would be in the absolute worst case, right? Situation where it did the maximum amount of work. What input is the embodiment of the worst case in selection sort? What's it gonna be? If it's flipped, right? If it's totally inverted, right? So if you have the maximum element in the front and the smallest element in the back, right? Every one of them has to travel the maximum distance in this form to get there.

What is the best case to do the least amount of work? It's all ready sorted. If it's all ready sorted then all it needs to do is verify that, oh, I don't need to go at all. So it actually – the inner loop only does one test to say do you need to move over at least one? No. Okay. So then it turns out it would run completely linear time. It will just check each element with its neighbor, realize it's all ready in the right place relative to the left, and move one. So, in fact, it will run totally quickly, right? In that case. And in the average case you might say, well, you know, given any sort of random permutation of it, each element probably has to go about half the distance. So I don't have to go all the way, some don't have to go very far, some will go in the middle. You would add another factor of a half onto the n squared, which ends up still in the big O just being lost in the noise.

You say, well, it's still n squared. But it probably does. It is a little bit less of an n squared than selection sort. So if I go back here to my – well, it was counting for me. That the mix of operations between insertion sort and selection sort, so that's selection sort on the top and insertion sort that's beneath it, shows that selection sort is doing a lot of compares. In this case, I had about 500 elements and it's doing about n squared over two of them, 250,000 divided by two there, and so it really is doing a full set of compares. A whole n squared kind of compares is doing a small number of moves. In this case, each element is swapped so there's actually two moves. One in, one out. So it does basically a number of moves that's linear relative to the number of elements.

The insertion sort though is making it, sort of, a different tradeoff. It's doing a move and a compare for most elements, right? In tandem and then that last compare doesn't do a move with it. So there should be roughly tracking in the same thing. But they look closer to n squared over four. Showing that kind of one-half expected being thrown in there. But in the total, the idea is that it does about 100,000 compares a move. This one does about 100,000 compares that when you look at them in real time they tend to be very, very close neck in neck on most things. So if I do this again giving it another big chunk and just let it go, again, it looks like insertion sorts off to that early lead, but if you were a betting person you might be putting your money on insertion sort. But selection sort is the ultimate comeback kid and toward the end it just really gets a second wind.

In this case, beat it by a little bit bigger fraction this time. If I put the data into partially sorted order. So now, in this case, I've got about half of the data all ready where it's supposed to be and another half that's been randomly permuted and now let it go. Insertion sort takes an even faster looking early lead and selective sort, in this case, never notices that the data was actually in sorted order. But time actually is a little bit artificial here and, in fact, the number of comparisons is maybe a better number to use here, but it really did a lot more work relative to what insertion sort did because insertion sort was more able to recognize the sortedness of the data and take advantage of it in a way that selection sort totally just continued doing the same amount of work always.

That's kind of interesting to note, right? Is that when we're talking about all these different sorting algorithms, right? That we have multiple evidence, because actually they really do make different tradeoffs in terms of where the work is done and what operation it prefers and what inputs it actually performs well or poorly on. That selection sort is good for it does exactly the same amount of work no matter what. If it's in sorted order, or reversal order, or in random order, right? It always is guaranteed to do the kind of same amount of work and you don't have any unpredictability in it. That's both an advantage and a disadvantage, right? On the one hand it says, well, if you knew that you needed this sort to take exactly this time and no better, no worse would be fine then actually having that reliable performer may be useful to know.

On the other hand, it's interesting to know that insertion sort if you gave it to data that was almost sorted then it would do less work, right? Is an appealing characteristic of that. And so having there be some opportunity for it to perform more efficiently is nice. The other thing, also, is about this mix of operations. Whether it considers comparisons or moves are more expensive operation. For certain types of data those really aren't one-to-one. That a comparison may be a very cheap operation and a move may be expensive or vice versa depending on kind of the data that's being looked at.

For example, comparing strings is often a bit more expensive than comparing numbers because comparing strings has to look at letters to determine when they distinguish. If you had a lot of letters in the front that were overlapping it takes, sort of, more work to distinguish at what point they divide and which one goes forward. On the other hand, moving a large data structure if it were a big structure of student information, right? Takes more time than moving an integer around. So depending on what the data that's being looked at, there may actually be a real reason to prefer using more comparisons versus moves. And so examples I often give for this are like if you think about in the real world, if you were in charge of sorting something very heavy and awkward like refrigerators you kind of line up the refrigerators by price in the warehouse or something.

You would probably want to do something that did fewer moves, right? Moves are expensive. I'm picking up this refrigerator and I'm moving, I don't want to move it more than once, right? And so you might want to go with the selection sort where you go and you figure out who's the cheapest fridge, let me pull that one and get it over here, and now I'm not gonna touch it again rather than kind of sitting there with your insertion sort and moving the fridges one by one until you got it to the right spot. But if I were in charge of finding out who was, let's say, the fastest runner in the mile in this class you probably would not enjoy it if my strategy were be to take two of you in a dead heat and say run a mile and see who won. And now whoever won, okay, well, you get to run against the next guy. And if you win again you get to run against the next guy.

Like you might just say hey, how about you let me get ahead of that person if I beat them once. I don't want to have to go through this again. Like fewer comparisons would certainly be preferred. So both of these, I would say, are pretty easy algorithms to write. That is certainly the strength of selection and insertion sort. They are quadratic algorithms which we're gonna see is not very tractable for large inputs, but the fact that you can write the code in eight lines and debug it quickly and get it working is actually a real advantage to the – so if you were in a situation where you just needed a quick and dirty sort these are probably the ones you're gonna turn to.

So just some numbers on them, right? Is 10,000 elements on my machine kind of an unoptimized contact was taking three seconds. You go up by a factor of two. We expected to go up by about a corresponding factor of four and the time it roughly did. Going up by another factor of two and a half again going up. By the time you get to 100,000 though, a selection sort is slow enough to really be noticeable. It's taking several minutes to do that kind of data and that means if you're really trying to sort something of sufficiently large magnitude a quadratic sort, like insertion sort or selection sort, probably won't cut it.

So here's an insight we're gonna kind of turn on its head. If you double the size of the input it takes four times as long. Okay. I can buy that, right? I went from ten to 20 and went up by a factor of four. So going in that direction it feels like this growth is really working against you, right? It is very quickly taking more and more time. Let's try to take this idea though and kind of turn it around and see if we can actually capitalize on the fact that if I have the size of the input it should take one quarter the amount of time. So if I had a data set of 100,000 elements and it was gonna take me five minutes if I try to sort it in one batch. If I divided it into two-50,000 element batches it will take just a little over a minute to do each of them.

Well, if I had a way of taking a 50,000 sorted element and a 50,000 sorted input and putting them back together into 100,000 combined sorted collection and I could do it in less than three minutes, then I'd be ahead of the game. So if I divided it up, sorted those guys, and then worked them back together and if it didn't take too much time to do that step I could really get somewhere with this. This kind of turning it on its head is really useful. So let's talk about an algorithm for doing exactly that. I take my stack. I've got a stack of exam papers. Maybe it's got a couple hundred students in it. I need to just divide it in half and I'm actually gonna make a very – the easy, lazy decision about dividing it in half is basically just to take the top half of the stack, kind of look at it, figure out about where the middle is, take the top half and I hand it to Ed.

I say, Ed, would you please sort this for me? And I take the other half and I hand it to Michelle and I say would you please sort this for me? Now, I get Ed's stack back. I get Michelle's stack back. They're sitting right here. All right. So that was good because they actually take a quarter of the time it would have taken me to do the whole stack anyway. So I'm all ready at half the time. What can I do to put them back together that realizes that because they're in sorted order there's actually some advantage to reproducing the full result depending on the fact that they were all ready sorted themselves?

So if I look at the two stacks I can tell you this, that someone over here starts with Adams and this one over here starts with Abbott. The very first one in the output has to be one of the two top stacks, right? They can't be any further down, right? This is the sorted left half. This is the sorted right half, right? That the very first one of the full combined result must be one of those two and it's actually just the smaller of the two, right? So I look at the top two things. I say, Abbott, Adams, oh Abbott proceeds Adams. Okay. Well, I take Abbott off and I stick it over there. And so now that exposes Baker over here. So I've got Adams versus Baker. And I say, oh, well, which of those goes first? Well, Adams does.

I pick Adams up and I stick it over there. So now that exposes, let's say, Ameliglue, my old TA. Ameliglue and Baker. And I say, oh, well, which of those? Ameliglue. And at any given point where there's only two I need consider for what could be the next one and so as I'm doing this, what's called the merge step, I'm just taking two sorted lists and I'm merging. So I'm kind of keeping track of where I am in those two vectors and then taking the top of either to push onto this collection I'm building over here and I just work my way down to the bottom. So that merge step, right? Is preserving the ordering. Just kind of merging them together into one sorted result.

If I could do that faster then I could have actually of sorted them and then I'm actually ahead of the game and there's a very good chance for that because that, in fact, is just the linear operation, right? I'm looking at each element, deciding, and so I will do that comparison and time to decide who goes next, right? I look at this versus that and I put it over there. I look this versus that and I put it over there. Well, I'm gonna do that until this stack has n things in it. All of them have been moved. So, in fact, it will take me n comparisons to have gotten them all out of their two separate stacks and into the one together. So it is a linear operation to do that last step and we know that linear much quicker to get the job done than something that's quadratic. Yeah?

Student: When you merge the halves are you taking the higher one in the alphabet or the lower one?

Instructor (Julie Zelenski):Well, it typically I'm taking it in the order I'm trying to sort them into, right? Which is increasing. So I'll typically start at A's and work my way to Z's, right? So it turns out it's completely –

Student: It doesn't really matter.

Instructor (Julie Zelenski):It doesn't really matter is the truth, but the – whatever order they're sorted in is the order I'm trying to output them in. So, in fact, these are the – if they're first on the sorted piles then they'll be first on the alphabet pile. So whatever that first is. So I'm gonna actually go – look at the code for a second before we come back and do the diagramming. We'll go over here and look at the stepping part of it. So this is the code that's doing merge sort and so it has a very recursive flavor to it. It does a little calculation here at the beginning to decide how many elements are going to the left and going to the right.

As I said, it does no smart division here. So this is called an easy split hard join. So the split process is very dumb. It says figure out how many elements there are, divide that in half, take the one from zero to the n over two and put them in one separate subvector and take the ones from n over two to the n and put them in a second subvector and then recursively sort those. So let's watch what happens here. Computes that and says, okay, copy a subvector of the first, in this case four elements, copy a subvector that has the second four elements, the second half. So I've got my left and my right half and then it goes ahead and makes a call under merge sort, which says merge that left half and merge that right half and then we'll see the merge together.

So I'm gonna watch it go down because the thing that's gonna happen is when I do a merge of this half it's kind of like it postponed these other calls and it comes back in and it makes another call, which does another division into two halves. So taking the four that are on the left and dividing them into two and two and then it says, okay, now I merge the left half of that, which gets us down to this case where it divides them into one on the left and one on the right. And then these are the ones that hit my base case. That an array of size one is trivially sorted. So, in fact, the merge sort never even goes into doing any work unless there's at least two elements to look at.

So when it makes this call to the merge the 45 it will say, okay, there's nothing to do. Then it'll merge the 92 and works for the 92, which also does nothing. And now the code up here's gonna flip. So be warned about what's gonna happen here is I'm gonna show you what the process of the merge looks like, which is gonna do the left-right copy to the output. So this code looks a little bit dense and, again, this is not the time to get really worried about what the details of the intricacies of the code are. I think you really want to kind of step back and say conceptually the process I described of taking them off the two piles and then merging them is very much the take home point for this. And so what this upper loop is doing is it's basically saying well, while the two stacks, the two piles, the two halves, whatever, each have something left in them.

Then compare them and take the smaller one off the top. So it's keeping track of all these indices, right? The indices of the left subarray, the indices of the right subarray and the indices of the right output array and it's actually kind of – and each step is putting a new one, copying one from one of the left or the right onto the output. And so that upper loop there said is 45 less than 92? It is, right? So it copies 45 and then at this point, right? There's nothing left on the left, so it actually drops down to those last two pieces of code, which, actually, do to copy the remainder from if there's only one stack left then just dump them all on the end. So we'll do the dump of the end of the 92. And so I've reassembled it. And so when I get back to here then I have merge sorted the left side and then I go through the processes of merge sorting the right side.

Which copies the 62 and the 41 down to the things and then does a merge back. Let me get to the stage where I'm starting to do a little bit bigger merges. So here I am with indices on my left and my right side and then my output index and it's like, okay, is 45 less than 41? If so then take 41 here and kind of advance P2 over and still see the 41 go across and it moved both the P and the P2 up an index to say, okay, now we're ready to pick the next one. So it looks at P1 versus P2's, decides it's pulling from the left side this time, and then now it still has the last most members of the two piles to look at, takes from the right side, and then we'll just dump the end of the other side.

So then in going through this process to do it all again, kind of break it all the way down. So it's just using recursion at every stage. So at the small stages it's a little bit hard to figure out what's going on, but once it gets back to here, right? It's just doing the big merge, let it go a little too fast there, to build it back up. So there's one thing about merge sort I should mention, which is merge sort is using extra storage. And I'm gonna get to a stage where I can explain why that's necessary. But selection sort and insertion sort both work with what's called in place and that means that they are using the one copy of the vector and just rearranging things within it so it doesn't require any auxiliary storage to copy things over and back and whatnot. That it can do all the operations on one vector with a little bit of a few extra variables around.

That merge sort is really making copies. That it divides it into these two subarrays that are distinct from the original array and copies them back. So it copies the data away and then it copies it back in. And the reason for that actually is totally related to what's happening in this merge step. That if I had – well, actually this is not the step that's gonna show it. I'm gonna show it on this side. That if it were trying to write over the same array – oh, now I made it go too fast and that was really very annoying. All right. Let me see if I can get it one more time. Do what I wanted it. Is that when it's doing the copying, if it were copying on top of itself it would end up kind of destroying parts of what it's working on. Oh, it's – I see what. We're gonna get this mistake the whole time. Okay.

Is that as it was doing the copy, right? If it's – if this really were sitting up here and this really were sitting up her and if we were pulling, for example, from the left side we would be overriding something that was all ready – pulling from the right side, I'm sorry. We'd be overriding something that was on the left side. And so if it did that it would have to have some other strategy for then, well, where did it put this one that it was overriding? Like if it's pulling from the left side it's actually fine for it to override in the output array, but otherwise, right? It would be writing on something it was gonna need later. So the easiest thing for merge sort to do is just to move them aside, work on them in a temporary scratch base, and then copy them back. There are ways to make merge sort in place, but the code gets quite a bit more complicated to do that.

So your standard merge sort algorithm does use some auxiliary storage that's proportional to the size of the array. So that ends up being a factor in situations where you're managing a very large data set where, in fact, maybe it requires, right? A large amount of memory all ready for the ones that making a duplicate copy of it to work on may actually cost you more than you can afford. So merge sort sometimes is ruled out just because of its memory requirements despite the fact that it has a performance advantage over insertion sort and selection sort. What does it sound like though? That's what you really want to know. So let's first just watch it do it's thing and so you'll see it kind of building up these sorted subarrays, kind of working on the left half, and then postponing the right and coming back to it and so you'll see little pieces of it and the little merge steps as it goes along.

And then you can get a little glimpse of kind of the divisions down there. Let me actually run it again in a little bit bigger thing. And I'll turn this on in just a second. Just like as it goes faster and faster let me make my sound go and we'll see how much we can stand of it. It's pretty noisy you'll discover. Volume, good volume? Oh, yeah. Yeah. A lot of noise. Oh, yeah. Okay. So you definitely get the kind of 1960's sound tone generation there, but you get this, I mean, you can hear the merge step, right? Is very distinguishable from the other activity.

It's doing the kind of smaller and smaller subarrays it sounds just a little bit like noise, but as you see the larger subarrays being joined this very clear merging sound emerges from that that you can hear and see that in the very end, sort of, one big long merge of taking the two piles and joining them into one. A lot of noise, right? So that should tell you that there is a pretty good amount of movement going on. Question?

Student:[Inaudible]

Instructor (Julie Zelenski):It's merging them after they've been sorted. So you'll – when you see a merge you'll always see two sorted subarrays being joined into one larger sorted subarray. So at the very end, for example, you'll have these two sorted halves that are being merged down.

Student: And how much is the sorting [inaudible]?

Instructor (Julie Zelenski):Well, it's recursion, right? It's like it sorts the half, which sorts the quarters, which sorts the A's. And so at any given point what it really looks like it’s sorting is these little one element arrays, which are being merged into a two element array. Like all the work is being done in merging really. That sorting is kind of funny. When does it actually sort anything? Like never actually. It only merges things and by dividing them all the way down into all these one element arrays the merging of them it says, well, here’s these trivially sorted things. Make a two element sorted thing out of them and that you have these two element things, merge them into one-four element thing, which you merge into an eight element thing and all the way back. So all of the work is really done in the merge step.

Kind of on the sorting angle it actually is deferring it down to the very bottom. So that’s a kind of odd thing to do, right? It really just divides it all up into one – a hundred piles, each of size one and then kind of joins those into one pile, these into one pile, these into one pile, and so now I have 50 piles, right? That are each a size two. Now I join them into 25 piles of size four and then 12 piles of size eight and so on. So this is the code for the outer point of the merge algorithm. I’m actually not gonna look at the merge algorithm very in depth. I’m really more interested in the kind of outer point of this algorithm and so the steps that are going on here is being a little bit of constant work. Let me actually – we do a copy operation that copies out the left half and the right half.

Both of those operations together require linear time. So I’ve looked at every element and I’ve copied the first half onto something, the second half onto something. So that took – I looked at every element in that process. I make two calls on the left and the right side and then I do a merge on the end. We talked earlier about how that was linear and the number of elements because as I build the two piles into one sorted pile every element is touched in that process as it gets chosen to be pulled out. So linear divide and a linear join and then there’s this cost in the middle that I’d have to kind of sort out what’s happening, which is there’s sort of n work being done at this level, but then I make two recursive calls that should take time n over two.

So the input they’re working on is half, again, as big as the one I have. How much time do they take? Well, there’s my recurrence relation that allows me to express it. T of n is n, so at this level this kind of represents the copy step and the merge step at this level plus the work it took to get the two halves in sorted order. So I’m gonna show you a slightly different way of looking at recursive analysis just to kind of give you different tools for thinking about how to do this. I showed you last time how to do the repeated substitution and generalization. The pattern – I’m gonna show you this way to do it with a little bit of tree that kind of draws out the recursive calls and what’s happening in them. That the merge sort of an input of size n, does n work at that level?

The copy in the merge step, right? For there plus it does two calls to merge sort of n over two. Well, if I look at each of those calls I can say, well, they contribute an n squared and an n squared on each side. So this one has n squared copy and merge. This one has n squared copy and merge and so, in effect, I have another n squared plus itself there and then this level, right? Looks at the n over four, which has four n over four components. So that actually at each level in the tree that every element, right? Is being processed in one of those subcalls across it. So every element is up here and they're in two subgroups and then four subgroups and then eight subgroups.

And that each element is copied and merged in its own subgroup at each level of the recursion. So that kind of gives me this intuition that there's n work being done on every level, every element copied and merged as part of that there. And so then what we need to complete this analysis is just to know how deep this tree grows. How far we get down in the recursion before we hit the base case. So we're dividing by two each time. N over the two, over two to the second, to the third, to the fourth, and so on. So at any given level K down, right? We have n divided by two to the K. What we want to solve for is where n over two to the K equals one. So we've gotten to this smallest case where we have those trivially sorted one element inputs to look at and so we just do a little math here, rearrange that, right? Divide by two to the K or multiply by two to the K both sides when n equals two to the K. Take the log base two of both sides and it will tell me that K is log base two of n.

That I can divide n by K by two K times, where K is the log base two of n, before it bottoms out with those one element vectors. So log n levels, n for level tells me the whole thing is n log n. So n log n is a function you may not have a lot of intuition with. You may not have seen it enough to kind of know what it's curve looks like, but if you remember how the logarithmic curve looks, right? Which is a very slow growing almost flat line and then n being linear it – the kind of combination of the two it's called the linear rhythmic term here is just a little bit more than linear. Not a lot more, but it grows a little bit more steeply than the state of linear was, but not nearly, right? As sharply as something that's quadratic.

So if we look at some times and let selection sort compared to merge sort, right? On an input of 10,000, right? Took a fraction, right? To do a merge sort than it did to do a selection sort and as it grows, right? So if we go from 20,000 to 50,000 we've a little bit more than doubled it that the merge sort times, right? Went up a little bit more than a factor of two in growing in these things. Not quite doubling. A little bit more than doubling, right? Because of the logarithmic term that's being added there, but growing slowly enough that you can start imaging using a sort like merge sort on an input of a million elements in a way that you cannot on selection sort, right?

Selection sort – my estimate I did not run it to find this out, right? Based on the early times I can predict it'll be about eight hours for it to sort a million elements, taking just a few seconds, right? For merge sort to do that same input. So a very big difference, right? From the n square to n log n that makes it so that if you have a sufficiently large data set, right? You're gonna have to look to an n log n sort where an n squared sort would just not work out for you. I'll mention here that actually n log n is the theoretical boundary for what a general purpose sort algorithm can do. So that our search from here will have to be a quest for perhaps an n log n that competes with merge sort and maybe exceeds it in some ways by having lower constant factors to it, but we won't get a better big O for a general purpose sorting algorithm.

If we have to sort kind of any amount of data and any permutation we just – and it has to work for all cases, then n log n is the best we can do in terms of a big O. Let me show you these guys kind of run a little race because there's nothing more important than having a reason to bet in class. So if I put insertion and selection and merge sort all up against one another. Let's turn off the sound. I really don't want to hear it all go. Okay. So merge sort just really smoking and insertion sort and selection sort not giving up early, but definitely doing a lot more work. So if you look at the numbers here in terms of comparisons and moves that are being done in the merge sort case, right? So I have 500 elements here is substantially less, right? Than the quadratic terms that we're seeing on the comparison and move counts, right? For us in selection sort and insertion sort.

So this is still on something fairly small, right? Five hundred elements, right? That you get into the thousands and millions, right? The gap between them just widens immensely. Does merge sort make any good sense out of things being all ready sorted? So thinking about what you know about the algorithm, does the fact that it's mostly sorted or all ready sorted provide an advantage to the way merge sort works? Nope, nope. Just not at all, right? In fact, I can make the data actually totally sorted for that matter, right? And say go ahead and sort and see what happens, right? And it's like insertion sort did 511 comparisons, zero moves, realized everything was in sorted order and finished very quickly.

Merge sort still did a lot of work, thousands of comparison moves. It did a slightly fewer number of compares than it would typically do. That's because when it, for example, divided into the left half and the right half it had all the smaller elements on the left, all the larger elements on the right, and it will sit thee and compare to realize that all of the left elements go first. Then it'll kind of just dump the remaining ones on. So it actually shaves off a few of the comparisons in the merge step, but it doesn't really provide any real advantage. It still kind of moves them away and moves them back and does all the work. And then selection sort just still taking its favorite amount of time, which is, yeah, I'll look at everything. That might be the smallest, but I'm not taking any chances. I'm gonna look through all of them to make sure before I decide to keep it where it was.

If I put them in reverse order, just because I can, watch insertion sort bog down into it's worst case and that gives selection sort a chance to, like, show it's metal and there selection sort showing okay, well, you give me my absolute worst input I definitely do have a little bit of a hard time with it. But merge sort still just doing its thing. What is gonna make all the sound go on? Just because we can. Because we have two minutes and I'm not gonna sort quick sort. Ah, you're going. They're duking it out. Oh, no. That sounds like a cartoon, sort of, like a race. All right. I could watch this thing all day. In fact, I often spend all day. I show this to my kids. I'm still waiting for them to come up with the next great sort algorithm, but so far really it's not their thing.

So I will give you a little clue about what we're gonna do next. We're gonna talk about a different recursive algorithm. Same divide and conquer strategy kind of overall, but kind of taking a different tactic about which part to make easy and which part to make hard. So quick sort, right? As I said, is this easy split hard join. We divided them into half using sort of no smart information whatsoever and then all of the work was done in that join. I’ve got these two sorted piles. I’ve gotta get them back into order. How do I work it out? Well, the quick sort algorithm is also recursive, also kind of a split join strategy, but it does more work up front that’s not even in the split phase.

If I kind of did a more intelligent division into two halves then I could make it easier on me in the join phase. And it’s strategy for the split is to decide what’s the lower half and what’s the upper half. So if I were looking at a set of test papers I might put the names A through M over here. So go through the entire pile and do a quick assessment of are you in the upper half or the lower half? Oh, you’re in the lower, you’re in the upper, these two go in the lower, these two go in the upper. I examine all of them and I get all of the A through M’s over here and all of the N through Z’s over there and then I recursively sort those.

So I get the A to M’s all worked out and I get the N through the Z’s all worked out. Then the task of joining them is totally trivial, right? I’ve got A through M. I’ve got N through Z. Well, you just push them together and there’s actually no comparing and looking and merging and whatnot that’s needed. That join step is where we get the benefit of all of the work we did in the front end. That split step though is a little hard. So we’ll come back in on Friday and we’ll talk about how to do that split step and then what is some of the consequences of our strategy for that split step and how they come back to get at us, but that will be Friday. There will be music. There will be dancing girls.

[End of Audio]

Duration: 50 minutes