

Instructor (Julie Zelenski): There must be endless toil and travail. Let's ask another question. So [inaudible] was the biggest thing he had done so far. [Inaudible] on the trail. Not really anything new, but definitely the prospect of getting it all working and bigger aspects of its own [inaudible] challenges.

So I expect to take a little bit more, but I'm always open to figure out what really happened there. So let's, we'll start with something ridiculous, but just in case somebody was just super hot, did anybody manage to finish [inaudible] with less than ten hours, a total start to finish – hey I like to see that.

That's a very good sign. Ten to fifteen – a very healthy sized group there. Fifteen to 20 – there's some people there. More than 20; okay, good, good and hopefully when you got done you felt like, "Okay, I have accomplished something good," but there's something really satisfying about writing a program.

The best thing you can't do as well as the program you've just written, the idea that taking in your own brain and cracking into it to write a program that can actually play a game and beat you is really a pretty neat little twist on things.

So, hopefully that was an enjoyable and satisfyable place to get to right before the midterm here. We're gonna take a little break in assignments meaning I'm not giving you an assignment now because what I really want you to spend your weekend doing is enjoying the sunshine for five-ten minutes at a time and then prepping yourself for the midterm which is coming up Tuesday night.

So we don't have class on Monday and so the next time I'll see you is Tuesday, 7:00 to 9:00 over in [inaudible] Auditorium and certainly if you have questions that need answered, sending an email over the weekend is totally fine, and we're showing up in the Lair; we'll have Lair hours on Monday evening [inaudible].

Student: [Inaudible].

Instructor (Julie Zelenski): No, we're not. Try again. Okay, now we got my phone. So I will probably put up [inaudible] in advance of the midterms, so if you actually wanted to pick it up before the midterm or right after the midterm to get started on it, and it will be due the following week and it's a little bit lighter of a week than [inaudible] was just in terms of planning.

What we're going to pick up today is we're going to talk Quick Sort which is the last of the sorting algorithms I want to talk through and then I'm actually going to go through the process of picking one of the sources, in this case [inaudible] and turning it into a fully generic sorting template as kind of capitalizing on the futures of C++. So the material here is the last sections of Chapter 7, a little bit of material pulled out of Chapter 11, which is the client callback stuff, is there.

I was actually totally planning on going to the café today until I got a call from my son's preschool right before I came to class that says he has swollen lymph nodes and a fever, so I've got to go get a little guy after class, so I'm sorry that won't happen today, but hopefully we'll be back on track a week from today.

Okay. So this is the last thing we had talked about at the end of the merge sort was comparing a quadratic and N-squared sort, this left and right sort right here to the linear arrhythmic which is the N-log in kind that merge sort runs in, right, showing you that really way out-performing even on small values and getting the large values [inaudible] just getting enormous in terms of what you can accomplish with a N-log-in algorithm really vastly superior to what you're gonna get out of a N-squared algorithm.

I said well, N-log-I told you without proof that you're going to take on faith that I wouldn't lie to you, that that is the best we can do in terms of a comparison based sort, but what we're going to look for is a way of maybe even improving on those kinds of merge sort by kind of a one thing we might want to do is avoid that copying of the data out and back that merge sort does and maybe have some lower cost in factors in the process, right that can bring our times down even better.

So the sort we're looking at is quick sort. And I think if you were gonna come up with an algorithm name, naming it Quick Sort becomes good marketing here so it kind of inspires you to believe actually that it's gonna live up to its name, is a divide-and-conquer algorithm that takes a strategy like merge sort in the abstract which is the divide into two pieces and [inaudible] sort those pieces and then join them back together.

But whereas merge sort does an easy split hard join, this one flips it on its head and said what if we did some work on the front step in the splitting process and then that may give us less to do on the joining step and so the strategy for quick sort is to run to the pile of papers, whatever we're trying to work at in a partitioning step is the first thing that it does and that's the splitting step and that partitioning is designed to kind of move stuff to the lower half and the upper half.

So having some idea of what the middle most value would be, and then actually just walking through the pile of papers and kind of you know, distributing to the left and right portions based on their comparison to this middle-most element.

Once I have those two stacks – I've got A through M over here and N through Z over there, that we're cursively sorting those takes place, kind of assuming my delegates do their work and in the process of re-joining them is really kind of simple. They kind of, you know, joined together with actually no extra work, no looking, no process there.

So the one thing that actually is is where all the trickiness comes into is this idea of how we do this partitioning. So I was being a little big disingenuous when I said well if I had this stack of exam papers and I know that the names, you know range over the alphabet A through Z then I know where M is and I can say well, that's a good mid pint around to divide.

Given that we want to make this work for kind of any sort of data, we're not going to [inaudible] know, what's the minimal most value. If we have a bunch of numbers, are they test scores, in which case maybe 50 is a good place to move. Are they instead, you know, population counts of states in which case millions would be a better number to pick to kind of divide them up.

So we have to come up with a strategy that's gonna work, you know, no matter what the input that's somehow gonna pick something that's kind of reasonable for how we're gonna do these divisions. There's this idea of picking, called a pivot. So given this, you know, region, this set of things to sort, how do I know what's a reasonable pivot. We're looking for something that's close to the median is actually ideal.

So if we could walk through them and compute the median, that would be the best we could do, we're gonna try to get around to not doing that much work about it, though. We're gonna actually try to kind of make an approximation and we're gonna make a really very simplistic choice about what my approximation would be. We're gonna come back and re-visit this a little bit later.

But I want to say, "Well, I just took the first paper off the stack. If they're in random order, right. I look at the first one; I say Okay, it's, you know, somebody king. Well, everybody less than king goes over here. Everybody greater than king goes over there.

Now king may not be perfectly in the middle and we're gonna come back to see what that will do to the analysis, but at least we know it's in the range of the values we're looking at, right, and so it at least did that for us. And it means no matter what our data is, we can always use that. It's a strategy that will always work. Let's just take the thing that we first see and use it to be the pivot and then slot them left and right.

So let me go through looking at what the code actually does. This is another of those cases where the code for partition is a little bit frightening and the analysis we're looking at is not actually to get really really worried about the exact details of the less than or the less than or equal to. I'm gonna talk you through what it does, but the more important take-away point is what's the algorithm behind Quick Sort. What's the strategy it's using to divide and conquer and how that's working out.

So the main Quick Sort algorithm looks like this. We're given this vector of things we're trying to sort and we have a start index and a stop index that identifies the sub region of the vector that we're currently trying to sort. I'm gonna use this in subsequent calls to kind of identify what piece we're recursively focusing on. As long as there's a positive difference between the start and the stop so there's at least two elements to sort, then we go through the process of doing the division and the sorting.

So then it makes the call to partition saying sub-region array do the partition and we'll come back and talk about that code in a second and the thing we're trying to find the partition is the index at which the pivot was placed.

So after we did all the work, it moved everything less than the pivot to the left side of that, everything greater than the pivot to the right side and then the pivot will tell us where the division is and then we make two recursive calls to sort everything up to but not including pivot, to sort everything past the pivot to the end, and then those calls are operating on the array in place, so we're not copying it out and copying it back. So in fact, there actually even isn't really any joint step here.

We said sort the four things on the left, sort the seven things on the right and then the joining of them was where they're already where they need to be so in fact we don't have anything to do in the joined wall.

The tricky part is certainly in partition. The version of the partition we're using here is one that kind of takes two indices, two fingers, you might imagine and it walks a one end from the stop position down and one from the start position over and it's looking for elements that are on the wrong side.

So if we start with our right hand finger on the very end of the array, that first loop in there is designed to move the right hand down looking for something that doesn't belong in left side, so we identified 45 as the pivot value. It says find something that's not greater than 45. That's the first thing we found coming in from the right downward that doesn't belong, so only that first step.

So it turns out it takes it just one iteration to find that, and so okay, this guy doesn't belong on the right-hand side. Now it does the same thing on the inverse on the left-hand side. Try to find something that doesn't belong on the left-hand side.

So the left hand moves up. In this case, it took two iterations to get to that, to this 92. So now we have a perfect opportunity to fix both our problems with one swap, exchange what's at the current position of left hand with right hand and now we will have made both of our problems go away and we'll have kind of more things on the left and the right that belong there and then we can walk inward from there, you know, to find subsequent ones that are out of order.

So those two get exchanged and now right again moves into find that 8, left moves over to find that 74. We swap again. And as we just keep doing this, right, until they cross, and once they've cross, right then we know that everything that the right scanned over is greater than the pivot, everything in the left was less than and at that point, right we have divided it, right, everything that's small is kind of in the left side of the vector. Everything that's large in the right side.

So I swap these two and now I get to that place and I say okay, so now big things here, small things there. The last thing we do is we move the pivot into the place right between them and know that it is exactly located right in the slot where they cross.

Now I have two sub arrays to work on. So the return from partition in this case will be the index four here and it says Okay, go ahead and quick sort this front-most part and then

come back and do the second part. So in recursion, kind of think of that as being postponed and it's waiting; we'll get back to it in a minute, but while we work on this step, we'll do this same thing.

It's a little bit harder to see it in the small kind of what's going on, but in this case, right, it divided that because the pivot was 8 into 8 and the three things greater than the pivot. In this case, the 41 is the pivot so it's actually going to move 41 over there. It's going to have the left of that as it keeps working its way down, it gets smaller and smaller and it's harder to see the workings of the algorithm in such a small case of it rearranging it, but we'll get back to the big left half in a second.

And so now that that's all been sorted, we're revisiting the side that's advancing above the pivot to get these five guys, six guys in the order, taking the same strategy. You pivot around 67 [inaudible] doing some swapping [inaudible] and we'll see it in slightly bigger case to kind of get the idea, but very quickly there's something very interesting about the way quick sort is working is that that partition step very quickly gets things kind of close to the right place.

And so now that we've done both the left and the right we're done. The whole thing is done. Everything kind of got moved into the right position as part of the recursive, part of the sorting and doesn't need to be further moved to solve the whole problem.

That first step of the partition is doing a lot of the kind of throwing things kind of left and right, but it's actually quickly moving big things and small things that are out of place closer to where they're gonna eventually need to go, and it turns out to be a real advantage in terms of the running time of this sort because it actually is doing some quick movement close to where it needs to be and that kind of fixes it up in the recursive calls that examine the smaller sub sections of the [inaudible].

Let me show you what it sounds like, because that's really what you want to know. So let's – it's gonna kind of [inaudible]. So you can see the big partitioning happening and then it kind of jumbling things up and then coming back to revisit them, but you notice that quite quickly kind of all the small ones kind of got thrown to the left all.

All the large elements to the right and then the kind of process of coming back and so the big partition that you can see kind of working across them and then kind of noodling down. If I turn the sound on for it and I'll probably take it down a little bit.

So the big partition steps making a lot of noise, right and moving things kind of quickly and it almost appears, you know, to hear this kind of random sort of it's hard to identify what's going on during the big partition, but then as you hear it make its way down the recursive tree it's focusing on these smaller and smaller regions where the numbers have all been gathered because they are similar in value.

And you hear a lot of noodling in the same pitch range region as its working on the smaller and smaller sub arrays and then they definitely go from low to high because of

the recursion chooses the left side to operate on before the right side that you hear the work being done in the lower tones before it gets to the finishing up of the high tones. Kinda cool.

Let us come back here and talk about how to analyze this guy a little bit. So the main part of the algorithm is very simple and deceptively simple because all the hard work was actually done in partition.

The partition step is linear and if you can kind of just go along with me conceptually, you'll see that we're moving this left finger in; we're moving this right finger in and we stop when they cross, so that means every element was looked at. Either they both matched over here and they matched over here, but eventually they let you look at every element as you worked your way in and did some swaps along the way.

And so that process is linear and the number of elements. There's a thousand of them kind of has to advance maybe 400 here and 600 there, but we'll look at all the elements and the process of partitioning them to the lower or upper halves.

Then it makes two calls, quick sort to the left and the right portions of that. Well, we don't know exactly what that division was, was it even, was it a little lop sided and that's certainly going to come back to be something we're gonna look at.

If we assume kind of this is the ideal 50/50 split sort of in an ideal world if that choice we had for the pivot happened to be the median or close enough to the median that effectively we got an even division, at every level of recursion, then the recurrence relation for the whole process is the time required to sort, an input of N is in to partition it and then 2, sorting two halves that are each half again as big as the original input.

We saw that recurrence relationship for merge sort; we solved it down. Think in terms of the tree, right you have the branching of three, branching of two, branching of two and at each level the partitioning being done across each level and then it going to a depth of $\log_2 N$, right, the number of times you can [inaudible] by two [inaudible] single case arrays that are easily handled.

So it is an $N \log N$ sort in this perfect best case. So that should mean that in terms of Big O growth curves, right, merge sort and quick sort should look about the same. It does have sort of better factors, though. Let me show you. I go back here and I just run them against each other.

If I put quick sort versus merge sort here, stop making noise, the quick sort pretty handily managed to beat merge sort in this case, merge sort was on the top, quick sort was underneath and certainly in terms of what was going on it was doing, looks like more comparisons right, to get everything there, that partition step did a lot of comparison, but fewer moves, and that kind of actually does sort of make intuitive sense.

You think that quick sort very quickly moves stuff to where it goes and does a little bit of noodling. Merge sort does a lot of moving things away and moving them back and kind of like as you see it growing, you'll see a lot of large elements that get placed up in the front, but then have to be moved again right, because that was not their final resting place.

And so merge sort does a lot of kind of movement away and back that tends to cost it overall, right. You know, a higher constant factor of the moves than something like quick sort does where it more quickly gets to the right place and then doesn't move it again.

So that was all well and good. That was assuming that everything went perfectly. That was given our simplistic choice of the pivot, you can imagine that's really assuming a lot right that went our way. If we look at a particularly bad split, let's imagine that the number we have to pick is pretty close to, you know, one of the extremes.

If I were sorting papers by alphabet and if the one on top happened to be a C, right, well then I'm gonna be dividing it pretty lop sided, right, it's just gonna be the As and the Bs on one side and then everything [inaudible] to the end on the other side.

If I got a split that was about 90/10, ninety percent went on one side, ten percent on the other, you would expect right for it to kind of change the running time of what's going on here, so I get one tenth of them over here and the low half and maybe be nine-tenths in the right half. Let's just say every time it always takes nine-tenths and moves it to the upper half, so I kept picking something that's artificially close to the front.

These like will kind of peter out fairly quickly diving N by 10 and 10 and 10 and 10, eventually these are gonna peter out very soon, but this one arm over there where I keep taking 90 percent of what remains.

I had 90 percent, but I had 81 percent. I keep multiplying by nine-tenths and I'm still kind of retaining a lot of elements on this one big portion, that the depth of this tree isn't gonna bottom out quite as quickly as it used to, but it used to the number of times you could divide N by 2, the log based 2 of N was where we landed.

In this case, I have to take N and multiply it by nine-tenths so instead of one half, right, to the K, it's nine tenths to the K and it's like how many iterations how deep will this get before it gets to the simple case and so solving for N equals ten-ninths to the K is taking the log-based ten-ninths of both sides, then K, the number of levels is the log-based ten-ninths of them.

We're kind of relying on a fact of mathematics here though is that all algorithms are the same value, so log-based A of N and log-based B of N, they only differ by some constant that you can compute if you just rearrange the terms around. So in effect this means that there is a constant in front of this. It's like a constant difference, the ratio between ten-ninths and 2 that distinguishes these, but it still can be logged based 2 of N in terms of Big O, right, where we can throw away those constants.

So even those this will perform, obviously in, you know, experimentally you would expect that getting this kind of split would cause it to work more slowly than it would have in a perfect split situation. The Big O [inaudible] is still gonna look in-log-in arrhythmic and [inaudible].

So that was pretty good to know. So it makes you feel a little bit confident, like sometimes it might be getting an even split and sometimes it might be getting one-third, two-thirds, sometimes it might be getting you know, nine-tenths, but if it was kind of in the mix still dividing off some fraction is still doing okay.

Now, let's look at the really, really, really worst case split. The really, really worst case split right, it didn't even take off a fraction. It just managed to separate one, but somehow the pivot here did a really terribly crummy job, right, of dividing it at all that in effect, all the elements were greater than the pivot or less than the pivot since they all landed on one side and the pivot was kind of by itself.

So starting with an input of size N , we sectioned off 1 and we had N minus 1 remaining. Well, the next time, we sectioned off another one, so this happened again and again and again. We just got really bad luck all the way through. Each of these levels is only making progress for the base case at a snails pace. Right, one element is being subtracted each subsequent level and that's gonna go to a depth of N .

And so now if we're doing N work across the levels, it isn't quite N work because in fact, some of these bottom out. It's still the Gaussian sum in the end is that we are got N levels doing N work each. We suddenly have what was a linear arrhythmic algorithm now performing quadratically. So in the class the selection sort, inscription sort in its worst-case behavior.

So quite a range whereas merge sort is a very reliable performer, merge doesn't care. No matter what the order is, no matter what's going on, it's the same amount of work, it means that there's some inputs that can cause quick sort to vary between being linear arrhythmic or being quadratic, and there's a huge difference as we saw in our earlier charts about how those things will run for large values.

So what makes the worst case – given how we're choosing a pivot right now is to take the first thing in the sub section to look at as the pivot, what's the example input that gives you this?

Student:Sorted.

Instructor (Julie Zelenski):If it's already sorted. Isn't that a charmer? Here's a sorting algorithm. If you ask it to do something and in fact if you accidentally [inaudible] twice, you already had sorted the data and you said, "Oh, you did something," and you passed it back to the sort, it would suddenly actually degenerate into its very worst case.

It's already sorted, so it would say, "I've got this stack of exam papers." I look at the first one and I say, "Oh, look it's Adam's." And I say, "Okay, well, let me go find all the ones that belong in the left side." None of them do. I said, "Okay, well put Adam's over here." I'll [inaudible] sort that.

That was easy. Oh, let me look at what I've got left, oh with Baker on the top. Okay, well let me put Baker by itself, but could we find the other ones? Oh, no, no more. It would just you know, continue because I'm doing this thing, looking at all N , looking at all N minus 1, looking at N minus 2, all the way down to the bottom making no, recognizing nothing about how beautifully the data already was arranged and you know, getting its worst case.

There's actually a couple of others that come in, too. That's probably the most obvious one to think of is it's coming in in increasing order. It's also true if it's in decreasing order. If I happen to have the largest value on the top, then I'll end up splitting it all into, everything to the left side and nothing to the right.

There are actually other variations that will also produce this. If at any given stage that we're looking at a sub array, if the first value happens to be the extreme of what remains whether it's the small to the large, and it could alternate, which would be kind of a funny pattern, but if you have the tallest and the shortest and the tallest and then another tallest and then a shortest, so those would look a little bit harder to describe, but there are some other alternatives, that produce this bad result.

All of these we could call degenerate cases. There's a small number of these relative to the N factorial [inaudible] your data could come in. In fact, there was a huge number, and so there's lots and lots of ways it could be arranged. There's a very small number of them that would be arranged in such a way to trigger this very, very worst-case behaviors.

Some are close to worst case, like if they were almost sorted, they might every now and then get a good split, but mostly a bad split, but the number that actually degenerate to the absolute worst case is actually quite small, but the truth is do we expect that those outputs are somehow hard to imagine us getting.

Are they so unusual and weird that you might be willing to say it's okay that my algorithm has this one or a couple of bad cases in it because it never comes up.

In this case, the only thing that your data came in sorted or almost sorted or reverse sorted is probably not unlikely. It might be that you know, you happen to be reading from a file on disc and somebody has taken the [inaudible] and sort it before they gave the data to you. If all the sudden that caused your program to behave really badly, that would really be a pretty questionable outcome.

So let's go aback and look at just to see, though. It's kind of interesting to see it happening in – this is against quick sort versus merge sort. If I change the data to be partially sorted and let it run again, if I still manage to beat it doing a little bit more, if I

change it to totally sorted or reverse sorted, for that matter, and so they look like they're doing a lot of work, a lot of work about nothing.

And you can see that quick sort really is still way back there. It has looked at the first 10 percent or so and is doing a lot of frantic partitioning that never moves anything. And visibly kind of traipsing it's way up there. Merge sort meanwhile is actually on the beach in Jamaica with a daiquiri mocking quick sort for all those times that it had said it was so much better than it was. "Duh, it was already sorted; you Doofus." Apparently it wasn't listening.

Merge sort almost looked like it did nothing and that's because it took the left half, which is the smallest half, moved it off, right, kind of copied it back and it ends up copying each element back to the same location it was already originally present in. There you go, quick sort taking its time and if I ran it against, let's say, insertion sort and selection sort, why not [inaudible] nothing better to do than to sort for you all day long.

So there insertion sort actually finishing very early because it does nothing. It sort of looked at them all and said they were already in the right order, merge sort doing a little bit more work to move everything back and forth in the same place.

But in this case, selection sort and quick sort here at the bottom and selection sort here at the top doing really roughly about the same work if you look at the sum total of the comparisons and moves and then time spent on those suddenly shows that yeah, in this input situation quick sort is performing like an N^2 sort, and obviously very similar constant factors to those present in selection sort, so really behaving totally quadratically in that worst-case input.

If I make it reverse sorted, it's a little more interesting because you can kind of see something going on. Everybody kind of doing their thing. Insertion sort now kind of hitting its worst case, so it actually coming in dead last because it did so much extra moving, but still showing the kind of quadratic terms, right, that selection sort and quick sort are both having, but higher constant factors there, so taking almost twice as long because actually doing a little bit extra work because of that inverted situation.

[Inaudible] something big, just cause. We'll put it back into random, back into full speed and kind of watch some things happen. So quick sort right, just sort of done in a flash, merge sort finishing behind it and ordinarily the quadratic sorts taking quite a much longer time than our two recursive sorts.

But it's the random input really buying quick sort its speed. Is it gonna win – oh, come on. Oh, come on. Don't you want selection sort to come behind? I always wanted it to come from behind. It's kind of like rooting for the underdog. Yes.

And yet, just remember don't mock insertion sort. What it sorted is the only one that recognizes it and does something clever about it. [Inaudible] sort is anti-clever about it, so it still can hold its head up and be proud. This kind of [inaudible].

I was thinking about like what algorithms, right, there's a reason why there's four that we talked about and there's actually a dozen more. It's like different situations actually produce different outcomes for different sorts and there are reasons that even though they might not be the best sort for all purposes, there are some situations where it might be the right one, so if you had it, as data said that you expect it to be mostly sorted, but had a few [inaudible] in it.

Like if you had a pile of papers that were sorted and you dropped them on the floor and they got scuffed up a little bit, but they're still largely sorted, insertion sort is the way to go. It will take advantage of the work that was already done and not redo it or create kind of havoc the way quick sort would.

But quick sort, the last thing we need to tell you, though, about it is we can't tolerate this. Like the way quick sort is in its classic form with this first element as pivot would be an unacceptable way to do this algorithm. So quick sort actually typically is one of the most standard elements that's offered in a library of programming languages, the sort element.

Well, it has to have some strategy for how to deal with this in a way that does not degenerate and so the idea is, what you need to do is just pick a different choice for the pivot, a little bit more clever, spend a little bit more time, do something that's a little less predictable than just picking that first most element.

So to take it to the far extreme, one thing you could do is just compute the median, analyze the data and compute the median. It turns out there is a linear time algorithm for this that would look at every element of the data set once and then be able to tell you what the median was and that would guarantee you that 50/50 split.

So if you go find it and use it at every stage, you will guarantee it. Most algorithms don't tend to do that. That's actually kind of overkill for the problem. We want to get it to where it's pretty much guaranteed to never get the worst case.

But we're not concerned with it getting 50/50. It's got 60/40 or something close enough, that actually, you know, 60/40, 70/30 and it was bopping around in that range it'd be fine, so the other two that are much more commonly used is some kind of approximation of the median with a little bit of guessing or something in it.

For example, median of three takes three elements and typically it takes three from some specified position, it takes the middle most element, the last element and the front element and it says, "Okay, given those three, we arrange them to find out what's the middle of those three, and use that."

If the data was already sorted, it turns out you've got the median, right because it wasn't the middle most. If data was just in random position, then you've got one that you know, at least there's one element on one side, right, and so the odds that that every single time would produce a very bad split is pretty low. There are some inputs that could kind of get it to generate a little bit, but it's pretty foolproof in most ordinary situations.

Even more unpredictable would be just choose a random element. So look at the start to stop index you have, pick an random there, flop in to the front and now use it as the pivot element. If you don't know ahead of time how your random number [inaudible] it's impossible to generate an input and force it into the worst-case behavior.

And so the idea is that you're kind of counting on randomness and just the probabilistic outcome if it managing to be such that the way it shows the random element was to pick the extreme and everything, it is just impossible, the odds are astronomically against it, and so it will, a very simple fix, right that still leaves the possibility of the worst case in there, but you know, much, much, much far removed probability sense.

So, just simple things, right, but from there the algorithm operates the same way it always has which is to say, "Pick the median, however, you want to do it, move it into that front slot and then carry on as before in terms of the left and the right hand and moving down and swapping and recursing and all that [inaudible]."

Any questions; anything about sorting, sorting algorithms? Why don't we do some coding actually, which is the next thing I want to show you because once you know how to write sorts, sort of the other piece I think you need to have to go with it is how would I write a sort to be generally useful in a large variety of situations that knowing about these sorts I might decide to build myself a really general purpose, good, high performance, quick sort, but I could use again, and again and again.

I want to make it work generically so I could sort strings, I could sort numbers, or I could sort students, or I could sort vectors of students or whatever, and I'd want to make sure that no matter what I needed to do it was gonna solve all my problems, this one kind of sorting tool, and we need to know a little bit of C++ to make that work by use of the function template.

So if you want to ask about sorting algorithms, now is the time.

Student:[Inaudible].

Instructor (Julie Zelenski):We don't bubble sort, which is kind of interesting. It will come up actually in the assignment I give you next week because it is a little bit of an historical artifact to know about it, but it turns out bubble sort is one of those sorts that is dominated by pretty much every other sort you'll know about in every way, as there's really nothing to recommend it.

As I said, each of these four have something about them that actually has a strength or whatever. Bubble sort really doesn't. It's a little bit harder to write than insertion sort. It's a little bit slower than insertion sort; it's a little bit easier to get it wrong than insertion sort.

It has higher constant factors. You know, it does recognize the data and sort order, but so does insertion sort, so it's hard to come up with a reason to actually spend a lot of time on

it, but I will expose you to it in the assignment because I think it's a little bit of a – it's part of our history right, as a computer science student to be exposed to. Anything else?

I'm gonna show you some things about function templates. Oh, these are some numbers; I forgot to give that. Just to say, in the end, right, which we saw a little bit in the analysis that the constant factors on quick sort are noticeable when compared to merge sort by a factor of 4, moving things more quickly and not having to mess with them again.

Student: Quick sort [inaudible] in totally random order?

Instructor (Julie Zelenski): Yes, they are. Yes –

Student: So it doesn't have them taking –

Instructor (Julie Zelenski): So this is actually using a classic quick sort with no degenerate protection on random data. If I had put it in, sorted it in on that case, you would definitely see like numbers comparable to like selection sorts, eight hours in that last slot, right.

Or you [inaudible] degenerate protection, and it would probably have an in the noise a small slow down for that little extra work that's being done, but then you'll be getting in log and performance reliable across all states of inputs.

Student: So [inaudible] protection as it starts to even out time wise with merge sort, does it still –

Instructor (Julie Zelenski): No, it doesn't. It actually is an almost imperceptible change in the time because it depends on which form of it you use because if you use, for example the random or median of three, the amount of work you're doing is about two more things per level in the tree and so it turns out that, yeah, over the space of log in it's just not enough to even be noticeable.

Now if you did the full median algorithm, you could actually start to see it because then you'd see both a linear partition step and a linear median step and that would actually raise the cause and factors where it would probably be closer in line to merge sort. Pretty much nobody writes it that way is the truth, but you could kind of in theory is why we [inaudible].

What I'm going to show you, kind of motivate this by starting with something simple, and then we're gonna move towards kind of building the fully generic, you know, one algorithm does it all, sorting function.

So what I'm gonna first look at is flop because it's a little bit easier to see it in this case, is that you know, sometimes you need to take two variables and exchange their values.

I mean you need to go through a temporary to copy one and then copy the other back and what not – swapping characters, swapping ends, swapping strings, swapping students, you know, any kind of variables you wanted to swap, you could write a specific swap function that takes two variables by reference of the integer, string or character type that you're trying to exchange and then copies one to a temporary and exchanges the two values.

Because of the way C++ functions work, right, you really do need to say, "It's a string, this is a string, you know, what's being declared here is a string," and so you might say, "Well, if I need more than one swap, sometimes that swaps strings and characters, you know, my choices are basically to duplicate and copy and paste and so on. That's not good; we'd rather not do that.

But what I want to do is I want to discuss what it takes to write something in template form. We have been using templates right, the vector is a template, the set is a template, the stack and queue and all these things are templates that are written in a very generic way using a placeholder, so the code that holds onto your collection of things is written saying, "I don't know what the thing is; is it a string; is it an N; is it a car?"

Well, vectors just hold onto things and you as a client can decide what you're gonna be storing in this particular kind of vector. And so what we're gonna see is if I take those flat functions and I try to distill them down and say, "Well, what is it that any swap routine looks like?" It needs to take two variables by reference, it needs to declare a template of that type and it needs to do the assignment all around to exchange those values.

Can I write a version that is tight unspecified leaving a placeholder in there for what – that's really kind of amazing, that what we're gonna be swapping and then let there be multiple swaps generated from that one pattern.

So we're gonna do this actually in the compiler because it's always nice to see a little bit of code happening. So I go over here and I got some code up there that I'm just currently gonna [inaudible] because I don't want to deal with it right now. We're gonna see it in a second. It doesn't [inaudible].

Okay, so your usual swap looks like this. And that would only swap exactly integers. If I wanted characters I'd change it all around. So what I'm gonna change it to is a template. I'm gonna add a template header at the top and so the template header starts with a keyword template and then angle brackets that says, "What kind of placeholders does this template depend on?"

It depends on one type name and then I've chosen then name capital T, type for it. That's a choice that I'm going to make and it says in the body of the function that's coming up where I would have ordinarily fully committed on the type, I'm gonna use this placeholder capital T, type instead.

And now I have written swap in such a way that it doesn't say for sure what's being exchanged, two strings, two Ns, two doubles. They're all have to be matching in this form, and I said, "Well, there's a placeholder, and the placeholder was gonna be filled in by the client who used this flop, and on demand, we can instantiate as many versions of the swap function as we need."

If you need to swap integers, you can instantiate a swap that then fills in the placeholder type with Ns all the way through and then I can use that same template or pattern to generate one that will swap characters or swap strings or swap whatever.

So if I go down to my main, maybe I'll just pull my main up [inaudible]. And I will show you how I can do this. I can say int 1 equals 54, 2 equals 32 and I say swap 1-2. So if the usage of a function is a little bit different than it was for class templates, in class templates we always did this thing where I said the name of the template and then it angled back, as I said.

And in particular I want the swap function that operates where the template parameter has been filled in with int that in the case of functions, it turns out it's a little bit easier for the compiler to know what you intended that on the basis of what my arguments are to this, is I called swap passing two integers, it knows that there's only one possibility for what version of swap you were interested in, that it must be the one where type has been filled in with int and that's the one to generate for you.

So you can use that long form of saying swap, angle bracket int, but typically you will not; you will let the compiler infer it for you on the usage, so based on the arguments you passed, it will figure out how to kind of match them to what swap said it was taking and if it can't match them, for example, if you pass one double and one int, you'll get compiler errors.

But if I've done it correctly, then it will generate for me a swap where type's been filled in with int and if I call it again, passing different kinds of things, so having string S equals hello and T equals world, that I can write swap S and T and now the compiler generated for me two different versions of swap, right, one for integers, one for strings on the [inaudible] and I didn't do anything with them. I put them [inaudible], so nothing to see there, but showing it does compile and build up.

That's a pretty neat idea. Kind of important because it turns out there are a lot of pieces of code that you're gonna find yourself wanting to write that don't depend, in a case like swap, swap doesn't care what it's exchanging, that they're Ns, that they're strings, that they're doubles. The way you swap two things is the same no matter what they are.

You copy one aside; you overwrite one; you overwrite the other and the same thing applies to much more algorithmically interesting problems like searching, like sorting, like removing duplicates, like printing, where you want to take a vector and print all of its contents right that the steps you need to do to print a vector of events looks just like the steps you need to do to print a vector of strings.

And so it would be very annoying every time I need to do that to duplicate that code that there's a real appeal toward writing it once, as a template and using it in multiple situations.

So this shows that if I swap, right, it infers what I meant and then this thing basically just shows what happened is when I said int equals four, [inaudible] use that pattern to generate the swap int where the type parameter, that placeholder has been filled in and established that it's int for this particular version which is distinct from the swap for characters and swap for strings and what not.

So let me show you that version I talked about print, and this one I'm gonna go back to the compiler for just a second. Let's say I want to print a vector, printing a vector it like iterating all the members and using the screen insertion to print them out and so here it is taking some vector where the elements in it are of this type name. In this case, I've just used T as my short name here.

The iterator looks the same; the bracketing looks the same. I see the end now and I'd like to be able to print vectors of strings, vectors of ints, vector of doubles with this one piece of code. So if I call print vector and I pass code vector events, all's good; vector doubles, all's good; vector strings, all's good.

Here's where I could get a little bit into trouble here. I've made this [inaudible] chord that has an X and a Y field. I make a vector that contains these chords. And then I try to call print vector of C. So when I do this, right, it will instantiate a versive print vector where the T has been matched up to, oh it's chords that are in there; you've got a vector of chords.

And so okay, we'll go through and iterate over the sides of the vector and this line is trying to say see out of a chord type. And struts don't automatically know how to out put themselves onto a string, and so at this point when I try to instantiate, so the code in print vector is actually fine and works for a large number of types, all the primitive types will be fine, string and things like that are fine.

But if I try to use it for a type for which it doesn't have this output behavior, right I will get a compiler error, and it may come as a surprise because a print vector appeared to be working fine in all the other situations and all the sudden it seems like a new error has cropped up but that error came from the instantiation of a particular version in this case, that suddenly ran afoul of what the template expected of the type.

The message you get, let's say in X code looks like this, no match for operator, less than, less than in standard CL, vector element, operator [inaudible]. So I'm trying to give you a little clue, but in its very cryptic C++ way of what you've done wrong.

So it comes back to what the template has to be a little bit clear about from a client point of view is to say well what is it that needs to be true. Will it really work for all types or is

there something special about the kind of things that actually need to be true about what's filled in there with a placeholder to make the rest of the code work correctly.

So if it uses string insertion or it compares to elements using equals equals, right, something like a strut doesn't by default have those behaviors and so you could have a template that would work for primitive types or types that have these operations declared, but wouldn't work when you gave it a more complex type that didn't have that data support in there.

So I have that piece code over here. I can just show it to you. I just took it down here. This is print vector and it's a little bit of template. And if I were to have a vector declared of ints, I say print vector, V that this compiles, there's nothing to print. It turns out it'll just print empty line because there's nothing in there, but if I change this to be chord T and my build is failing, right, and the error message I'm getting here is no match for trying to output that thing and it tells me that all the things I can output on a string.

Well, it could be that you could output a [inaudible] but it's not one of those things, right, chord T doesn't have a built-in behavior for outputting to a stream. So this is going to come back to be kind of important because I'm gonna keep going here is that when we get to the sort, right, when we try to build this thing, we're gonna definitely be depending on something being true about the elements, right, that's gonna constrain what the type could be.

So this is selection sort in its ordinary form, nothing special about it other than the fact that I changed it as so sorting an int, it's the sort of vector with placeholder type. So it's [inaudible] with the template typed in header, vector of type and then in here, operations like this really are accessing a type and another type variable out of the vector that was passed and then comparing them, which is smaller to decide which index.

It's using a swap and so the generic swap would also be necessary for this so that we need to have a swap for any kind of things we want to exchange. We have a template up there for swap, then the sort can build on top of that and use that as part of its workings which is then able to come under swap to any two things can be swapped using the same strategy. And this is actually where templates really start to shine.

Like swap in itself isn't that stunning of an example, because you think whatever, who cares if three lines a code, but every little bit does count, but once we get to things like sorting where we could build a really killer totally tuned, really high performance, you know, protection against a [inaudible] quick sort and we want to be sure that we can use it in all the places we need to sort.

I don't to make it sort just strings and later when I need to sort for integers, I need to copy and paste it and change string to everywhere, and then if I find a bug in one, I forget to change it in the other I have the bug still lurking there.

And so template functions are generally used for all kinds of algorithmically interesting things, sorting and searching and removing duplicates and permuting and finding the mode and shuffling and all these things that like okay, no matter what the type of thing you're working on, the algorithm for how you do it looks the same whether it's ints or strings or whatever.

So we got this guy and as is, right, we could push vectors of ints in there, vectors of strings in there and the right thing would work for those without any changes to it, but it does have a lurking constraint on what much be true about the kind of elements that we're trying to sort here.

Not every type will work. If I go back to my chord example, this XY, if have a vector of chord and I pass it to sort and if I go back and look at the code, you think about instantiating this with vector, is all chord in here, all the way through here, this part's fine; this part's fine; this part's fine, but suddenly you get to this line and that's gonna be taking two coordinate structures and saying is this coordinate structure less than another.

And that code's not gonna compile. Yes.

Student:Back when we were doing sets, we were able to explicitly say how this works.

Instructor (Julie Zelenski):Yeah, so you're right where I'm gonna be, but I'm probably not gonna finish today, but I'll have to finish up on Wednesday is we're gonna do exactly what Seth did, and what did Seth do?

Student:Comparison function.

Instructor (Julie Zelenski):That's right, he used a comparison function, so you have to have some coordination here when you say, well instead of trying to use the operate or less than on these things, how about the client tell me as part of calling sort, why don't you give me the information in the form of a function that says call me back and tell me are these things less than and so on and how to order them.

So that's exactly what we need to do, but I can't really show you the code right now, but I'll just kind of like flash it up there and we will talk about it on Wednesday, what changes make that happen.

We'll pick up with Chapter 8, actually after the midterm in terms of reading.

[End of Audio]

Duration: 50 minutes