Programming Methodology-Lecture23

Instructor (Mehran Sahami): So welcome to the 106A lecture just before Thanksgiving. I thought attendance might not be spectacular today. People are still filtering in, but it's good to see so many of you here, though. I'm sure some of you have already headed home.

You may be wondering who I am. I would have thought before we got back mid-quarter evaluations that you stood a chance of recognizing me as the TA of your class, but the comment of more than half of the people who responded to the question of how is Ben doing was, "I don't know Ben. I've never interacted with Ben. So I assume he's doing a great job."

Most of you jumped to that conclusion. But – so I sort of chased at this, but not much. I realize, though, that I would enjoy the opportunity to develop a little more exposure. And so Maron and I decided to switch roles for today. In fact, he's making copies as we speak, and there's going to be one handout that's sort of pertinent to what you'll be doing for [inaudible] the next assignment. And he should be back any time, but I certainly can't blame him for being – running a little late since I was supposed to make the copies before class myself. So anyway, in my capacity as Maron, I don't hope to be as spontaneous or as funny as he is, but I do hope to talk about something that is a little bit off of the sort of charted path of CS106A's curriculum.

So one way of describing the things that you're learning in 106A is that you're learning how to take the ways that you already know of doing things that may take you some time and maybe error-prone in the doing of them and tell a computer precisely how to do them so that you reap the benefits of a machine being able to execute those instructions. And they'll be faster. There'll be fewer errors as long as you get the instructions right in the first place. And that's got to be an exhilarating feeling. It's got to be empowering, or at least it was when I was sitting in your place as a freshman, to know that, all of a sudden, if you have a good idea – if you can come up with a procedure for solving a problem that you've always had and tell a computer how to do it, then the computer will be able to do that over and over and over again. All right? So that's 106A in a nutshell. So it's sort of – it's not a perfect metaphor. What does it mean to run breakout by hands? I don't know.

So what I – the distinction I'm trying to make between what you've been learning so far and what you will learn, hopefully, in this lecture – and certainly if you go on to CS106B – is that there are things that computers can do that you couldn't possibly have simulated by hand. Computers can handle so much more data than you could ever manage to sort through on your own manually that it's worth you're learning a bit about how to make them do that – how to think about instructing a computer to do something that you couldn't possibly have done. So today, we're going to talk about two of the most important kinds of those problems in computer science, and I'm going to stop just talking at you and show you some examples in a second. But those two topics are searching and sorting. How do you find something in a set of things that you have? And how, if you could impose some structure on that set, would you both find it more quickly, and – I

guess I got ahead of myself – how would you go about imposing the sort of structure that would help you find it more quickly?

All right. So if I'm seeming to ramble, that stops now.

So searching and sorting – I say they are important, and I hope you believe me, and you think that for that reason, it's worth talking about for an entire lecture. But they really are just a way of getting to talk about something a little deeper, which is this concept of algorithmic efficiency that we haven't said much about in the class so far. So that's the third part of this two-part lecture. What's the deal with searching? Well, searching doesn't quite fit the mold as something you couldn't do by hand. You find things all the time. Chapter 12 of the book looks at two operations of the searching and sorting. This is pretty generic. All right. So searching is simpler. You can define a search problem – say you have an array or some other collection of things, and you have something you want to find. The typical way this is done is that you want to find the index into the array – where that element was – or, in a more generic case, you want to find out if that element is even in the array. So you want to have some way of determining that it wasn't actually found.

And that may be all you care about. It may be the case that, if the search routine returns anything other than negative one, you don't actually care what its value was. But we adopt – I will adopt the convention for this lecture that if you don't find what you're searching for, then the method that you wrote to do the search should return a negative value since that's not a valid index into the array. All right. So if you have a set of things that you're trying to search through, the easiest way of doing that is just to look at all of them and see if each one, in turn, is what you're looking for. That's called a linear search because, to sort of pre-figure what we're going to talk about at the end of the lecture, the number of operations you have to do to find the element that you're looking for is linearly proportional to the number of elements that you had to begin with. Now it may not be obvious right now that there's something better that you can do, and with an arbitrary set of data, there is not anything better that you can do. But this procedure that I've written up here is an implementation of what you already could have written – a procedure for finding an element in an array of integers.

So there's nothing very tricky about this. If it had been a string, you might have even avoided having to write the function and called it something like, "Index of" with a character. And inside of "Index of", though you don't have to write the code, it would have had a four loop that iterated over each of the characters of the string, tested each one for quality with the key that you were looking for and then, presuming it found one that equaled that character – or in this case, that integer – it would return the index of it. And if the four loop completes, then you must not have examined any elements that matched, and so you return negative one. Okay? So here's a simulation of how that would work, although the – leaving nothing to the imagination here, this is pretty easy to sort of envision.

So we have this array of primes, and it has 17 in it. So we are going to expect to find that this linear search shouldn't return negative one. But the second one, we're looking for 27 – which sort of looks prime, but of course, it's not because it's nine times three. All right? Okay. So we called linear search, and here's our new stat frame, which has the local variables I and the parameters P and array in it. This is sort of Eric Robert's – who wrote the textbooks – way of displaying the execution model of JAVA. All right? So we're just looping over it, and we're testing as I equals zero and then one and then two and then three and four, five, six – where should we find it? Well, we were looking for seven – no. Right now, we're looking for 27. So we're just gonna go to the – I think there was a problem with the way that – well, anyway. The console here prints out that, indeed, we found 17 at position six. There may have been a problem converting PowerPoint slides to Keynote slides, so I apologize for the sort of shakiness of these animations. But the content of the slides should be adequate. Cool.

All right. So now we're in our second called linear search, and the only thing that's changed at this point from the beginning of the first called was that the key is different. So I is still gonna start at zero, and it's still gonna iterate over all of the positions of the array. But we're gonna go through the entire array, which means going all the way up to ten where I is no longer less than ten. And we didn't find 27 anywhere. So I hope this is all fairly clear. But these are the results that you would have expected. So we found the position of 17, and then we looked for 27 but didn't find it and returned negative one. Cool.

How many people think they could have written that in their sleep? That that was too much time to spend on such a simple idea? All right. Well, it gets more interesting from here on out, but – so talk to me. What is the problem with linear search?

Student: It takes a lot of time.

Instructor (Mehran Sahami): It takes a lot of time, right? It takes as much time as you have elements. So if I asked a question, which was – if I knew the SUID number of one of you in the room, and I asked, "Is this your SUID number?

And you say, "No."

And I ask you, "Is 05179164 your SUID number?"

Is it yours? Be kind. Tell me it's yours.

Student: [Inaudible].

Instructor (Mehran Sahami): It's not. So it actually turns out that it's mine, so I would have had to search the entire room before I finally got back to myself if I didn't make that shortcut. Okay.

So another illustration of what is really a very simple idea – that if you're forced to look at all of the elements that you are searching among, then that's bad in and of itself if you could do better. So we'll see if we can do better, but here is a rather larger data set – 286 area codes – the three-digit numbers that come before the rest of the seven digits in your phone numbers.

But we're trying to find 650, which is this county's area code. So here's a larger array. What would happen if we were searching in this? Well, it depends. If we have a fast computer, it may not matter. These are – there's only about 300 numbers here, and doing something 300 times is something that computers are reasonably good at.

But I thought I'd put together this sort of compelling example of why this can become a bad idea.

All right. So I wrote a program in JAVA, which has this implementation of linear search that we were just looking at. And the difference that you may notice is that I have an instance variable, which is sort of behaving as though it were an array, and you'll see why that is in a second. But all that I need it for is to get values from it at certain positions, and I wanna know what its length is so that I can iterate over it.

And I'm also – so why is it called DIS? It's short for display, and it's this – an instance of this number display class that I've written. And if there's time and interest, I'll talk about how that was implemented. But it's intended to sort of mimic the behavior of an array and, at the same time, show you a linear search in action.

All right. So this is a dialogue program that's just reading line to ask me for some input. And here are all of those numbers.

So I slowed it down – even if this looks like it's going relatively fast – so that you could see it sequentially considering each of these numbers. All right?

You may notice that you can – you could probably beat this. Right? It's going slowly enough that, since these numbers are already in sorted order, you can jump ahead.

So what is it – I want you to think about – while you're waiting on this to finish, I want you to think about what it is that you're doing when you search for the number in this list of numbers that happens to be sorted, which allows you to go so much more quickly than the computer. All right.

So why didn't we find 650 here?

Student: [Inaudible].

Instructor (Mehran Sahami): It wasn't even there? Yeah. Tricky. Let's fix that.

Where do we wanna stick it? Probably between 630 – they skipped whole swats of – I just pulled this off of the Internet, which is notoriously unreliable. All right. They probably aren't gonna let this run all the way through the next time. But 650, as you will notice, is now right here. So it would eventually be found. I'll let that tell me when it's finished.

So now I'm going to switch over to a different kind of search, which I hope has something to do with the way a human looks at a sorted array and finds the element that they're looking for quickly. But – no. Help me out before I just show you that implementation. What is the insight that lets you find elements more quickly if there's some structure – if there's a sorted order to the elements that you're looking for?

Go ahead.

Student: [Inaudible].

Instructor (Mehran Sahami): Okay. So if I start with the first number in the array, and I ask myself, "Is the number I'm looking for higher or lower than the first number in the array?"

What's the answer probably going to be?

Student: [Inaudible].

Instructor (Mehran Sahami): It's probably higher if it's a sorted array, right?

So I know that's not quite what you were getting at, but I'm pushing on the definition of this procedure so that you – would you like to add a clarification to what you were intending?

Student: [Inaudible].

Instructor (Mehran Sahami): Yeah. The number in the middle. Okay?

So if you look at the number in the middle, and you see that that number is less than the number you're looking for, there's a whole set of numbers that you now don't have to worry about. Right? It's all of those numbers up to and including the number in the middle of the array. Right?

So now you're looking at half of the array, and in some sense, your problem is simpler. In fact, it's a lot simpler because if you apply the same procedure again, then you can cut that remaining half in half, and you can cut the remaining half in half. Right?

So this kind of searching is called binary searching because you make a binary choice at each decision point, and you can then cut the space that you're searching in half. So what would that look like if we wrote it out in code?

Well, if you trust me, then it would look something like this. And the insight behind these admittedly too-short variable names is that we're keeping track of the LH or left-hand side of the portion of the array that we're considering and also the right-hand side. So if you look at these two initializations, it should make sense that you start with the left-hand side as far left as it can be at zero and the right-hand side at the index of the very last element in the array, which is one less than the length of the array. Right?

And so you may already be able to anticipate that we're going to sort of move these in closer and closer to each other until they identify an array that has only one element or, potentially, zero elements, in which case we would decide that we were not able to find the element that we were looking for. All right?

So we're going to do this halving procedure until it is the case that LH and RH have collided with each other in some sense. And the idea of getting the middle element is captured by this next line here where we take the average of left hand and right hand.

Now searching is such an important operation, and it's so often needed for very large data problems that talking to people in industry who have to do a lot of searching – you may hear someone say that there's a bug on this line. And it's the sort of bug that has nothing to do with computer science in the abstract, but it's just one of those details that is unfortunate – the bug being that you don't really have to add LH to RH before dividing them. You can divide each one by two and then add their halves to get the average.

This is just sort of a side comment, but why might it be problematic for you to add them together?

All right. If this rings no bells – why would you say?

Student: [Inaudible].

Instructor (Mehran Sahami): Integer division? That's – I think if you work with some small representative cases, you'll see that that doesn't prove to be a problem, although that's a good insight. But there is a maximum value – yeah?

Student: What if they're too large?

Instructor (Mehran Sahami): What if they're too large? So an integer can be as large as – an [inaudible] integer can be as large as, like, four billion. But what if each one of LH and RH is, at some point, three billion or greater? Right? And there's going to be a problem adding them together.

But we've already spent too long talking about that. I would just throw it out as sort of an interesting detail. But after many, many years of using a binary search that looks exactly like this, only recently have people begun to realize that little problems like that can sort of spell disaster with really, really big problems – really, really big sets of data.

Okay. So anyway. That is somewhat an immaterial point.

We now have the index of the middle of the array – or at least the middle of the portion of the array that we're considering right now – and we ask our sort of array-like object what number is at that index. And if it's equal to our key, then we're done. We can just return that we found the index that we – of something that matched the key. So in the linear search case, if there were multiple copies of the key that we were looking for, which one would be returned?

The first one, right?

Do you have any insight yet about which one might be returned in this case?

It's not at all clear to me. And in fact, binary searches generally don't guarantee anything about which element they return if there are multiple copies.

Anyway. Assuming that we don't get lucky, though, and that we don't find the element that we were looking for on our first try, then we have to modify the size – the part of the array that we're looking at. So how do we do that?

Well, we need to determine, as was suggested, whether the element that we're looking for is less than the middle or greater than or equal to the middle. And in the case that it's less than the element of the middle of the array, then we want to adjust which side.

So we know that it must be in the first portion of the array, and we're going to ignore the second portion of the array. So the right-hand marker that we're – is what we need to update this time. So that's what this does. It updates the right-hand marker to mid minus one. And why is it minus one?

These sorts of issues can be really frustrating to be off by one errors in otherwise already complicated algorithms.

Student: [Inaudible].

Instructor (Mehran Sahami): Okay. Arrays start at zero. Well, could it be the case after we examine this test that the element is actually at the mid position? Okay, why is that?

Student: [Inaudible].

Instructor (Mehran Sahami): Yeah. We already – oops. We already checked, and the way that we checked is because it's just a less than sign. It's strictly less than, so that implies that the two are not equal. So there may be different ways of interpreting why there's a minus one here, but my interpretation is that we never want to consider that index again. So we want to move one past the middle index that we now know is not equal to the element that we were looking for. Okay?

So in the other case, we want to adjust the other end of this portion of the array that we're considering, and so we update the left-hand side to mid plus one. All right? Why plus one?

Student: [Inaudible].

Instructor (Mehran Sahami): It's a similar reason. Right. We don't want to consider the midpoint ever again. Right?

So at the very least, we've gotten rid of the midpoint. But more importantly, we've gotten rid of everything on one or the other sides of it. Okay?

So you can now see that we are – we've got an instance of the original problem, and when we go back to the top of the Y loop, we calculate a new midpoint, and we do the comparison again. But now we're in a much smaller array, and eventually, the only directions in which RH and LH can move are directions toward each other. And no matter what happens, unless we go ahead and return mid, we're changing one of them each time. So you should be able to see that the array that we're considering is definitely getting smaller. Okay?

So that's important. This is a Y loop with an interesting test. So it's always worth considering whether the Y loop is really going to exit. But in this case, I hope that I've persuaded you that, in fact, it should exit.

Okay? So what will the search look like if we use binary search now. I'll just change this name. All right. That was quick. We could have fiddled with the delay that I added. But holding the delays the same, it should be clear that the difference in running times between these two algorithms is pretty substantial. All right?

So that's searching. Any questions about what I've explained? Yeah.

Student: [Inaudible].

Instructor (Mehran Sahami): That's an excellent question. So it depends on – I could put the question back to you. I haven't explained a good way of thinking about exactly how fast these two operations are. And I haven't even shown you how one would go about sorting the array, much less how fast that would be. But if you were going to do a bunch of look-ups, then the savings that you would reap by being able to search in a sorted array would potentially, for some number of look-ups, dwarf whatever cost it took to sort the array the very first time. Right?

But if you're in a situation where you've got all sorts of arrays, and for this array you want to find this element, and next to it's going to be some completely different array – some completely different element that you're trying to find, then your intuition – or the intuition that I imagine is behind that question that it's more costly to sort the array and then search it is perfectly valid. It would be better in those situations just to do a linear

search. And sometimes, that's all you can do. There are situations where there's no easy way to put the elements into sorted order. It's not clear what that would mean, so in situations like that, sometimes linear search, which is sort of our null hypothesis – our default algorithm – is the only thing that's available. All right. So we've already gone through the idea of binary search. Here is a prose version of the algorithm that we just walked through. And here is – I inherited these slides from someone else's, as you may be able to tell. But here is – you can see that part of the array is no longer being considered. Successively, that portion gets smaller and smaller and smaller until finally, there's only one element. Okay? Okay. So we've already talked about how linear search depends on its running time on the number of elements. Do you have a question?

Student: Yeah. I was wondering what type of search mechanisms does [inaudible] use [inaudible]?

Instructor (Mehran Sahami): Yeah. Yeah, so – we'll talk about exactly how efficient binary search is, but the basic insight is that it – the number of steps that you take is the number of times you would have to divide the original number of elements by two in order to get it down to just one element. Right?

So that's a logarithmic proportion. Right? That's just sort of jargon, and actually, I didn't think much of logarithms until I started being a computer scientist, and now they make a lot of sense. They pop up everywhere. But I'll talk about that in a second.

But yes. Okay, so back to your question about the hashmap. The thing about a hashmap is that, for all you can tell, it takes the same amount of time to find the element or to insert the element that you're trying to find or insert, no matter how many elements you put into the hashmap so far. So how is that even possible?

So the book talks about this in some detail, but the basic idea is that the hashmap has a bunch of buckets behind the scenes, and it can figure out in constant time – that is in an amount of time that doesn't depend on the number of buckets or the number of elements – which bucket the element that you're inserting or looking for should go or should be found. And then hoping that the number of things that end up in that bucket – presuming you have enough buckets – is relatively small, you can just do a linear search on the list of things for each bucket.

So the idea is that very quickly, you pair it down to a single small list to search in, and then you just search in that list. So for very short lists – or I guess I should say for very small numbers of elements – I have said actually has a lot of overhead that may outweigh the cost of just doing a linear search on the elements. But for – again, the way to think about these algorithms in relationship to each other is to consider really big examples and think about how the running time would sort of grow as a function of the input.

All right. So let's say a little bit about binary search and it's efficiency. I've already suggested that we divide the number of elements that we're considering by two each time. So if you're mathematically inclined, you may notice that we want to solve this

equation at the bottom where we find N such that N divided by two to the K times – no, N we already know. N is the size of our data set. We want to find K, which is the number of times we divide N by two. Okay?

So we can rearrange that by multiplying each side, and then the only way of solving that is to find K by taking the base two logarithm of N. That's – you could find it by inspection, but you could also type this into your calculator. There's an equivalence between these two things that, if you've forgotten, may be worth remembering now, but that is certainly not something we'll ever test you about.

I would also say that the content of this lecture is not something that you are going to be responsible for in your assignments or the exams. So if you find it interesting, then let that be it's own reward. And if not, I'm terribly, terribly sorry. Okay.

So what – how do we compare these two numbers? So you look at an expression like log base two of N, and that may not mean anything to you. It certainly didn't mean much to me. But intuitively, if you remember that what we're talking about is dividing this array in half repeatedly, then it should seem like that's a lot faster than searching every element. And indeed, it is considerably, considerably faster. If it had been log base ten, the number would be approximately the number of digits in – NN, if you wrote it out as a decimal number. In log base two, it's the number of binary digits if you were to write it out as a binary number.

So you know that it's easy to write very large numbers with a string of digits. And in this case, if we scale in all the way up to a billion here, then a billion has approximately 30 binary digits – or another way of saying this is that the logarithm of N base two when N is a billion is 30. So notice how log base two of N is growing relative to N. The difference is just enormous. So by imposing a little bit of structure on the data that we wanted to search through, we made it much, much easier to find elements. Okay. All right. So that's searching, and I hope the points there are well taken because sorting is now sort of like the dual concept of the searching that we've just been doing. If it's easier to search when we have imposed some order on the elements that we're trying to search among, then how do we go about imposing that order? So 106B spends quite a bit of time talking about different sorting algorithms. There's a great assignment where they give you a program that implements these sorting algorithms, but not the source code. And you just have to run it on a bunch of different kinds of input. And given what you know about how those sorting algorithms behave, figure out which one of them is which. So it's sort of like a practical lab from a natural science. It's the only time I've done anything like that in the computer science world, but if that sounds exciting, think about taking 106B. All right. So here's the sorting algorithm that corresponds, perhaps the best, to the way that the humans sort things. So if you had a deck of cards, and you wanted to put those cards in – back in order after sort of – after shuffling them, you would look at the first card, and you'd think to yourself, "Gosh. This is great. I've already got another deck of cards that is sorted. It happens to have only one card in it." So it's sort of sorted vacuously. But starting with this sorted deck, can I add more cards to it and keep them in the same sorted order? Right? So you would take the next card off of the larger deck, and

you would either put it in front of or behind the card that you had. And now you'll have a deck of two cards that are sorted with respect to each other. And you would continue on in this way until you had exhausted the original deck, and you had a fully sorted secondary deck. Would anybody go about this differently? Is there a better way of sorting? Okay. Well, that's certainly what I would have done, and that's what is implemented by this as long as you trust the defined smallest function. So we just step through the array – and this is a bit different. We're not – we don't have two separate arrays that we're building up. But for each element, we look for the smallest element that's after that. And on the assumption that there shouldn't be any smaller elements after that, if we find such a smaller element, then we swap it with the element that we're currently looking at. Right? So by the time we get to the end of the array, and we swapped out each position with anything that was smaller than the element at that position, then our array should be sorted. Right? As a bit of review, why do we call swap elements? You'll have to imagine how swap elements is implemented. But based on what you can assume, why do we have to call swap elements with three arguments? Why couldn't we just pass an array sub LH and array sub RH?

Student: [Inaudible].

Instructor (Mehran Sahami): Yeah. It only passes N. So those are just the values from the array, and our goal is to modify the array in place. Right? So the swap elements function needs to know about that array, too.

Okay? So I don't know if I can trust the simulation, but I actually coded up a version of select and sort myself and an eclipse, so let us see a somewhat more interesting simulation of how that would work.

Okay. So I'm gonna use a much smaller file because as I have – as I've already suggested, sorting takes more time than even linear search. All right.

So just to give you some bearings as to what's going on in this program, I have this number display, which is just a G canvas that has these operations that let us highlight numbers and swap them around. And I'm creating one of those down here as an instance variable and adding it and run and then initializing it with the contents of the file that's typed in. Right?

So if I were to just run the program as it is right now – I use a smaller text file. It should slow this down, but it's going to take enough time anyway that you can see sort of what's happening.

Each number is turning red when we examine it, so in some sense, you can think of the running time of the program as depending on the number of times, given the input that you have to examine elements of the array. So the way I structured this was just that examining any element of this pseudo-array would have the side effect of making it a little bigger and turning it red – and also pausing for about a quarter of a second.

Okay. So actually, I think it would be worthwhile to slow that down and look at what's happening. The value of constants – that was easy. Okay.

So we're considering four right now. One of the things that this doesn't keep track of very well is what number we're considering. Right? But after four, the smallest number that we found was two, and so we swapped them out. Two happened to occur immediately after four. And so we considered four. Again, this is just sort of a special case where four happened to be the – happened to be swapped into the position that we next considered.

So now is anything less than eight? Well, seven is. Oh, four is even less than seven. Now we swap the eight with the four. It's going to turn out that nothing is less than – or everything is less than nine – five in particular. Okay.

So how long did that take? Was anybody timing that? What would it mean? I mean, if we could just change the amount of delay arbitrarily, right, we could have made this take any amount of time. And in some sense, when computers get faster and faster because the hardware people are doing their jobs, that's what's happening. The delay that someone artificially – or in that case, not artificially – put into the system is just being reduced. Right? So everything seems to run faster. The rising tide raises all boats. So what would – I mean, would it even be useful to have timed that – to have a number of seconds representing how long it took to sort these ten elements? It's a lot harder to count when you're standing in front of a bunch of people. But it – yeah, I think that's ten. Okay. Right?

So clearly, it's sort of a result of a number of factors exactly how much time these operations take. So I – with what remains of the lecture, I'd like to give you sort of a better way of talking about how fast a program is. So if you buy my contention that you can't just time your program without making sure that all of the other factors stay the same, then what ideally we would want to be able to say is that, given some kind of input, about how long is it going to take? And in particular, how long would it take if, for instance, I doubled the size of the input – if there were twice as many numbers here.

For linear search, if we were searching for the five in this array – or let's just take the pathological case. If we were searching for the nine, we would have to examine all of the elements before we found it. If we double the length of the array and made sure that nine didn't appear anywhere except the end, again we would have doubled the time that it took to find the nine. But would that be true in the binary search case?

Well, no. We had only – we doubled N. And remember that the running time is, in some sense, proportional to the logarithm of N base two. So we wouldn't have doubled the running time. We would have only increased it by a factor of logarithm of N times two minus – or divided by the logarithm of N. I didn't say the base there because it actually doesn't matter. If you divide two logarithms by each other, you get the same answer no matter what base you took them at originally. It's sort of a nice property of logarithms.

Okay. So I think my simulation was better. Feel free to disagree. But I'm trying to skip – oh. This actually has a little finger pointing to the elements. Maybe there's something to be said for this. Anyway.

I'll post that program online incase anyone is curious. This is taking a bit longer to blow through than I was hoping. All right.

So how efficient is selection sort? Well, it's not okay to just give a number seconds. We want to give some kind of proportion, as we did for the searches. Right?

And if you noticed, for every element of the array, we had to consider every remaining element of the array. If we wanted to be a little less efficient in some sense, we could have just – well, I was going to say we could have considered all of the elements and just ignored the ones that did not actually come afterwards. But we can calculate sort of in a rough sense what that proportion should be in this case. But before I do that, here's a slide where I attempted to do the timing test. So I held all the factors constant that I could, except that I changed the size of the input. So I went from having just ten elements, which took me a very, very short amount of time, to having 1,000 elements, which took me a matter of minutes – 10,000 elements. Excuse me. Right? So this sort of illustrates the point, and it shows in a heuristic sense how the running time grows with the size of the input. But what's going on here? The – in the far column, we have the standard deviation, which is a measure of how different the timings were. And if you look on a particular row for any of these trials, all of those numbers are really supposed to be the same because I'm sorting the same number of elements. Right? But why are they different? Why such variation?

So I'm sorting 100 elements, and sometimes it takes .146 seconds, and sometimes it takes as much as .176. Just chalk it up to the randomness of the universe? I guess, but it's even easier to explain that problem away because your computer is doing so many more things than just running your eclipse program. It's written in JAVA. It may have to pay attention to activity on the network. You may be running iTunes at the same time, and it has to load a file in from the disc, which takes away time.

So these time measurements are just in terms of absolute time, and for that reason, they don't give a very accurate indication of the job that your program was doing all by itself. All right.

So [inaudible] is sort of crazy values. For 1,000 for the most part, it took just 13 seconds almost as much as 20 and almost as little as ten. The average was 21, though, because one of those time trials took 81 seconds. Right? Just because something else on the computer was happening that was taking time away. Right?

So clearly, we can begin to supply a better intuition for the question that was asked earlier about whether it would be okay to sort an array and then search it if all we wanted was to do a single search. All right.

So on the very first time through, we're looking at all of the elements in the array after the first one. How many are we looking at on the second time through?

Student: [Inaudible].

Instructor (Mehran Sahami): One less than that? Okay. And on the third time through, one less than that. So the total number that we're looking for is – or the total number of times we have to examine some element of the array is N plus N minus one plus N minus two all the way down to one. Right?

Does anyone know how – what this sums up to? There's actually a closed form for it. Yeah?

Student: [Inaudible].

Instructor (Mehran Sahami): Okay. Yeah. So if you work it out with small examples, you'll find that this is true, and you can probably then convince yourself of it. This sums to N times N plus one over two. Okay?

So how do we make sense of that? How do you look at that and tell intuitively what kind of algorithm this is – how efficient this selection sort is?

Well, in computer science – this is one of the things I love about computer science – when we're talking about algorithmic efficiency because we really don't care about any of the sort of details. We only care about the aggregate behavior of these programs. We can drop pieces of information like this, too. Right?

Who cares if the algorithm is twice as fast or half as fast as another algorithm? If it's – you just need a computer that's twice as fast or half as fast. But if, say, an algorithm takes the square of the amount of time that another algorithm took, then really, getting a faster computer is not the answer for sufficiently large inputs.

So – whereas we can drop constant factors like two and just pretend that this is – well, you can see that this simplifies to N squared plus N. Right? Not only can we drop constant factors, but we can drop anything that doesn't really contribute in a big way to the running time. So it doesn't even matter that we have this plus in here. So we can look at this and sort of think of it intuitively as a quadratic that is squared function of the size of the input.

So what does this mean? It means that if we double the size of our input, then the selection sort that we just walked through is going to take the square of the amount of time that it took originally. All right?

So this is a computation of that. You can also think about it geometrically if you treat each of the accesses of array elements as these circles. All right.

So this isn't quite as bad as the difference between – no. Actually, I should say that we – to demonstrate the difference between linear search and binary search, we had to make N really large, while log base two of N was still relatively small. Here, it's sort of the other way around. I guess explaining slides that I didn't write is a bit of a waste of time for you guys. So do you have questions about how we did that calculation? Why I modified that to make it simpler?

If you do afterwards, then feel free to ask.

So I sort of wanted to leave you with some insight about how we could possibly do a little bit better than selection sort. And it turns out there are algorithms that do substantially better than selection sort. And one of the easiest of them to describe is an algorithm called radix – or radix sort. And it comes from back in the days when we had these punch cards with holes in them to represent numbers. And – so say you had a bunch of three digit numbers that you wanted to sort. Well, the basic idea with radix sort is that you would set the – you would have this machine that could sort of read the holes and sort the elements into buckets such that every – the elements in this case being the cards. So each one of the cards is sorted into a bucket according to its – the first digit of the number that it represents. Right?

So you've got these buckets that have cards that aren't in any order within the bucket, but then you at least know that each bucket contains all the cards that are less than all the cards in the next bucket. So you can take them all out and stick them back together as a single thing, and then set the machine – the machine has this thing called the hopper, which is just a place to put the cards – all right. So run the machine again, but the second time, you run it – have it sort them in the order of the second digit in the numbers, and assuming that we have three digit numbers, then – now the cards are all going to be sorted within their bins with respect to both the last digit and the second to last digit. And so if we do this just on more time and sort the resulting set of cards on the one remaining digit, then we end up with bins that actually represent a complete ordering. If we took out the dividers between the bins, we could stick a stack of cards together, and that would be ordered.

So this again is an animation that didn't come through so well. I think we're about out of time. So I wish that I could have given a better intuition about how much better radix sort is than selection sort, but thank you very much for paying attention to me. And have a great Thanksgiving.

[End of Audio]

Duration: 51 minutes