ProgrammingAbstractions-Lecture08

Instructor (Julie Zelenski):Good afternoon. You guys are talkative. That sounds like a good weekend. I heard like some low noise on Assignment 2, and I thought we could elevate that to a little bit broader level, so if there's somebody who's already gotten fairly far on one or both of the problems on there, and has stumbled across some insight that perhaps was a little bit painful to get through, but you'd like to provide the benefit to your fellow classmates, now would be a great time to offer any bit of advice you think that might make the world a happier place.

Anyone? No one? [Inaudible]. You're sitting in the wrong seat. You don't sit there.

Student: When you make an iterator of like a map or [inaudible] like that, capitalize the word iterator.

Instructor (Julie Zelenski):Okay. So this insight is that the iterator class that is nested within the map and nested within the set, its name really is capital I iterator, so it's capital M map of something you're putting in there, int, whatever, and then colon colon capital I iterator.

It's pretty easy to look at the code and be a little bit – the lower case I and capital I often in small fonts look about the same, too.

Student:[Inaudible].

Instructor (Julie Zelenski):I've no doubt, and that actually is a great bit of insight is that you think the error would be like "Spell it the right way, doofus," and C++ never gives you the error and says here's what you'd like to do to fix it. It says according to my internal specs of which I have referenced point Article 72 of the C++ standard and all this stuff that is just total gobbledygook.

So one of the things you get very good at is letting the compiler direct you to where to look for the problem, but then ignoring what it told you was the problem because its analysis of it was often not very helpful. You just get good at kind of looking at the line yourself to figure it out. Okay?

Student:I keep forgetting to clear the classes or renew like objects and things like that.

Instructor (Julie Zelenski):Yeah. So all those pound includes, right? So C++, very fussy about that. You start using set. You start using map. You need to make sure you've got that pound include of the set and the map and whatnot, and that none of the code you write that will use it will make any sense to the compiler until it's been fully informed about what map and set look like. So if you don't do that – And same thing for like iostream and all these other things you're using the random library, when you start making those calls, C++ wants to know what the interfaces are and how they're set up.

And the way to tell it about it is to get the right pound includes in there. Without them, it won't get you very far at all.

Student:I have a question. Why isn't it possible to just give access to every [inaudible]?

Instructor (Julie Zelenski):Well, you could certainly do that. We could make one sort of big master include, which is like include the whole world, and just grab everything and bring it on in. And what that will do is it will just slow down all your compiles because it will be looking through that thing again and again every single time. And even though you're not using it, it will have to have kind of read past it and understood it. And then it turned out to be a wasted effort. In the case of our programs, the amount of headers we're talking about is small enough that you could imagine doing that without really slowing your development down. But for the large-scale development, you will typically include exactly those headers you need to avoid slowing down all your compiles by rereading a lot of things that aren't necessary for this unit.

Student: Careful when you concatenate characters to strings.

Instructor (Julie Zelenski):Yes.

Student: Because it will give you – because if you do it a certain way, it will give you a bunch of junk.

Instructor (Julie Zelenski):That is a great point. And we talked about that at strings, but it's a fine time to reiterate it – is that when you're using [cuts out] plus equal to take a string and extend it with some new characters or another string, one of those operands needs to be a C++ string, which means a variable, a parameter, something that was declared and created as a string.

And the things in double quotes remember are old style strings, and until they have been converted they are still old style. So if you have an expression that looks like this, and you try to add a character there, this compiles but it does not do what you want. It takes the old style string and uses this character kind of as an offset and causes the string basically to turn into garbage. So if you were to say something equals this, you will not get what you want. You're expecting to get my string extended with that new character. So remember that one or both the arguments needs to be a string. If you ever need to force, you can introduce that typecast around it, and does the promotion. In most situations, the promotion will happen automatically and you won't have to be involved, but this is one case where the legacy of C++ being derived from C means that the old language did have a meaning for this, and they couldn't break that old meaning, so they left it in place for you to stumble across when you least expect. So that's a great thing to always be keeping in mind when you're doing that concatenation, looking at your arguments, making sure one of them at the very least [cuts out] C++ screen. Way over there.

Student: Why does the set pair operator need to have ordering?

Instructor (Julie Zelenski):Because it actually is more than just a quality. So he's asking [cuts out] just say yes, no, are they the same? It's really using sorting to [cuts out] are you exactly like this one, but should I put you to the things that are smaller or the things that are larger because it's largely using that to kind of quickly throw away the parts of the set that it doesn't need to explore to find a match. If you actually made yours just equalities that just returned zero and one all the time –

Student: Yeah, and it doesn't work.

Instructor (Julie Zelenski):It definitely doesn't work. It will end up just losing things. You'll put them in the set, and it won't be able to find them again. That's because you told it they would be one place, and in fact, they didn't get put there because in effect your comparison looks a little random. So it really does want full ordering. We'll keep going. So Assignment 2 is coming in this Wednesday, right? So hopefully you're making good progress on that, and then we will get out your third assignment then, which will be your recursion problem set, which allows you to practice on recursion. We're gonna be talking about recursion all week. The reader chapters that go along with this pretty much lecture for lecture 4, 5, and 6, and this is the place where I just encourage you again. I think that's some of the best material in the reader, so I encourage you to make some time to do the reading, especially in advance of lecture will pay off the best. So we talked just a little bit about the vocabulary of recursion at the end of Friday. It was very rushed for time, so I'm just gonna repeat some of those words to get us thinking about what the context for solving problems recursively looks like. And then we're gonna go along and do a lot of examples [cuts out]. So the idea is that a recursive function is one that calls itself. That's really all it means at the most trivial level. It says that in the context of writing a function binky, it's gonna make a call or one or more calls to binky itself past an argument as part of solving or doing some task. The idea is that we're using that because the problem itself has some self-similarity where the answer I was seeking – so the idea of surveying the whole campus can actually be thought of as well if I could get somebody to survey this part of campus and this part of campus, somebody to survey all the freshmen, somebody to [cuts out] and whatnot, that those in a sense are – those surveys are just smaller instances of the same kind of problem I was originally trying to solve. And if I could recursively delegate those things out, and then they themselves may in turn delegate even further to smaller but same structured problems to where we could eventually get to something so simple – in that case of asking ten people for their input that don't require any further decomposition – we will have worked our way to this base case that then we can gather back up all the results and solve the whole thing. This is gonna feel very mysterious at first, and some of the examples I give you'll say I have really easy alternatives other than recursion, so it's not gonna seem worth the pain of trying to get your head around it. But eventually, we're gonna work our way up to problems where recursion really is the right solution, and there are no other alternatives that are obvious or simple to do. So the idea throughout this week is actually just a lot of practice. Telling you what the terms mean I think is not actually gonna help you understand it. I think what you need to see is examples. So I'm gonna be doing four or five examples today, four or five examples on Wednesday, and four or five examples on Friday that each kind of build on each other, kind of take the ideas and get a little more

sophisticated. But by the end of the week, I'm hoping that you're gonna start to see these patterns, and realize that in some sense the recursive solutions tend to be more alike than different. Once you have your head around how to solve one type of problem, you may very well be able to take the exact same technique and solve several other problems that may sound different at first glance, but in the end, the recursive structure looks the same. So I'd say just hold the discomfort a little bit, and wait to see as we keep working which example may be the one that kind of sticks out for you to help you get it through. So we're gonna start with things that fit in the category of functional recursion. The functional in this case just says that you're writing functions that return some non-void thing, an integer, a string, some vector of results, or whatever that is, and that's all it means to be a functional recursion. It's a recursive function that has a result to it. And because of the recursive nature, it's gonna say that the outer problem's result, the answer to the larger problem is gonna be based on making calls, one or more calls to the function itself to get the answer to a smaller problem, and then adding them, multiplying them, comparing them to decide how to formulate the larger answer. All recursive code follows the same decomposition into two cases. Sometimes there's some subdivisions within there, but two general cases. The first thing – this base case. That's something that the recursion eventually has to stop. We keep saying take the task and break it down, make it a little smaller, but at some point we really have to stop doing that. We can't go on infinitely. There has to be some base case, the simplest possible version of the problem that you can directly solve. You don't need to make any further recursive calls. So the idea is that it kinda bottoms out there, and then allows the recursion to kind of unwind. The recursive cases, and there may be one or more of these, are the cases where it's not that simple, that the answer isn't directly solvable, but if you had the answer to a smaller, simpler version, you would be able to assemble that answer you're looking for using that information from the recursive call. So that's the kind of structure that they all look like. If I'm at the base case, then do the base case and return it. Otherwise, make some recursive calls, and use that to return a result from this iteration. So let's first look at something that – the first couple examples that I'm gonna show you actually are gonna be so easy that in some sense they're almost gonna be a little bit counterproductive because they're gonna teach you that recursion lets you do things that you already know how to do. And then I'm gonna work my way up to ones that actually get beyond that. But let's look at first the raise to power. C++ has no built-in exponentiation operator. There's nothing that raises a base to a particular exponent in the operator set. So if you want it, you need to write it, or you can use – there's a math library called pow for raise to power. We're gonna run our own version of it because it's gonna give us some practice thing about this. The first one I'm gonna show you is one that should feel very familiar and very intuitive, which is using an iterative formulation. If I'm trying to raise the base to the exponent, then that's really simply multiplying base by itself exponent times. So this one uses a for loop and does so. It starts the result at one, and for iterations through the number of exponents keeps multiplying to get there. So that one's fine, and it will perfectly work. I'm gonna show you this alternative way that starts you thinking about what it means to divide a problem up in a recursive strategy. Base to the exponent – I wanna raise five to the tenth power. If I had around some delegate, some clone of myself that I could dispatch to solve the slightly smaller problem of computing five to the ninth power, then all I would need to do is take that answer and multiply it by one more five,

and I'd get five to the tenth. Okay. If I write code that's based on that, then I end up with something here – and I'm gonna let these two things go through to show us that to compute the answer to five to the tenth power, what I really need is the answer to five to the ninth power, which I do by making a recursive call to the same function I'm in the middle of writing. So this is raise that I'm defining, and in the body of it, it makes a call to raise. That's what is the mark of a recursive function here. I pass slightly different arguments. In this case, one smaller exponent which is getting a little bit closer to that simplest possible case that we will eventually terminate at where we can say I don't need to further dispatch any delegates and any clones out there to do the work, but if the exponent I'm raising it to is zero, by definition anything raised to the exponent of zero is one, I could just stop there. So when computing five to the tenth, we're gonna see some recursion at work. Let me take this code into the compiler so that we can see a little bit about how this actually works in terms of that. So that's exactly the code I have there, but I can say something that I know the answer to. How about that? First, we'll take a look at it doing its work, so five times five times five should be 125. So we can test a couple little values while we're at it. Two the sixth power, that should be 64, just to see a couple of the cases, just to feel good about what's going on. And then raising say 23 to the zero power should be one as anything raised to the zero power should be. So a little bit of spot testing to feel good about what's going on. Now I'm gonna go back to this idea like two to the sixth. And I'm gonna set a breakpoint here. Get my breakpoint out. And I'm gonna run this guy in the debugger. Takes a little bit longer to get the debugger up and running, so I'll have to make a little padder up while we're going here. And then it tells me right now it's breaking on raise, and I can look around in the debugger. This is a – did not pick up my compilation? I think it did not. I must not have saved it right before I did it because it's actually got the base is 23 and the exponent is zero. It turns out I don't wanna see that case, so I'm gonna go back and try again. I wanna see it – no, I did not. And I'm just interested in knowing a little bit about the mechanics of what's gonna happen in a recursive situation. If I look at the first time that I hit my breakpoint, then I'll see that there's a little bit of the beginnings of the student main, some stuff behind it. There's a little bit of magic underneath your stack that you don't really need to know about, but starting from main it went into raise, and the arguments it has there is the base is two, the exponent is six. If I continue from here, then you'll notice the stack frame got one deeper. There's actually another indication of raise, and in fact, they're both active at the same time. The previous raise that was trying to compute two to the sixth is kind of in stasis back there waiting for the answer to come back from this version, which is looking to raise two to the fifth power. I continue again, I get two to the fourth. I keep doing this. I'm gonna see these guys kinda stack up, each one of those kind of waiting for the delegate or the clone to come back with that answer, so that then it can do its further work incorporating that result to compute the thing it needed to do. I get down here to raising two to the first power, and then finally I get to two to the zero, so now I've got these eight or so stacked frames, six up there. This one, if I step from here, it's gonna hit the base case of returning one, and then we will end up kind of working our way back out. So now, we are at the end of the two to the one case that's using the answer it got from the other one, multiplying it by two. Now I'm at the two to the two case, and so each of them's kind of unfolding in the stack is what's called unwinding. It's popping back off the stack frames that are there and kind of revisiting them, each passing up the

information it got back, and eventually telling us that the answer was 64, so I will let that go. So the idea that all of those stack frames kind of exist at the same time – they're all being maintained independently, so the idea that the compiler isn't confused by the idea that raise is invoking raise which is invoking raise, that each of the raise stack frames is distinct from the other ones, so the indications are actually kept separate. So one had two to the sixth, the next one had two to the fifth, and so on. And then eventually we need to make some progress toward that base case, so that we can then stop that recursion and unwind. Let me actually show you something while I'm here, which is one thing that's a pretty common mistake to make early on in a recursion is to somehow fail to make progress toward that base case or to – not all cases make it to the base case. For example, if I did something where I forgot to subtract one [cuts out], and I said oh yeah, I need to [cuts out]. In this case, I'm passing it exactly the same arguments I got. If I do this and I run this guy, then what's gonna happen is it's gonna go two to the sixth, two to the sixth, two to the sixth, and I'm gonna let go of this breakpoint here because I don't really wanna watch it all happening. And there it goes. Loading 6,493 stack frames, zero percent completed, so that's gonna take a while as you can imaging. And usually, once you see this error message, it's your clue to say I can cancel, I know what happened. The only reason I should've gotten 6,500 stack frames loaded up is because I made a mistake that caused the stack to totally overflow. So the behavior in C++ or C is that when you have so many of those stack frames, eventually the space that's been allocated or set aside for the function stack will be exhausted. It will use all the space it has, and run up against a boundary, and typically report it in some way that suggests – sometimes you'll see stack overflow, stack out of memory error. In this case, it's showing you the 7,000 stack frames that are behind you, and then if you were to examine them you would see they all have exactly the same argument, so they weren't getting anywhere. I'm not gonna actually let it do that because I'm too impatient. Let me fix this code while I'm here. But other things like even this code that actually looks correct for some situations also has a subtle bug in it. Even if I fixed this, which is that, right now it assumed that the exponent is positive, that it's some number that I can subtract my way down to zero. If I actually miscalled raise, and I gave it a negative exponent, it would go into infinite recursion as well. If you started it at ten to the negative first power, it would go negative first, negative second, negative third. 6,500 stack frames later, we'd run out of space. In this case, since we're only intending to handle those positive powers, we could just put an error that was like if the exponent is less than zero then raise an error and don't even try to do anything with it. Okay. So let me show you a slightly different way of doing this that's also recursive, but that actually gets the answer a little bit more efficiently. This is a different way of dividing it up, but still using a recursive strategy which is that if I'm trying to compute five to the tenth power, but if I have the answer not of five to ninth power, but instead I have the answer of five to the fifth power, and then I multiply that by itself, I would get that five to the tenth power that I seek. And then there's a little bit of a case though of what if the power I was trying to get was odd, if I was trying to raise it to the eleventh power, I could compute the half power which get me to the fifth – multiplied by the fifth, but then I need one more base multiplied in there to make up for that half. Okay. And so I can write another recursive formulation here. Same sort of base case about detecting when we've gotten down, but then in this case the recursive call we make is to base of the exponent divided by two, and then we use it with an if else whether the

exponent was even or odd, it decided whether to just square that number or whether to square it and tack in another power of the base while we're at it. So this one is gonna be much more quick about getting down to it, whereas the one we saw was gonna put one stack frame down for every successive exponent power. So if you wanted to raise something to the 10th, or 20th, or 30th power, then you were giving yourself 10, 20, 30 stack frames. Something like 30 stack frames is not really something to be worried about, but if you were really trying to make this work on much larger numbers, which would require some other work because exponent is a very rapidly growing operator and would overflow your integers quickly, but this way very quickly divides in half. So it goes from a power of 100, to a power of 50, to 25, to 12, to 6, to 3, to 2, to 1, so that dividing in half is a much quicker way to work our way down to that base case and get our answer back, and we're doing a lot fewer calculations than all those multiplies, one per base. So just a little diversion. So let me tell you something, just a little bit about kind of style as it applies to recursion. Recursion is really best when you can express it in the most kind of direct, clear, and simple code. It's hard enough to get your head around a recursive formulation without complicating it by having a bunch of extraneous parts where you're doing more work than necessary, or redundantly handling certain things. And one thing that's actually very easy to fall in the trap of is thinking about there's lots of other base cases you might be able to easily handle, so why not just go ahead and call out for them, test for – you're at the base case. You're close to the base case. Checking before you might make a recursive call, if you're gonna hit the base case when you make that call, then why make the call. I'll just anticipate and get the answer it would've returned anyway. It can lead to code that looks a little bit like this before you're done. If the exponent's zero, that's one. If the exponent's one, then I can just return the base. If it's two, then I can just multiply the base by itself. If it's three, I can start doing this. Certainly, you can follow this to the point of absurdity, and even for some of the simple cases, it might seem like you're saving yourself a little bit of extra time. You're like why go back around and let it make another recursive call. I could stop it right here. It's easy to know that answer. But as you do this, it complicates the code. It's a little harder to read. It's a little harder to debug. Really, the expense of making that one extra call, two extra calls is not the thing to be worried about. What we really want is the cleanest code that expresses what we do, has the simplest cases to test, and to examine, and to follow, and not muck it up with things that in effect don't change the computational power of this but just introduce the opportunity for error. Instead of a multiply here, I accidentally put a plus. It might be easy to overlook it and not realize what I've done, and then end up computing the wrong answer when it gets to the base case of two. In fact, if you do it this way, most things would stop at three, but these would suddenly become kind of their own special cases that were only hit if you directly called them with the two. The recursive cases will all stop earlier. It just complicates your testing pass now because not everything is going through the same code. So we call this arm's length recursion. I put a big X on that looking ahead, testing before you get there. Just let the code fall through. So Dan, he's not here today, but he talked to me at the end of Friday's class, and it made me wanna actually just give you a little bit of insight into recursion as it relates to efficiency. Recursion by itself, just the idea of applying a recursive technique to solving a problem, does not give you any guarantee that it's gonna be the best solution, the most efficient solution, or the fact that it's gonna give you a very inefficient solution. Sometimes people

have kind of a bad rap on recursion. It's like recursion will definitely be inefficient. That actually is not guaranteed. Recursion often requires exactly the same resources as the iterative approach. It takes the same amount of effort. Surveying the campus – if you're gonna survey the 10,000 people on campus and get everybody's information back, whether you're doing it divide and conquer, or whether you're sitting out there in White Plaza asking each person one by one, in the end 10,000 people got survey. The recursion is not part of what made that longer or shorter. It might actually depending on how you have resources work out better. Like if you could get a bunch of people in parallel surveying, you might end up completing the whole thing in less time, but it required more people, and more clipboards, and more paper while the process is ongoing than me standing there with one piece of paper and a clipboard. But then again, it took lots of my time to do it. So in many situations, it's really no better or no worse than the alternative. It makes a little bit of some tradeoffs of where the time is spent. There are situations though where recursion can actually make something that was efficient inefficient. There are situations where it can take something that was inefficient and make it efficient. So it's more of a case-by-case basis to decide whether recursion is the right tool if efficiency is one of your primary concerns. I will say that for problems with simple iterative solutions that operate relatively efficiently, iteration is probably the best way to solve it. So like the raise to power, yeah you could certainly do that iteratively. There's not some huge advantage that recursion is offering. I'm using them here because they're simple enough to get our head around. We're gonna work our way toward problems where we're gonna find things that require recursion, which is kind of one of the third points I put in. Why do we learn recursion? What's recursion gonna be good for? First off, recursion is awesome. There are some problems that you just can't solve using anything but recursion, so that the alternatives like I'll just write it iteratively won't work. You'll try, but you'll fail. They inherently have a recursive structure to them where recursion is the right tool for the job. Often, it produces the most beautiful, direct, and clear elegant code. The next assignment that will go out has these problems that you do in recursion, and they're each about ten lines long. Some of them are like five lines long. They do incredible things in five lines because of the descriptive power of describing it as here's a base case, and here's a recursive case, and everything else just follows from applying this, and reducing the problem step by step. So you will see things where the recursive code is just beautiful, clean, elegant, easy to understand, easy to follow, easy to test, solves the recursive problem. And in those cases, it really is a much better answer than trying to hack up some other iterative form that may in the end be no more efficient. It may be even less efficient. So don't let efficiency be kind of a big fear of what recursion is for you. So I'm gonna do more examples. I've got three more examples, or four I think today, so I will just keep showing you different things, and hopefully the patterns will start to kind of come out of the mist for you. A palindrome string is one that reads the same forwards and backwards when reversed. So was it a car or a cat I saw, if you read that backwards, it turns out it says the same thing. Go hang a salami, I'm a lasagna hog. Also handy when you need to have a bar bet over what the longest palindrome is to have these things around. There are certainly ways to do this iteratively. If you were given a string and you were interested to know is it a palindrome, you could kind of do this marching – you're looking outside and kind of marching your way into the middle. But we're gonna go ahead and let recursion kinda help us deal with the subproblem of this,

and imagine that at the simplest possible form – you could say that a palindrome consists of an interior palindrome, and then the same letter tacked on to the front and the back. So if you look at was it a car or a cat I saw, there are two Ws there. It starts and it ends with a W, so all palindromes must start and end with the same letter. Okay. Let's check that, and if that matches, then extract that middle and see if it's a palindrome. So it feels like I didn't really do anything. It's like all I did was match two letters, and then I said by the way delegate this problem back to myself, making a call to a function I haven't even finished writing to examine the rest of the letters. And then I need a base case. I've got a recursive case, right? Take off the outer two letters. Make sure they match. Recur on the inside. What is the simplest possible palindrome?

Student: One letter?

Instructor (Julie Zelenski):One letter. One letter makes a good palindrome. One letter is by definition first and last letter are the same letter, so it matches. That's a great base case. Is it the only base case?

Student:[Inaudible].

Instructor (Julie Zelenski):Two letters is also kind of important, but there's actually an even simpler form, right? It's the empty string. So both the empty string and the single character string are by definition the world's simplest palindromes. They meet the requirements that they read the same forward and backwards. Empty string forwards and backwards is trivially the same, so that makes it even easier than doing the two-letter case. So if I write this code to look like that where if the length of the strength is one or fewer, so that handles both the zero and the one case, then I return true. Those are trivial palindromes of the easiest immediate detection. Otherwise, I've got a return here that says if the first character is the same as the last character, and the middle – so the substring starting at Position 1 that goes for length minus two characters that picks up the interior discarding the first and last characters – if that's also a palindrome, then we've got a palindrome. So given the short circuiting nature of the and, if it looks at the outer two characters and they don't match, it immediately just stops right there and says false. If they do match, then it goes on looking at the next interior pair which will stack up a recursive call looking at its two things, and eventually we will either catch a pair that doesn't match, and then this false will immediately return its way out, or it will keep going all the way down to that base case, hit a true, and know that we do have a full palindrome there. So you could certainly write this with a for loop. Actually, writing it with a for loop is almost a little bit trickier because you have to keep track of which part of the string are you on, and what happens when you get to the middle and things like that. In some sense, the recursive form really kinda sidesteps that by really thinking about it in a more holistic way of like the outer letters plus an inner palindrome gives me the answer I'm looking for. So this idea of taking a function you're in the middle of writing and making a call to it as though it worked is something that – they require this idea of the leap of faith. You haven't even finished describing how is palindrome operates, and there you are making a call to it depending on it working in order to get your function working. It's a very kind of wacky thing to get your head around. It feels a little bit

mystical at first. That feeling you feel a little bit discombobulated about this is probably pretty normal, but we're gonna keep seeing examples, and hope that it starts to feel a little less unsettling. Anybody wanna ask a question about this so far? Yeah?

Student: So I guess create your base case first, then test it? [Inaudible].

Instructor (Julie Zelenski):That's a great question. So I would say typically that's a great strategy. Get a base case and test against the base case, so the one character and the two character strings. And then imagine one layer out, things that will make one recursive call only. So in this case for the palindrome, it's like what's a two-character string? One has AB. One has AA. So one that is a palindrome, one that isn't, and watch it go through. Then from there you have to almost take that leap and say it worked for the base case. It worked for one out. And then you have to start imagining if it worked for a string of length N, it'll work for one of N plus one, and that in some sense testing it exhaustively across all strings is of course impossible, so you have to kind of move to a sort of larger case where you can't just sit there and trace the whole thing. You'll have to in some sense take a faith thing that says if it could have computed whether the interior's a palindrome, then adding two characters on the outside and massaging that with that answer should produce the right thing. So some bigger tests that verify that the recursive case when exercised does the right thing, then the pair together – All your code is going through these same lines. The outer case going down to the recursive case, down to that base case, and that's one of the beauty of writing it recursively is that in some sense once this piece of code works for some simple cases, the idea that setting it to larger cases is almost – I don't wanna say guaranteed. That maybe makes it sound too easy, but in fact, if it works for cases of N and then N plus one, then it'll work for 100, and 101, and 6,000, and 6,001, and all the way through all the numbers by induction. Question?

Student: You have to remove all the [inaudible], all the spaces?

Instructor (Julie Zelenski):Yeah. So definitely like the was it a car to match I should definitely be taking my spaces out of there to make it right. You are totally correct on that. So let me show you one where recursion is definitely gonna buy us some real efficiency and some real clarity in solving a search problem. I've got a vector. Let's say it's a vector of strings. It's a vector of numbers. A vector of anything, it doesn't really matter. And I wanna see if I can find a particular entry in it. So unlike the set which can do a fast contains for you, in vector if I haven't done anything special with it, then there's no guarantee about where to find something. So if I wanna say did somebody score 75 on the exam, then I'm gonna have to just walk through the vector starting at Slot 0, walk my way to the end, and compare to see if I find a 75. If I get to the end and I haven't found one, then I can say no. So that's what's called linear search. Linear kind of implies this left to right sequential processing. And linear search has the property that as the size of the input grows, the amount of time taken to search it grows in proportion. So if you have a 1000 number array, and you doubled its size, you would expect that doing a linear search on it should take twice as long for an array that's twice as large. The strategy we're gonna look at today is binary search which is gonna try to avoid this looking in every one of those boxes to find something. It's gonna take a divide and conquer strategy,

and it's gonna require a sorted vector. So in order to do an efficient lookup, it helps if we've done some pre-rearrangement of the data. In this case, putting it into sorted order is gonna make it much easier for us to be able to find something in it because we have better information about where to look, so much faster. So we'll see that surely there was some cost to that, but typically binary search is gonna be used when you have an array where you don't do a lot of changes to it so that putting it in a sorted order can be done once, and then from that point forward you can search it many times, getting the benefit of the work you did to put it in sorted order. If you plan to sort it just to search it, then in some sense all the time you spent sorting it would kind of count against you and unlikely to come out ahead. So the insight we're gonna use is that if we have this in sorted order, and we're trying to search the whole thing – we're looking for let's say the No. 75 – is that if we just look at the middlemost element, so we have this idea that we're looking at the whole vector right now from the start to the end, and I look at the middle element, and I say it's a 54. I can say if 75 is in this vector because it's in sorted order, it can't be anywhere over here. If 54's right there, everything to the left of 54 must be less than that, and 75 wouldn't be over there. So that means I can just discard that half of the vector from further consideration. So now I have this half to continue looking at which is the things that are the right of 54 all the way to the end. I use the same strategy again. I say if I'm searching a vector – so last time I was searching a vector that had 25 elements. Now I've got one that's got just 12. Again, I use the same strategy. Look at the one in the middle. I say oh, it's an 80, and then I say well the number I'm looking for is 75. It can't be to the right of the 80. It must be to the left of it. And then that lets me get rid of another quarter of the vector. If I keep doing this, I get rid of half, and then a quarter, and then an eighth, and then a 16th. Very quickly, I will narrow in on the position where if 75 is in this vector, it has to be. Or I'll be able to conclude it wasn't there at all. Then I kind of work on my in, and I found a 74 and a 76 over there, then I'm done. That base case comes when I have such a small little vector there where my bounds have crossed in such a way that I can say I never found what I was looking for. So let's walk through this bit of code that kind of puts into C++ the thing I just described here. I've got a vector. In this case, I'm using a vector that's containing strings. It could be ints. It could be anything. It doesn't really matter. I've got a start and a stop, which I identify the sub-portion of the vector that we're interested in. So the start is the first index to consider. The stop is the last index to consider. So the very first call to this will have start set to zero and stop set to the vector's size minus one. I compute the midpoint index which is just the sum of the start and stop divided by two, and then I compare the key that I'm looking for to the value at that index. I'm looking right in the middle. If it happens to match, then I return that. The goal of binary search in this case is to return the index of a matching element within the vector, or to return this not found negative one constant if it was unable to find any match anywhere. So when we do find it at whatever the level the recursion is, we can just immediately return that. We're done. We found it. It's good. Otherwise, we're gonna make this recursive call that looks at either the left half or the right half based on if key is less than the value we found at the midpoint, then the place we're searching is – still has the same start position, but is now capped by the element exactly to the left of the midpoint, and then the right one, the inversion of that, one to the right of the midpoint and the stop unchanged. So taking off half of the elements under consideration at each stage, eventually I will get down to the simplest possible case. And the simplest possible

case isn't that I have a one-element vector and I found it or not. The really simple case is actually that I have zero elements in my vector that I just kept moving in the upper and lower bound point until they crossed, which meant that I ran out of elements to check. And that happens when the start index is greater than the stop index. So the start and the stop if they're equal to each other mean that you have a one element vector left to search, but if you – For example, if you got to that case where you have that one element vector left to search, you'll look at that one, and if it matches, you'll be done. Otherwise, you'll end up either decrementing the stop to move past the start, or incrementing the start to move past the stop. And then that next iteration will hit this base case that said I looked at the element in a one-element vector. It didn't work out. I can tell you for sure it's not found. If it had been here, I would've seen it. And this is looking at just one element each recursive call, and the recursive calls in this case stack up to a depth based on the logarithm of the size to the power of two, so that if you have 1000 elements, you look at one, and now you have a 500-element collection to look at again. You look at one, you have a 250 element, 125, 60, 30, 15. So at each stage half of them remain for the further call, and the number of times you can do that for 1000 is the number of times you can divide 1000 by two, which is the log based two of that, which is roughly ten. So if you were looking at 1000 number array, if it's in sorted order, it takes you ten comparisons to conclusively determine where an element is if it's in there, or that it doesn't exist in the array at all. If you take that 1000 element array, and you make it twice as big, so now I have a 2000 number array, how much longer does it take?

Student: One more step.

Instructor (Julie Zelenski):One more step. Just one, right? You look at one, and you have a 1000 number array, so however long it took you to do that 1000 number array, it takes one additional comparison, kinda one stack frame on top of that to get its way down. So this means actually this is super efficient. That you can search so for example a million is roughly two the 20th power. So you have a million entry collection to search, it will take you 20 comparisons to say for sure it's here or not, and here's where I found it, just 20. You go up to two million, it takes 21. The very slow growing function, that logarithm function, so that tells you that this is gonna be a very efficient way of searching a sorted array. A category thing called the divide and conquer that take a problem, divide it typically in half, but sometimes in thirds or some other way, and then kind of – in this case it's particularly handy that we can throw away some part of the problem, so we divide and focus on just one part to solve the problem. All right. So this is the first one that's gonna start to really inspire you for how recursion can help you solve problems that you might have no idea how to approach any other way than using a recursive formulation. So this is an exercise that comes out of the reader in Chapter 4, and the context of it is you have N things, so maybe it's N people in a dorm, 60 people in a dorm, and you would like to choose K of them. Let's K a real number, four – four people to go together to Flicks. So of your 60 dorm mates, how many different ways could you pick a subset of size four that doesn't repeat any of the others? So you can pick the two people from the first floor, one person from the middle floor, one person from the top floor, but then you can kind of shuffle it up. What if you took all the people from the first floor, or these people from that room, and whatnot? You can imagine there's a lot of kind of

machinations that could be present here, and counting them, it's not quite obvious unless you kinda go back to start working on your real math skills how it is that you can write a formula for this. So what I'm gonna give you is a recursive way of thinking about this problem. So I drew a set of the things we're looking at? So there are N things that we're trying to choose K out of. So right now, I've got 12 or so people, or items, whatever it is. What I'm gonna do is I'm gonna imagine just designating one totally at random. So pick Bob. Bob is one of the people in the dorm. I'm gonna kind of separate him from everybody else mentally in my mind, and draw this line, and kinda mark him with a red flag that says that's Bob. So Bob might go to Flicks or might not go to Flicks. Some of the subsets for going to Flicks will include Bob, and some will not. Okay. So what I'd like to think about is how many different subsets can I make that will include Bob, and how many different subsets can I make that don't include Bob. And if I added those together, then that should be the total number of subsets I can make from this collection. Okay, so the subsets that include Bob – so once I've committed to Bob being in the set, and let's say I'm trying to pick four members out of here, then I have Bob and I have to figure out how many ways can I pick three people to accompany Bob to the Flicks. So I'm picking from a slightly smaller population. The population went down by one, and the number I'm seeking went down by one, and that tells me all the ways I can pick three people to go with Bob. The ones that don't include Bob, and Bob's just out of the running, I look at the remaining population which is still one smaller, everybody but Bob, and I look for the ways I can still pick four people from there. So what I have here is trying to compute C of NK is trying to compute C of N minus one K minus one and add it to C of N minus one K. So this is picking friends to accompany Bob. This is picking people without Bob. Add those together, and I will have the total number of ways I can pick K things out of N. So we're very much relying on this recursive idea of if I had the answer – I don't feel smart enough to know the answer directly, but if I could defer it to someone who was actually willing to do more scrutiny into this thing, if you could tell me how many groups of three you can join with Bob, and how many groups of four you can pick without Bob, I can tell you what the total answer is. The simplest possible base case we're gonna work our way down to are when there are just no choices remaining at all. So if you look back at my things that are here, in both cases the population is getting smaller, and in one of the recursive calls, the number of things I'm trying to pick is also getting smaller. So they're both making progress toward kind of shrinking those down to where there's kind of more and more constraints on what I have to choose. For example, on this one as I keep shrinking the population by one and trying to get the same number of people, eventually I'll be trying to pick three people out of three, where I'm trying to pick of K of K remaining. Well, there's only one way to do that which is to take everyone. On this one, I'll eventually keep picking people, so the K is always less than the N in this case. The K will eventually kinda bottom out, or I'll say I've already picked all the people. I've already picked four people. I need to pick zero more. And at that point, that's also very easy, right? Picking zero out of a population, there's only one way to do that. So what we end up with is a very simple base case of if K equals zero – so I'm not trying to choose anymore. I've already committed all the slots. Or if K is equal to N where I've discarded a whole bunch of people, and now I'm down to where I'm facing I've gotta get four, and I've got four left. Well, there's only one way to do those things, and that's to take everybody or to take nobody. And then otherwise, I make those two

recursive calls with Bob, without Bob, add them together to get my whole result. That's wacky. I'm gonna read you something, and then we'll call it a day. I brought a book with me. I stole this from my children.

[End of Audio]

Duration: 48 minutes