ProgrammingAbstractions-Lecture11

Instructor (Julie Zelenski):Hi there. Good afternoon. Welcome to Monday. Today's a big day. A big day, right? We've got a couple of recursive backtracking examples that we didn't get to on Friday that I'm gonna talk you through today, and then we're gonna talk a little bit about pointers, the long anticipated introduction to one of the scarier parts of the C++ language as kind of a step toward building linked lists and the recursive data idea that we will study today and continue into Wednesday.

And so the material at this point jumps around a little bit, right? We go back and pick up some of the pointers in array information that was in earlier chapters. Linked lists are covered a little bit later in kind of a different context that is – you can do but it's not the best match to how we're covering it here. And then handout 21, which I gave out today, is more similar to the way I'm going to be showing you linked lists and its concepts.

Once we get the linked list up, we go back to the reader of chapter seven looking at algorithms and big O. We'll spend actually several days on that on sorting, and analysis of algorithms, and things like that. [Inaudible] you guys should be working on that, right, coming in on Wednesday, right, some good practice getting your recursive decomposition skills down and figuring out how to work your way toward the base case and things like that.

And then what goes out at [inaudible] will be actually kind of your first really big complete program, right, it is the venerable bottle that you may have heard of because actually it's such a legend in the 106 program that kind of brings together a lot of the stuff, ADTs, and recursion, and all sorts of things we study all term kind of build one big complete program now that we've kind of got a bunch of skills to put together on that, which will go out on Wednesday when assignment three comes in.

Note that tomorrow's Super Tuesday, so if you are resident of a state who is one of the 24 or so who are participating in tomorrow's big primary, be sure to get out and vote. Anything administratively you want to ask about? Questions? How many people have done, you know, at least one of the problems on the recursion problem set now? Oh, yeah, yeah. How many of you have done all of them?

Not quite. Okay. Anybody who's gotten along the way have any insights that they want to offer up to their – those who are a little further behind you? Any way of lending a hand to your fellow student?

Student: Draw a diagram.

Instructor (Julie Zelenski):Draw a diagram. What kind of diagrams have you drawn?

Student: Like, each step [inaudible] trying to figure out what it's doing if I go all the way down and it's a little difficult.

Instructor (Julie Zelenski):So he's suggesting here, right, start with, you know, one of your bigger cases. Maybe that's gonna take four or five, you know, calls before it hits that base case, and watch it do its work, right, think about, okay, what the first call makes, what the second call makes, what the third call makes, make sure you're working toward that base case, and see how it both goes down into the calls and then unwinds its way back out, right, can definitely help a lot.

For the ones that have a pretty high branching factor, that gets a little bit tricky, right, to sort of – [inaudible] has a five way branch with a five way branch under it, it would go a little crazy, so you'd have to pick some pretty small examples for the more complex problems. But certainly for the simple cases, right, being able to do that. Anything else? Yeah.

Student:[Inaudible].

Instructor (Julie Zelenski):Yes. So the [inaudible] we gave you, right, you really do need to match our prototypes, but they are very likely in many cases to not be enough, right, they'll get you started but there's gonna be more housekeeping. You're gonna be keeping them along the way, so probably a lot of them are just gonna be those one line wrappers that make a call into your real recursive function that then picks up the outer state plus some other kind of housekeeping to work its way down the recursive call.

So yeah, it's definitely true. A lot of little one-line wrappers in our prototype going into your recursive call. Over here.

Student: This doesn't really have to do with recursion, but go back to [inaudible] I guess C++ or header file, you need to, like, physically move it to the right folder.

Instructor (Julie Zelenski):Yes. Yeah, sure.

Student: Add it in Visual Studio doesn't do it. It never compiles.

Instructor (Julie Zelenski):Yes, so you – when we give you a .cpp file, right, with some code included in the project, you really have to get it into the right place and get your project to include it, otherwise it'll turn up saying I've never heard of this lexicon, you know, it will fail to compile or link one or the other, depending on which step it got hung up on. So if we give you some new code, make sure you incorporate it into your project, right, so that it actually is kind of built into it, and you can use that code in solving your problems. Anything else? Okay. Oh, wait.

Student:[Inaudible].

Instructor (Julie Zelenski):The what?

Student: Failure cases?

Instructor (Julie Zelenski):Yes, failure cases, right? Like if – often you get focused on what the truth will be, what the right answer – get to the success case, and then kind of completely ignore these other things about what about the dead ends, right, the things that are going nowhere. For example, on the phone T9 Text one, right, there definitely are some cases where you have to kind of stop things going down dead ends, and if you don't, right, you can get into this sort of nasty exhaustive, you know, infinite recursion that can really make quite a mess of things.

So making sure you're thinking both about how you know when you got to where you want to be, and where you get to something that you don't want to be but that you can back out of. So the two samples I want to do are both based on the same backtracking pseudocode that we were using on Friday. I just want to go through them. I'm gonna do a little bit less attention to the code and a little bit more attention to the problem solving because at some point I think the problem solving is really where the tricky stuff comes on.

And then the kind of – turning it into code, there's some details there but they're not actually as important, so I'm gonna de-emphasize that just a little bit here and think more about solving it. So this is the pseudocode for any form of a backtracker, right, that has some sort of, you know, failure and success cases, like when we run out of choices, we've hit a dead end, or we've hit a goal state, we've – you know, there's no more decisions to make, right, is this want to be, yes or no, and then otherwise there are some decisions to make.

And for all the possible decisions we could make, we're gonna try one, be optimistic about it working out, make that recursive call that says that choice was where I wanted to be. If it returns true, or whatever the success return value is, then we return true, right? No need to look any further. That choice was good enough. If it didn't work then we gotta unmake – try some other choices. If we try all the things available to us and no case, right, did solve ever return true, then we can only conclude that the configuration as given to this call was unsolvable, which is where that return false causes it to back up to some earlier call and start unmaking that next layer of decisions, and then recur further down, eventually either unwinding the entire recursive [inaudible] all the way back to the beginning and saying it was totally unsolvable no matter what, or eventually finding some sequence of decisions that will lead to one that will get us to a success case. So the one I'm gonna show you here is the sudoku, which is those little puzzles that show up in newspapers everywhere. They're actually not attributed to – apparently, to the Japanese, but it apparently got a lot more popular under its Japanese name than it did under the original English name for it. And the goal of a sudoku, if you haven't ever done one, right, is it's a nine by nine grid in which you're trying to place numbers, and the requirement for the numbers is such that within any particular row, or any particular column, or any particular block, which is a three by three subsection of that, the numbers one through nine each appear once. So you can never have more than two ones in a row, two twos in a column, three threes in a block, or anything like that. And so there has to be some rearrangement, or, in fact, permutation of the numbers one through nine in each row, in each column, and each block, such that the whole puzzle kind of works out logically. So when it's given to you, you know, usually some fraction of the slots are already filled in, and the goal for you is to fill in those remaining slots without violating any of these rules. Now the sort of pure sudoku solvers don't really use guessing. It's considered, actually, poor form, you know, that you're supposed to actually logically work it out by constraints about what has to be true here versus what has to be true there to kind of realize that – what choices you have. We're actually not gonna be a pure, artistic, you know, sudoku solver. What we're gonna do is we're actually gonna use a brute force recursive algorithm that's just gonna try recursive backtracking, which is to say, make an assignment, be optimistic about it, and if it works out, great, and if not, we'll eventually come back to that decision and revisit it. So what we have here basically is a big decision problem. You know, of the 81 squares on here, you know, about 50 of them, right, need to be chosen. For each of those 50 squares, right, we're gonna do them one at a time and recur on the remaining ones. So, you know, choose one then we'll just go left to right from the top. Choose that one at the top, make an assignment that works, and so that's what we'll use the context we have in problem here. So, for example, if you look at this first row, there's a one in this column so we can't use one. There's a two in that block so we can't use two, but there's not a three in either that row, or that column, or that block, so we'll say, well, three looks good, you know, just trying the numbers in order. We'll be optimistic, say that works, and say, well if we planted a three here, could we recursively solve the remaining 49 holes and work it out? And so we get to that next one – we look at this one and say, okay, well we could put a one here, right, because there's not a one in this column, not a one in that row, not a one in that block, so we'll kind of move on. I'm gonna do a little demo of that for you. And maybe it's to kind of keep moving our way across, and only when we get to a dead end based on some of our earlier decisions will we unmake and come back. So let me – okay. So that's the same set of numbers there. So I put the three up in the corner. And so it puts the one here thinking, okay, that looks good. So it gets to the next square over here and then the one can't be used because it's actually already in use both in that column and in the row we're building so far, but two can be used. Two doesn't conflict with anything we have so far. And so it just keeps going optimistically, right? At that stage over there it turns out almost all the numbers are in use. Most of the numbers all the way up through seven and nine are there, and seven's in its column. So, in fact, the only number that that could possibly be is nine, so the only choice we have here to try is nine, and then we'll place the seven next to it. And so now we have a whole row that doesn't conflict with any of its blocks this way, and then we just keep moving on, right, so that is to keep kind of going from top to bottom, left to right. We'll place that four. We'll place another one. Place a three. So it's actually choosing them – it's actually going in order from one to nine just picking the first one that fits, right, so when we get to the next square here, right, it can't use one, it can't use two, it can't use three, it can't use four, it can't use five because they're all in that row. It can't use six because it's in the column. It can't use seven because it's in that row, but it should be able to use eight, and so it'll place an eight there. So it just kind of examines them in sequence until it finds the first one that doesn't violate any things already decided, and then kind of moves optimistically forward. So about this point, right, we're doing pretty well, but we're starting to run into some troubles if you look at the next one over here. It'll place the six there, but then it will – once the six is placed there, then it looks to the right and it says, oh, I need to put a nine there. That's the only number left. It tries all the numbers one, two, three, four, and it says that isn't gonna work. And so it actually fails on the rightmost column, which causes it to back up to the one right before and it says, well, why don't you try something else here? Well, it looks at seven, eight, and nine, none of those can work either, so it's actually gonna back up even further and then say, well what if we try to put a nine here? That also doesn't work. So now it's gonna start seeing that it's kind of unwinding as the constraints we have made have kind of got us down a dead end and it's slowly working its way back out to where the original bad decision was made. So it tries again on moving nine in here, and moving across, right, but again, kind of, you know, working its way forward but then kind of backing its way up. And let me go ahead and just run it faster so you can kind of see it. But, you know, it's working on that row for a while, but essentially to note that the top row stays relatively constant. It kind of believes that, well, that first three must have been good because, you know, we're getting somewhere on that, and it keeps kind of going. You can see that the activity is kind of always at the kind of tail end of that decision making, which eventually, right, worked its way out. And so it turns out, like, those three ones that we did put in the first spots were fine. That is, choices, right, it did work out. We didn't know that when we started, right, but it was optimistic, right, it put it down and then kept moving, and then eventually, right, worked out how the other things that had to get placed to make the whole puzzle be solvable. And so this thing can solve, actually, any solvable sudoku, right, and if it's not animating, instantaneously, even though it is really doing it in a fairly crude way, right, it's basically just trying everything it can, moving forward, and only when it kind of reaches a contradiction based on some of those choices that they will – that can't possibly be because at this point I'm forced into a situation where there's nothing that works in this square, so it must be that some earlier decision was wrong. And you notice that when it backs up, it backs up to the most immediate decision. So if you think of it in terms of recursive call, here's your first decision, your second decision, your third decision, your fourth decision, your fifth decision. If you get down here and you're, like, trying to make your eighth decision, and there's nothing that works, right, the decision that you come back to will be your seventh one. The one right before it. You don't go throw everything away and start over. You don't go all the way back to the beginning and say, oh, well that didn't work, let's try again. It's that you actually use the kind of context to say, well, the last decision I made was probably the one that needs a little fixing. Let me just back right up to that one. That's the way the calls unwind, and it says we'll pick up trying some other options there. Which ones have we not tried on that one? And then go forward again, and again, if you get down to that eighth decision and you're still stuck, you come back to the seventh decision again, and only after kind of the seventh decision has gone back and forth with the eighth unsuccessfully through all its options would we eventually return to that sixth decision, and potentially back to the fifth, and fourth, and so on. The code for this guy, a little abstracted that should very much fit the pattern of what you think recursive backtracking looks like, and then the kind of sort of goofier parts that are about, well, what does it mean to test a sudoku for being a sign of having conflicts is actually then passed out into these helper functions that manage the more domain specific parts of the problem. So at the very beginning it's like, find an unassigned location on the grid, and it returns the row and column by reference. It turns out in this case those are reference parameters. So [inaudible] searches from top to bottom to find the first slot that actually does not have a value currently in it. If it never finds one, so exhaustively searched the grid and didn't find one, then it must be that we have a working sudoku because we never put a number in unless it worked for us, and so if we've managed to assign them all, we're done. If this didn't return true, that meant it found one, and it assigned them a row and column, and then what we're gonna go through the process of is assigning that row and column. So we look at the numbers one through nine. If there are no conflicts for that number, so it doesn't conflict with the row, column, or block, that number isn't already in use in one of those places, then we go ahead and make the assignment, and then we see if we can solve it from here. So having updated the grid to show that new number's in play, you know, if we move on, the next [inaudible] of sudoku will then do a search for find unassigned location. This time the one that we previously found, right, has been assigned, so it actually won't get triggered on that one. It'll look past it further down into the puzzle, and eventually either find the next one to make a call on, and kind of work its way through, or to – you have to solve the whole thing. If it didn't work, so we made that assignment to the number nine, and we went to solve, and eventually this tried all its possibilities from here and nothing came up good, then this unassigned constant is used to unmake that decision, and come back around, and try assigning it a different number. If we try all of our examples, so for example, if we never find one that doesn't already conflict, or if we try each one and it comes back false, right, that this return false here is what's triggering the backtracking up to the previous recursive call to reconsider some earlier decision – the most recent early decision for this one, and say that was really our mistake, right, we've got to unmake that one. So it should look like kind of all the recursive backtracking all through looks the same, right? You know, if we're at the end, here's our base cases for all our options. If we can make this choice, make it, try to solve it, be optimistic, if it works out, return. Otherwise, unmake it, allow the loop to iterate a few more times trying again. If all of those things fail to find a solution then that return false here will cause the backtracking out of this decision. I just let it become constant. You know, I made it up. I used negative one, in fact, just to know that it has no contents. Just specific to sudoku in this case. Now would be a great time to ask a question. You guys kind of have that sort of half-okay look on your face. It could be I'm totally bored. It could be I'm totally lost.

Student:

Where do row and column get assigned?

Instructor (Julie Zelenski):So they – finding assigned location takes [inaudible] reference. So if you look at the full code for this – this is actually in the handout I gave out last time, and so there's a pass by reference in that function, and it returns true if it assigned them something, and then they have the coordinates of that unassigned location. Question over here.

Student: How'd you know to write sol sudoku as a bool rather than as, like, a void function?

Instructor (Julie Zelenski):Well, in this case – that's a great question, right, in most cases in recursive backtracking I am trying to discover the success or failure of an operation, and so that's a good way to tell because otherwise I need to know did it work. And so if I made it void then there had to be some other way I figured it out. Maybe, you know, that you have to check the grid when you're done and see that it has – no longer has any unassigned locations, but the bool is actually just the easiest way to get that information out.

So typically your recursive backtracking machine will probably return something. Often it's a true or false. It could be, you know, in some other case, just some other good know value, and some other sentinel that says, you know, bad value. So, for example, in the finding an anagram of the words it might be that it returned the word if it found one, or returned an empty string if it didn't. So using some sort of state that says here's how – if you made the call, you'll know whether it worked because we really do need to know. Make the call and find out whether it worked. Well, how are we gonna find out? One of the best ways to get that information is from a return value. Way in the back?

Student: Exactly does the return calls trigger in the backtracking [inaudible]?

Instructor (Julie Zelenski):So think about this as that, you know, it's hard to think about because you have to kind of have kind of both sides of the recursion in your head, but when – let's say we're on the fifth decision, right? We make the call to solve the sudoku that's gonna look at the sixth decision, right? If the sixth decision fails, it says, I looked at all my choices and none of them worked, right? It's that that causes the fifth decision to say, well I – here go solve for the sixth decision down.

Well the sixth decision came back with a false. It got to this case after trying all its options, then it's the fifth decision that then says, okay, well then I'd better unmake the decision I made and try again. And so at any given stage you can think about the caller and the callee, it's saying, I'm making a decision. You tell me if you can make it work from here. If the delegate that you passed on the smaller form of the recursive problem comes back with a return false, and then I've tried everything I could possibly do, but there was no way.

The situation you gave me was unworkable. And that says, oh, okay, well you know what, it must have been my fault, right? I put the wrong number in my box. Let me try a different number and now you try again. And so they're constantly kind of these – you have to keep active in your mind the idea that there's this whole chain of these things being stacked up, each of them being optimistic and then delegating down.

And if your delegate comes back with the – there was nothing I could do, then you have to revisit your earlier optimistic decision, unmake it, and try again. And so that – this is the return false to the outer call, the one that made the solved call that unwinds from. Way over here?

Student:[Inaudible]?

Instructor (Julie Zelenski):Pretty much any recursive piece of thing can be reformed into a backtracking form, right? You need to – to do a little bit like the [inaudible] we tried last time, we'll take a void returning thing and make it return some true or false, and there's kind of, like, make a decision and be optimistic, see if it worked out. But that kind of rearrangement of the code should work with pretty much all your standard recursion stuff.

So any kind of puzzle that actually, like, all these kind of jumble puzzles, and crossword puzzles, and things that involve kind of choosing, right, can be solved with recursive backtracking. It's not necessarily the fastest, right, way to get to a solution, but it will exhaustively try all the options until a sequence of decisions, right, leads you to a goal if there is a goal to be found.

And so that if you can think of it as – if you can think of your problem as a decision problem – I need to make some decisions, and then from there make some more decisions and so on, and eventually, right, I will bottom out either at a goal or dead end, then you can write a recursive problem solving – backtracker to solve it. Way in the back?

Student: This doesn't have anything to do with recursion, but how did you slow down the simulation?

Instructor (Julie Zelenski):How did I slow it down? I'm just using pause. There's a pause function in our stenographics library, and you just give it a time. You say one second, half a second, and just – I use that a lot in our demos, for example, for the maze, so that you can actually see what's going on, and that way it just animates a little more. Stenographics, CSTgraph.h. [Inaudible] me show you one more kind of just in the same theme. The idea of taking sort of some, you know, puzzle that somebody might give you, trying to cast it in terms of a decision problem, right, where you make decisions and then you move on, can I solve something like these little cryptographic puzzles? So here's send plus more equals money. I've got eight digits in there – eight letters in there. You know, D E M N O R S Y across all of them, and the goal of this puzzle is to assign these letters – eight letters to the digits zero through nine without replacement so that if D's assigned to three, it's assigned to three in all places. For example, O shows up two places in the puzzle. Both of those are either a two or both are three. There's not one that's one and one that's another. And each digit is used once. So, like, if I assigned two to O, then I will not assign two to D. So what we've got here is eight letters. We've got 10 digits. We want to make an assignment. What we've got here is some choices. The choices are, for each letter, what digit are we gonna map it to? Another way to think about it is if you took these eight letters, and then you attach two dashes on the end, then you considered that the letter's position in the string was – the index of it was its digit. It's really just like trying the permutations, right, rearranging those letters in any of those sequences. So right now, for example, maybe D is assigned zero, and one, and two, and so on, and these guys are eight, nine. Well, if you rearrange the letters into some other permutation then you've made a different assignment. So in effect, right, what you're looking at is a problem that has a lot in common with permutations, right? I'm gonna make an assignment, take a letter off, decide what index it's gonna be placed at, so it's kind of like deciding where it would go in the permutation, and then that leaves us with a smaller form of the problem which has one fewer letters to be assigned, and then recursively explore the options from there to see if we can make something that makes the addition add up correctly that D plus E equals Y means that if D was assigned five, and E was three, then Y better be eight for this to work out. So the first form I'm gonna show you is actually just the dumb, exhaustive recursive backtracking that works very much like the sudoku problem where it just – it finds the next unassigned letter, it assigns it a digit of the next auto sign digits, right, and then just optimistically figures that'll work out. After it makes an assignment for all the letters it says, take the puzzle, convert all the letters into their digit equivalents, and see if it adds together correctly. If so, we're done. If not, then let's backtrack. So let me – I mean, actually, I'll show you the code first because it's actually quite easy to take a look at. Again, it has helper routines that kind of try to abstract the pieces of the puzzle that actually aren't interesting from kind of looking at just the core of the recursive algorithm, so it looks a lot like sudoku in that if letters do assign, so it actually keeps the string of the letters that haven't yet been assigned. If there are no more letters in that string, we take one off at each time we assign it, then we check and see if the puzzle's solved. So that kind of does the substitution, and does the math, and comes back saying yes or no. If it worked out, we'll get a true. If it didn't work out we get a false. If we still have letters to assign then it goes through the process of making a choice, and that choice is looking at the digits zero through nine if we can assign it. So looking at that first letter in that digit, and then that's making sure that we don't already have an assignment for that letter, that we don't have an assignment for that digit. Sort of make sure that just the constraints of the problem are being met. If we're able to make that assignment then we go ahead and make a recursive call, having one fewer letters to make a decision for, and if that worked out true, otherwise we do an unassignment and come back around that loop and then eventually the same return false at the bottom, which said, well given the previous assignments of the letters before we got to this call, there was nothing I could do from here to make this work out. So let me show you this guy working because you're gonna see that it is actually a crazy way to try to solve this problem in terms of what you know about stuff. So I say C S plus U equals fun, a well-known equation. So I did this little animation to try to get you visualized what's going on at any given stage. So it has the letters down across the bottom, S U N C O Y F. That's the string of letters to assign. It's gonna assign it the next available digit when it makes that recursive call, so the first recursive call is gonna be about assigning S, and then the next call will be assign U, and the next call – and so on. So it always gets up to seven deep on any particular thing. And so it says, okay, well first digit available is zero, go ahead and assign that to S. So it's gonna make a call there. And then it says, okay, well look at U. Well we can't use zero because zero's already in use. What don't we assign U to be one? Okay. Sounds good. Keep going. And then it'll say, okay, let's get to N. Let's make an assignment for N. It says I'll look around. Okay, well zero's in use, one's in use, how about two? Okay. Good. And then it assigns this to three, this to four, this to five, and this to six. Okay. It gets to here and it says, hey, does that work, 30 plus 541 equals 612? No? Okay. Well, you know what the problem was? It was F, right? I assigned F to six. How stupid of me. It can't be six. Let's make it seven. And then it says, oh, oh no, oh I'm sorry. I'm sorry, 30 plus 541 that's not 712. How silly. I need to assign F to be eight. And it's gonna do this for a little while. A long while [inaudible]. And then it says, oh, okay. Well, you know what, given the letters you'd assigned to the first six things, when you got to F, I tried everything I could and there was nothing I could do to fix this. You know what the problem was? It's not me. It's not me. I'm not the problem. You're the problem. So it returns false from this call at the bottom, having tried all its options, which causes Y to say, oh, yeah, yeah, I know. I know I said five. Did I said five? I didn't meant to say five. I meant to say six. And so it moves up to the six option. Again, optimistically saying that's good. Go for it. See what you can do. So it picks five. That won't work, right? It picks seven. It's gonna go for a long time, right, because it turns out, right, this is one of those cases where that very first decision, that was one of your problems, right? If you assign S to be zero, there's nothing you can assign U and N to be that are gonna work out. So what it's gonna be going through is this process though of having, you know, committed to this early decision and kind of moving on it's gonna try every other variation over here before it gives up on that. So let me set it to going. Even though C S plus U does equal fun. I guarantee it. We'll let it do some work for a while. So those bars is they grow up are desperation. You can think of that as, like, it's running out of options. It's running out of time, you know, and it's like oh, oh, wait, earlier, back up, back up. And so, okay, you can kind of see how far its backed up but sort of how tall some of the deeper recursive calls, right, the earlier ones in the sequence. And so it doesn't, you know, revisit any of these decisions because it's really busy over here, but you can see that C is now up to seven. It'll get up to eight. It'll get up to nine. And that was when it will cause itself to say, you know, I tried everything that was possible from C on down, and there was no way to make this thing work out. It must be that the bad decision was made earlier. Let's back up and look at that. And so it'll come back to the decision N, bump it up by one, and then go again. It's gonna do this a long time, right? Go through all the options for N and its neighbors before it comes back and revisits the U. It's gonna have to get U all the way up, right, through having tried one, two, three, four. So adding zero to one, and two, and three, and four, and discovering it can never make a match over there before it will eventually decide that the S is really where we got ourselves in trouble. So this is kind of in its most naïve form, right? The recursive backtracking is doing basically permutations, right? You can see this is actually just permuting the digits as they're assigned to the numbers, and there are, in this case, you know, seven factorial different ways that we can make this assignment, and there are a lot of them that are wrong, right? It's not being at all clever about how to pick and choose among which to explore. So in its most naïve form, right, recursive backtracking can be very expensive because often you're looking at things that have very high branching, and very long depth to them, which can add up to a lot of different things tried. Just using some simple – in this case, heuristics, sort of information you know about the domain can help you to guide your choices instead of just making the choices in order, trying the numbers zero through nine as though they're equally likely, or kind of waiting to make all the assignments before you look at anything to see if it's actually good. There's actually some ways we can kind of shape our decision making to look at the most likely options before we look at these more first. Finding the problem instead of niggling around this dead end. So I'm gonna let that guy work for a little bit while I show you another slide over here. So what the smarter solver does, is that it doesn't really consider all permutations equally plausible. We're gonna use a little grade school addition knowledge. If I look at the rightmost column, the least significant digit, I'm gonna assign

D first. I assign D to zero. I assign E to one, and then I look at Y – I don't try Y is five, or seven, or anything like that. I say there's exactly one thing Y has to be, right, if D is zero and E is one, then Y has to be one, and I can say, well that won't work because I already used one for E. So it'll say, well that's impossible. Something must be wrong early. Rather than keep going on this kind of dumb dead end, it's to realize right away that one of the decisions I've already made, D and E, has gotta be wrong. So it'll back up to E. It'll try two, three, four, very quickly realizing that of all the things you could assign to E, once you have assigned D to zero, you're in trouble. And so it will quickly unmake that decision about D and work its way down. So using the kind of structure of the problem. So it makes it a little bit more housekeeping about where I'm at, and what I'm doing, and what's going on, but it is using some real smarts about what part of the tree to explore, rather than letting it kind of just go willy nilly across the whole thing. Let me get that out of the way. And I'll run this one. I say C S plus U equals fun. Okay. So it goes zero here, and tries the one, and it says no, that won't work. How about the two? No, that won't work, right, because there's nothing you can assign the N that'll make this work. And so it immediately is kind of failing on this, and even after it tries kind of all nine of these, it says none of these are looking good, then it comes back to this decision and says, no, no, actually, S's a zero. That wasn't so hot. How about we try S as one? And then kind of works its way further down. Hello? Okay. So let me try this again. Let me get it to just go. Okay. So it took 30 assignments – 30 different things it tried before it was able to come up with 41 plus 582 equals 623, which does add correctly. It didn't have to unmake once it decided that S was one. It turns out that was a workable solution so it didn't have to go very far once it made that commitment, and then you can see it kind of working its way up. So 30 assignments, right, across this tree that has, you know, hundreds of thousands in the factorial, very large number, but very quickly kind of pruning down those things that aren't worth looking at, and so focusing its attention on those things that are more likely to work out using information about the problem. It doesn't really change what the recursion does. It's kind of interesting if you think that the same backtracking and recursive strategy's the same in all these problems, but what you're trying to do is pick your options, like, looking at the choices you have and trying to decide what are the more likely ones, which ones are not worth trying, right, so sort of directing your decision making from the standpoint of trying to make sure you don't violate constraints that later will come back to bite you, and things like that. This one's back here. It's at 13,000 and working hard. Getting desperate. Doing its thing. Still has not unmade the decision about S is zero, so you can get an idea of how much longer it's gonna take. We'll just let it keep going. I'm in no hurry – and let it do its thing. So let me – before I move away from talking about recursion, just try to get you thinking just a little bit about how the patterns we're seeing, right, are more alike than they are different. That solving a sudoku, solving the eight queens, you know, solving the find an anagram in a sequence of letters, that they all have this general idea of there being these decisions that you're making, and you're working your way down to where there's, you know – fewer and fewer of those decisions until eventually you end up, sort of, okay, I'm done. I've made all the decisions I can. What letter goes next, or whether something goes in or out. And the two problems that I call the kind of master or mother problems, right, of permutations and subsets are kind of fundamental to adapting them to these other domains. And so I'm gonna give you a couple examples of some things that come up that actually are just kind of permutations, or subsets, or some variation that you can help me think about a little bit. One that the CS people are fond of is this idea of knapsack feeling. It's often cast as someone breaking into your house. All criminals at heart apparently in computer science. And you've got this sack, and you can put 50 pounds of stuff in it, right, and you're looking around, you know, at all the stuff that's up for grabs in this house that you've broken into, and you want to try to pack your sack, right, so that you've got 50 pounds of the high value stuff. So let's say there's, like, 100 things, right, you could pick up, right, and they weigh different amounts, and they're worth different amounts. What's the strategy for going about finding the optimal combination of things to stick into your sack, right, so that you got the maximum value from your heist? What problem does that look like that you've seen? It's a subset. It's a subset. [Inaudible]. Right? So if you look at the, you know, the Wii, and you say, oh, should I take the Wii? You know, it weighs this much, and it's this much value, well let's try it in and see what happens, right, and see what else I can stuff in the sack with that, right, and then see how well that works out. I should also try it with it out. So while you're standing there trying to decide what to steal, you have to type in all the values of things in every computer program. Go through the kind of machinations of well, try this with that because some things are big, but have a lot of value, but they only leave a little bit of odd space left over you might not be able to use well, or something. But what we're looking for is the optimal combination, the optimal subset. So trying the different subsets tells you how much value and weight you can get in a combination, and then you're looking for that best value you can get. You're the traveling salesman. You've got 10 cities to visit. Boston, New York, Phoenix, Minneapolis, right? You want to cover them in such a way that you spend the minimal amount of time, you know, in painful air travel across the U.S. Well you certainly don't want to be going Boston, New York, right, like, Boston, to L.A., to New York, to Seattle, to D.C., to Los Angeles back and forth, right? There's gotta be a pattern where you kind of visit the close cities and work your way to the far cities to kind of minimize your total distance overall. What problem was that really in disguise? We've got 10 cities. What are you trying to do? Help me out a little. I hear it almost. Louder. Permutations. You've got a permutation problem there, all right? You got 10 cities. They all have to show up, right? And it's a permutation. Where do you start, where do you end, where do you go in the middle, right? What sequencing, right, do you need to take? So what you're looking at is try the permutations, tell me which ones come up short. There are things, right, about heuristics that can help this, right? So the idea that certainly, like, the ones that are closer to you are likely to make a better choice than the longer one. So kind of using some information about the problem can help you decide which ones are more promising avenues to explore. But in the end it's a permutation problem. I'm trying to divide you guys into fair teams. I want to, you know, divide you up into 10 teams to have a kind of head-to-head programming competition. I happen to know all your Iqs. I don't know. Some other, you know, useless fact that perhaps we could use as some judge of your worth. And I want to make sure that each time has kind of a fair amount of IQ power in it relative to the others that I didn't put, you know, all the superstars on one team. What am I doing? How do I divide you guys up? It's a subset problem, right? So if – in this case it's a subset that's not just in or out. It's like which team am I gonna put you in? So I've got 10 subsets to build. But in the end it's an in out problem that looks like, well, I have the next student. Which of the 10 teams can I try them in? I can try them in each of them, right? So in some sense it's trying in the first team, the second team, the third team, right, and then pull the next student, try him in those teams, and then see whether I can come up with something that appears to get a balance ratio when I'm done. It turns out you can take the letters from Richard Millhouse Dickson and you can extract and rearrange them to spell the word criminal. I am making no conclusion about anything, having done that, just an observation of fact. But that sort of process, right, I'm trying to find, well what is the longest word that you can extract from a sequence of letters? What kind of things is it using? Anything you've seen before? Oh, somebody come on. I hear the whispering but no one wants to just stand up and be committed. How about a little of both, right? The what?

Student:

Like the sudoku puzzle?

Instructor (Julie Zelenski):It's a little bit like the sudoku, right? It's got a permutation sort of backing, and it's got a little bit of a subset backing, right? That is that I'm not choosing all the letters from this. And in fact there's a subset process, which is deciding whether it's in or out, and then there's a permutation problem, which is actually rearranging them, right? That picking the C, and the R, and the I, and the M, and then kind of rearranging them.

So in fact it has kind of a little bit of both, right? Which is what sequence of letters can I extract and then rearrange to make a word – would end up using kind of elements of both of those kinds of recursions sort of mixed together so that when you look at a lot of recursion problems, they end up actually just mapping down to one or the other, or maybe a little bit of both.

And so feeling very comfortable with those problems, how you turn them into a recursive backtracker, and how you can recognize, right, their roots inside these other problems, it's kind of a really good first step to becoming kind of a journeyman, in a way, or recursion. So just whenever you see a new problem you start thinking, okay, is it permutations? Is it subset? Or is it something totally new?

It's probably much more likely to be the first two then the third, and so trying to use the ones that you feel really comfortable with to kind of branch out to solve similar problems, right, where the domain is a little different – the structure of the code is still very much the same. All right. I've got 10 minutes to teach you about pointers. This should be no problem. Let's go see what that guy back there's doing.

Oh, look at that, 38,000 assignments, still has not given up on that S is zero. It's a trooper. Okay. So this is definitely gonna be your first introduction on kind of the first day of talking about this. This is not the only day. So don't get to worried about me trying to do it in 10 minutes. What I'm gonna do is give you the kind of – a little bit of the basics today, and we're gonna follow up with some more stuff tomorrow where we start kind of making use of them and doing something kind of interesting with them.

So there is this notion in C++, which is inherited from the C language in fact, of using a variable type called a pointer as part of the things that you operate on. Okay. People tend to have a lot of trepidation. Just the word pointer causes a little bit of fear, kind of to immediately rise in a new programmer. But hopefully we can demystify a little bit, and then also get a little bit of understanding of why there are reasons to be a little bit wary of working with pointers.

A pointer is actually an address. So here's a couple things we have to kind of think about a little bit how the machine works to have a vision of what's going on here, that when you declare variables, you know, here I am in main. And I declare, you know, int Num, and string S, that there is space set aside as part of the working memory of this program that is gonna record whatever I assign to Num or S.

So if I say Num equals 45, what is that? S equals hello, that there has to be memory that's being used to hold onto that. And as you declare variables, right, more of the space is set aside, and, you know, when you initialize it, when you read to it, when you write to it, right, that space is being accessed and updated as you go through. When you open a new scope, right, new variables come in a scope.

When you close that scope, really that function, that space gets de-allocated. All this happens on your behalf without you actually taking a really active role in it, but in fact you do have to have a little bit of understanding of that to kind of idea of what a pointer's about, is that there is this just big sequence of data, starting from an address zero.

You can think of these things as actually having a little number on them. So maybe this is number 1,000, and this is number 1,004. In this case, assuming that we're numbering by the byte, which is the smallest chunk of memory we can talk about, and that an integer requires four of those bytes to store all the different patterns for different kinds of numbers so that the bytes from 1,000 to 1,003 are reserved for holding onto the information about 45.

And then from 1,000 forward – to however big the string is, which we don't even really know how big it is, so we'll kind of leave it a little bit vague here – is gonna be set aside for storing information about that string. Okay. So usually it's of interest to the compiler, right, to know where things are being stored and what's going on. But it also turns out to be somewhat useful for us to be able to talk about something by address.

Instead of saying the variable whose name is Num, I can talk about the variable who lives at location 1,000. And say there's an integer, and I can point to it, or refer to it by saying go look at memory address 1,000, and read or write the contents there that are of integer type. Okay. It seems a little bit odd at first. You're like, well I already have other ways to get to Num. Why is it I want to go out of my way to find another different mechanism that can get me access to that same variable?

And we're gonna see that actually there's a lot of flexibility that is being bought by us adding this to our feature set. We can say, well, I could actually have one copy of something. So imagine that you have a system, like, an access system where you have student records, and they're enrolled in a bunch of different courses, that your enrolled in four, five different courses, that when it comes time for a class to know who's in their list, it might be handy for them, instead of actually having a copy of that student's information to have a pointer to one student record, and have a lot of different placers where there are additional pointers taken out to that same location and says, go look at the student who's at address 1,000.

I have the students at address 1,000, at 1,026, at, you know, 1,035, whatever those places are, and that that's one way I can talk about what students I have, and those same student addresses might show up in someone else's class list in math 51, or physics, you know, 43 or whatever, and that we are all referring to one copy of the student data without duplicating it.

So this idea of being able to refer to stuff not just by name, but by where it lives, is gonna give us some flexibility in terms of how we build things out of that. So let me kind of show you a little bit of the basic operations, and then – and I'll talk a little bit along the way about this thing about why they're scary because using, you know, memory access as your way to get to something is a little bit more error prone and a little bit harder to deal with than some of the more – other operations we have.

So what I'm gonna show you here is a little piece of code. It shows some simple use of pointers. All right. So I'm gonna draw some of the variables that are going on here. This is main and it declared an integer whose name is Num, so I draw a box for that, and it declares two pointer variables. So the addition of the asterisk by the name there says that P and Q are pointers to integers.

So P and Q themselves are variables that live in the stack. So all the local variables we say live in the stack. They are automatically allocated and de-allocated when you enter a routine. The space for them comes into being. When you leave it, it goes away, and that P and Q are designed not to hold integers themselves. They don't hold numbers, but they hold the address of a number somewhere else.

And so the first thing I did with P there was to assign it the address of a new integer variable that came out of the heap. So the new operator is like the new operator in Java. It takes the thing you want to create one of, it makes one of those in the heap, and it returns to you its address.

In that way it works exactly like in Java. So, in fact, Java actually has pointers despite what anybody ever told you, that the way you create objects and you use new to access them and stuff like that is exactly the same as it is in C++ as it is pointers behind the scene. So I say P gets the value of a new integer. This memory over here is called the heap. So this is not to confuse you with the idea of the stack ADT. We've been using the [inaudible] but it does kind of – it helps you to remember that the stack actually kind of, like, by the virtue of the way function calls get made, main calls A, which calls B, which calls C, that that memory actually kind of is laid out like a stack.

The heap is just this unordered crazy pile of stuff. I ask for new integer, so this might be address 1,000. This might be address 1,004. This might be address 1,008. Typically the stack variables are laid out right next to each other. This could be, like, address 32,016 or something over here. Some other larger address. So I've assigned P to be that thing. So what I actually gotten written into P's box is behind the scenes there really is the number, you know, the address, 32,016.

What I'm gonna draw is an arrow to remind myself that what it is is it's a location of an integer stored elsewhere. The de-reference operator, which is the first thing we see on this next line, says to follow P and assign it a 10. So this is taking that address that's in P, using it as a location to go look up something, and it says, and go write to that location in address 32,016 the number 10. And I said Q equals no int.

So Q gets an [inaudible] maybe this is at 23,496. Some other place out there. And so that's actually kind of what's being stored here, 23,496. And then I did this thing where I assigned from D referencing P to get an integer, and assign that onto what Q is that has the effect of kind of following P, reading its 10, and then writing it over here. So copying the integers at the end of those two pointers to make them point to the same value.

So they point to two different locations, but those locations hold the same integer value that is different than the next line. Without the stars, where I said Q equals P, without the stars on it, it's saying take the address that's in P, and assign it to Q, causing Q and P now to be aliases for the same location. So now I have two different ways of getting to that same piece of memory, either by reaching through P and de-referencing, or reaching through Q and de-referencing it – both of them are reading and writing to that same location in the heap, where the 10 is, and then this one down here's no longer accessible. I sort of lost track of it when I overwrote Q with the copy of P.

When I am done in C++ it is my job to delete things that are coming out of the heap. Yes, it should. I'll take that one at 32,016. Whereas in Java, when you say new, and then you stop using something, it figures it out and it does what's called garbage collection to kind of clean up behind you. In C++, things that you new out of the heap are your responsibility to delete.

So you delete something when you're done with it. If I'm no longer using this new integer I created in the heap, I say to delete P to allow that memory to be reclaimed. And so that causes this piece of memory to get marked as freed, or reusable, so that a subsequent new call can have that space again and use it for things. I will note that right now, the code as written has a little bit of an error in it because it says delete P.

And delete P says we'll follow out to 32 [inaudible] and mark it as available. The next thing I did was said delete Q. Well Q points to the same place P did. So in fact I'm saying take that piece of freed memory and mark it freed again. There is no real guarantee about whether that's gonna do something sensible, or whether it's just gonna ignore me.

One thing it could do is just say, well look, that memory's already freed, so you saying to free it twice is kind of stupid. On the other hand, it could still cause some more problems depending on how the heap allocater works, and there's no guarantee. So it becomes very [inaudible] on the programmer to be very careful about this matching, that if you make a new call, you make a delete call.

And if you already made a delete call, you don't make another one accidentally. So I really should have that line out of there. The piece of memory that Q originally pointed to, right, I no longer have any way to get to, and so I have no way of making a delete call to it and freeing it. And so this little piece of memory we call an orphan.

He's stranded out there in the heap, no way to get back to it, and C++ will not automatically reclaim it for us. So we have created a little bit of a mess for ourselves. If we did that a lot, right, we could end up clogging our heap filled with these orphans, and have to keep getting new memory because the old memory, right, was not being properly reclaimed. We're gonna talk more about this, so this is not the all – end all of pointers, but just a little bit to think about today. We'll come back on Wednesday, talk more about it, and then talk about how we can use this to build linked lists, and that will be fun times for all.

[End of audio]

Duration: 52 minutes