ProgrammingAbstractions-Lecture10

Instructor (Julie Zelenski):Hey there. Welcome. Welcome. Welcome. My computer's not quite talking to anything yet here. You guys got an answer for that? Unplug and replug that? Okay, I don't know what's – whether somebody can help make my computer talk to the projector? Yeah. It's plugged in like ordinary. Yeah, I – we do this every day. There we go. Thank you very much. All right, back to your regularly scheduled lecture.

Today I'm gonna keep talking a little bit more about another procedural recursion example and then go on to talk about, kind of, the approach for recursive backtracking, which is taking those procedural recursion examples and kind of twisting them around and doing new things with them. This corresponds with to the work that's in chapter six of the reader.

And then from here we're actually gonna do a little detour to pick up an explanation of recursive data in the form of linked lists, and that's gonna give us a chance to start talking about C++ pointers, which I know everyone has been eagerly awaiting a chance to get a little bit of mystery of the pointers revealed to them. And so the coverage for that is in chapter 2 in the early sections, 2-2 and 2-3. I [inaudible] covered in the text in sort of 9-5. There's actually a handout I'll give out on Monday's lecture that helps to go along with what we're doing there so there's a little bit of supplemental material to kind of work through that. And so it'll be our first chance to see points [inaudible] we can really use them to do something, and so I kind of differed talking about them until now because we don't have a good use until we get there, and that we'll be exploring a recursive data type called the link list. I'll be hanging out at the Turbine Café after class, so if, actually, you're available this afternoon and want to come and hang out, we'll be there for an hour or two, so if you want to walk over with us or come later when you're free, that would be great. If not this Friday, hopefully some other one. How many people started the recursion problem set? Got right on it. How many people have already solved a problem or two? Oh, yeah. How many people solved all of them? Okay. That's good though. And how is it going so far? Thumbs up? Lots of joy? Joy in Mudville. All right. So as I say, the handout, you know, there's six problems, each of them is actually very, very short, and that may lull you into believing that, well, you can start at it right before because it's not gonna be typing speed that's gonna keep you there all night, but it is actually some really dense, complex code to kind of get your head around and think about. And I think, actually, it helps a lot to have started at it, and thought about it, and let it gel for a little time before you have to kind of really work it all the way out. So I do encourage you to get a jump on that as soon as you've got some time. Okay. This is the last problem I did at the end of Wednesday's lecture. It's a very, very important problem, so I don't actually want to move away from it until I feel that I'm getting some signs from you guys that we're getting some understanding of this because it turns out it forms the basis of a lot of other solutions end up just becoming permutations, you know, at the core of it. So it's a pretty useful pattern to have [inaudible] list the permutations of a string. So you've got the string, you know, A B C D, and it's gonna print all the A C D A B C D A variations that you can get by rearranging all the letters. It's a deceptively short piece of code that does what's being done here – is that it breaks it down in terms of there being the permutation that's assembled so far, and then the remaining characters that haven't yet been looked at, and so at each recursive call it's gonna shuffle one character from the remainder onto the thing that's been chosen so far. So chose the next character in the permutation, and eventually, when the rest is empty, then all of the characters have been moved over, and have been laid down in permutation, then what I have is a full permutation of the length end. In the case where rest is not empty, I've still got some choices to make. The number of choices is exactly equal to the number of characters I have left in rest, and for each of the remaining characters at this point, maybe there's just C and D left, then I'm gonna try each of them as the next character in the permutation, and then permute what remains after having made that choice. And so this is [inaudible] operations here of attaching that [inaudible] into the existing so far, and then subtracting it out of the rest to set up the arguments for that recursive call. A little bit at the end we talked about this idea of a wrapper function where the interface to list permutations from a client point of view probably just wants to be, here's a string to permute the fact that [inaudible] keep this housekeeping of what I've built so far is really any internal management issue, and isn't what I want to expose as part of the interface, so we in turn have just a really simple one line translation that the outer call just sets up and makes the call to the real recursive function setting up the right state for the housekeeping in this case, which is the permutation we've assembled at this point. So I'd like to ask you a few questions about this to see if we're kind of seeing the same things. Can someone tell me what is the first permutation that's printed by this if I put in the string A B C D? Anyone want to help me?

Student:

A B C D.

Instructor (Julie Zelenski):A B C D. So the exact string I gave it, right, is the one that it picks, right, the very first time, and that's because, if you look at the way the for loop is structured, the idea is that it's gonna make a choice and it's gonna move through the recursion. Well, the choice it always makes is to take the next character of rest and attach it to the permutation as its first time through the for loop. So the first time through the very first for loop it's got A B C D to chose, it chooses A, and now it recurs on A in the permutation and B C D to go.

Well, the next call does exactly the same thing, chooses the first of what remains, which is B, attaches it to the permutation and keeps going. So the kind of deep arm of the recursion making that first choice and working its way down to the base case, right, right remember the recursion always goes deep. It goes down to the end of the – and bottoms on the recursion before it comes back and revisits any previous decision, will go A B C D, and then eventually come back and start trying other things, but that very first one, right, is all just the sequence there.

The next one that's after it, right, is A B D C, which involved unmaking those last couple decisions. If I look at the tree that I have here that may help to sort of identify it. That permute of emptying A B C, the first choice it makes is go with A, and then the first choice it makes here is go with B, and so on, so that the first thing that we can see printed our here will be A B C D, we get to the base case.

Once it's tried this, it'll come back to this one, and on this one it'll say, well, try the other characters. Well, there's no other characters. In fact, this one also finishes. It'll get back to here as it unwinds the stack the way the stack gets back to where it was previously in these recursive calls. And now it says, okay, and now try the ones that have, on the second iteration of that for loop, D in the front, and so it ends up putting A B D together, leaving C, and then getting A B D C.

And then, so on kind of over here, the other variations, it will print all of the ones that have A in the front. There are eight of them. I believe three, two – no, six. Six of them that have A in the front, and we'll do all of those. And only after having done all of working out that whole part, which involves kind of the [inaudible] of this whole arm, will eventually unwind back to the initial permute call, and then say, okay, now try again. Go with B in the front and work your way down from reading the A C D behind it. So we'll see all the As then the bunch that lead with B, then the bunch that leads with C, and then finally the bunch that leads with D.

It doesn't turn out – matter, actually, what order it visits them in because, in fact, all we cared about was seeing all of them. But part of what I'm trying to get with you is the sense of being able to look at that recursive code, and use your mind to kind of trace its activity and think about the progress that is made through those recursive calls. Just let me look at that code just for a second more and see if there's any else that I want to highlight for you.

I think that's about what it's gonna do. So if you don't understand this cost, if there's something about it that confuses you, now would be an excellent time for you to ask a question that I could help kind of get your understanding made more clear. So when you look at this code, you feel like you believe it works? You understand it?

Student:[Inaudible]. I have a question. Is there a simple change you can make to this code so that it does combinations [inaudible]?

Instructor (Julie Zelenski):Does combinations – you mean, like, will skip letters?

Student: Right.

Instructor (Julie Zelenski):Yes. It turns out we're gonna make that change in not five minutes. In effect, what you would do – and there's a pretty simple change with this form. I'm gonna show you a slightly different way of doing it, but one way of doing it would be to say, well, give me the letter, don't attach it to next right? So right now, the choices are pick one of the letters that remain and then attach it to the permutation. Another option you could do was pick a letter and just discard it, right? And so, in which case, I wouldn't add it into so far. I would still subtract it from here, and I'd make another recursive call saying, now how about the permutations that don't involve A at all? And I would just drop it from the thing and I would recurs on just the B C D part. So a pretty simple change right here. I'm gonna show you there's a slightly different way to formulate the same thing in a little bit, but – anybody want to own up to having something that confuses them?

Student:

[Inaudible] ask how you, like, what you would set up to test it?

Instructor (Julie Zelenski):So [inaudible] one of the, you know, the easiest thing to do here, and I have the code kind of actually sitting over here just in case, right, hoping you would ask because right now I just have the most barebones sort of testing. It's like, yeah, what if I just, you know, throw some strings at it and see what happens, right? And so the easiest strings to throw at it would be things like, well what happens if I give it the empty string, right? You know, so it takes some really simple cases to start because you want to see, well what happens when, you know, you give it an empty input. Is it gonna blow up?

And it's, like, the empty input, right, immediately realizes there's no choices and it says, well, look, there's nothing to print other than that empty string. What if I give it a single character string? Right? I get that string back. I'm like, okay, gives me a little bit of inspiration that it made one recursive call, right, and then hit the base case on the subsequent one. Now I start doing better ones.

A and B, and I'm seeing A B B A. Okay, so I feel like, you know, I got this, so if I put a C in the front, and I put the B A behind, then hopefully what I believe is that it's gonna try, you know, C in the front, and then it's gonna permute A and B behind it, and then similarly on down. So some simple cases that I can see verifying that it does produce, kind of, the input string as itself, and then it does the back end rearrangement, leaving this in the front, and then it makes the next choice for what can go in the front, and then the back end rearrangement. And kind of seeing the pattern that matches my belief about how the code works, right, helps me to feel good. What happens if I put in something that has double letters in it? So I put A P P L E in there. And all the way back at the beginning, right, I can plot more of them, right, that grows quite quickly right as we add more and more letters, right? Seeing the apple, appel, and stuff like that. There's a point where it starts picking the P to go in the front, and the code as it's written, right, doesn't actually make it a combination for this P and this P really being different. So it goes through a whole sequence of pull the second character to the front, and then permute the remaining four. And then it says, and now pull the third character to the front and permute the remaining four, which turns out to be exactly the same thing. So there should be this whole sequence in the middle, right, of the same Ps repeated twice because we haven't gone over a way to do anything about that, right? So – but it is reassuring to know that it did somehow didn't get, you know, confused by the idea of there being two double letters. [Inaudible] if I do this – if I just say A B A, right? Something a little bit smaller to look at, right? A in the front. A in the front goes permuted, B in the front, and then it ends up permuting these two ways that ends up being exactly the same, and then I get a duplicate of those in the front. So right now, it doesn't know that there's anything to worry about in terms of duplicates? What's a really easy way to get rid of duplicates? Don't think recursively, think – use a set. Right? I could just stuff them in a set, right? And then if it comes across the same one again it's like, yeah, whatever. So it would still do the extra work of finding those down, but would actually, like, I could print the set at the end having coalesced any of the duplicates. To actually change it in the code, it's actually not that hard either. The idea here is that for all of the characters that remain, right, and sometimes what I want to choose from is of the remaining characters uniqued, right? Pick one to move to the front. So if there's three more Ps that remain in rest, I don't need to try each of those Ps individually in the front, right? They'll produce the same result – pulling one and leaving the other two behind is completely irrelevant. So an easy way to actually stop it from even generating them, is when looking at the character here is to make sure that it is – that it doesn't duplicate anything else in the rest. And so probably the easiest way to do that is to either pick the first of the Ps or the last of the Ps, right, and to recur on and ignore the other ones. So, for example, if right here I did a find on the rest after I – do you see any more of these? And if you do, then skip it and just let those go. Or, you know, do the first. Don't do the remaining ones. You can either look to the left of look to the right and see if you see any more, and if so, skip the, you know, early or later ones depending on what your strategy is. Any questions about permute here?

Student:

[Inaudible].

Instructor (Julie Zelenski):So the [inaudible] can be just – look down here, right, it just looks like that, right? This is a very common thing. You'll see this again, and again, and it tends to be that the outer call, right, it tends to have this sort of this, you know, solve this problem, and then it turns out during the recursive calls you need to maintain some state about where you are in the problem. You know, like, how many letters you moved, or what permutation you've built so far, or where you are in the maze, or whatever it is you're trying to solve.

And so typically that wrapper call is just gonna be setting up the initial call in a way that has the initial form of that state present that the client to the function didn't need to know about. It's just our own housekeeping that we're setting up. Seems almost kind of silly to write down the function that has just line that just turns it into, basically, it's just exchanging – setting up the other parameters.

I am going to show you the other kind of master patter, and then we're gonna go on to kind of use them to solve other problems. This is the one that was already alluded to, this idea of a combinations. Instead of actually producing all of the four character strings that involve rearrangements of A B C D, what if I were to [inaudible] in kind of the subgroups, or the way I could chose certain characters out of that initial input, and potentially exclude some, potentially include some?

So if I have A B C, then the subsets of the subgroups of that would be the single characters A B and C. The empty string A B and C itself the full one, and then the combinations of two, A B, A C, and B C. Now, in this case we're gonna say that order doesn't matter. We're not – whereas permutations was all about order, I'm gonna use – I'm gonna structure this one where I don't care. If it's A B or B A I'm gonna consider it the same subset. So I'm just interested in inclusion. Is A in or out? Is B in or out? Is C in or out?

And so the recursive strategy we're gonna take is exactly what I have just kind of alluded to in my English description there, is that I've got an input, A B C. Each of those elements has either the opportunity of being in the subset or not. And I need to make that decision for everyone – every single element, and then I need to kind of explore all the possible combinations of that, right? When A is in, what if B is out? When A is in, what if B is in? So the recursion that I'm gonna use here is that at each step of the way, I'm gonna separate one element from the input, and probably the easiest way to do that is just to kind of take the front most element off the input and sort of separate it into this character by itself and then the remainder.

Similarly, the way I did with permute, and then given that element I have earmarked here, I can try putting it in the current subset or not. I need to try both, so I'm gonna make two recursive calls here, one recursive call where I've added it in, one recursive call where I haven't added it in. In both cases, right, I will have removed it from the rest, so what's being chosen from to complete that subset always is a little bit smaller, a little bit easier. So this is very, very much like the problem I described as the chose problem. Trying to choose four people from a dorm to go to flicks was picking Bob and then – [inaudible] to Bob and not Bob, and then saying, well, Bob could go, in which I need to pick three people to go with him, or Bob could not go and I need to pick four people to go without Bob.

This is very much that same pattern. Two recursive calls. Identify one in or out. The base case becomes when there's nothing left to check out. So I start with the input A B C – A B C D let's say. I look at that first element – I just need to pick one. Might as well pick the front one, it's the easy one to get to. I add it to the subset, remove it from the input, and make a recursive call. When that recursion completely terminates, I get back to this call, and I do the same thing again.

Remaining input is B C D again but now the subset that I've been building doesn't include. So inclusion exclusion are the two things that I'm trying. So the subset problem, right, very similar in structure to the way that I set up permutations, right, as I'm gonna keep track of two strings as I'm working my way down the remainder in the rest, right, things that I haven't yet explored so far is what characters I've chosen to place into the subset that I'm building. If I get to the end where there's nothing left in the rest, so there's no more choices to make, then what I have in the subset is what I have in the subset, and I go ahead and print it.

In the case that there's still something to look at I make these two calls, one where I've appended it in, where I haven't, and then both cases, right, where I have subtracted it off of the rest by using the subster to truncate that front character off. So the way that permute was making calls, right, was in a loop, and so sometimes it's a little bit misleading. You look at it and you think there's only one recursive call, but in fact it's in inside a loop, and so it's making, potentially, end recursive calls where end is the length of the input. It gets a little bit shorter each time through but there's always, you know, however many characters are in rest is how many recursive calls it makes. The subsets code actually makes exactly two recursive calls at any given stage, in or out, and then recurs on that and what is one remaining. It also needs a wrapper for the same exact reason that permutations did. It's just that we are trying to list the subsets of a particular string, in fact, there's some housekeeping that's going along with it. We're just trying the subset so far is something we don't necessarily want to put into the public interface in the way that somebody would have to know what to pass to that to get the right thing. Anybody have a question about this? So given the way this code is written, what is the first subset that is printed if I give it the input A B C D? A B C D. Just like it was with permute, right? Everything went in. What is the second one printed after that? A B C, right? So I'm gonna look at my little diagram with you to help kind of trace this process of the recursive call, right, is that the leftmost arm is the inclusion arm, the right arm is the exclusion arm. At every level of the tree, right, we're looking at the next character of the rest and deciding whether to go in or out, the first call it makes is always in, so at the beginning it says, I'm choosing about A. Is A in? Sure. And then it gets back to the level of recursion. It says, okay, look at the first thing of rest, that's B. Is B in? Sure. Goes down the right arm, right? Is C in? Sure. Is D in? Sure. It gets to here and now we have nothing left, so we have [inaudible] A B C D. With the rest being empty, we go ahead and print it [inaudible] there's no more choices to make. We've looked at every letter and decided whether it was in or out. It will come back to here, and I'll say, okay, I've finished all the things I can make with D in, how about we try it with D out. And so then D out gives us just the A B C. After it's done everything it can do with A B C in, it comes back to here and says, okay, well now do this arm, and this will drop C off, and then try D in, D out. And so it will go from the biggest sets to the smaller sets, not quite monotonically though. The very last set printed will be the empty set, and that will be the one where it was excluded all the way down, which after it kind of tried all these combinations of in out, in out, in out, it eventually got to the out, out, out, out, out, out, out case which will give me the empty set on that arm. Again, if I reverse the calls, right, I'd still see all the same subsets in the end. They just come out in a different order. But it is worthwhile to model the recursion in your own head to have this idea of understanding about how it goes deep, right? That it always makes that recursive call and has to work its way all the way to the base case and terminate that recursion before it will unfold and revisit the second call that was made, and then fully explore where it goes before it completes that whole sequence. Anybody want to ask me a question about these guys? These are just really, really important pieces of code, and so I'm trying to make sure that I don't move past something that still feels a little bit mystical or confusing to you because everything I want to do for the rest of today builds on this. So now if there's something about either of these pieces of code that would help you – yeah.

Student:

What would be the simplest way to do that?

Instructor (Julie Zelenski):So probably the simplest way, like, if you said, oh, I just really want it to be in alphabetical order, would be to put them in a set, right, and then have the set be the order for you.

Student: So you said order doesn't matter?

Instructor (Julie Zelenski):Oh, you did care about how they got ordered.

Student: So, like, let's say you didn't want B C D to equal C D B.

Instructor (Julie Zelenski):So in that case, right, you would be more likely to kind of add a permutation step into it, right? So if B C D was not the same thing as A B C, right, it would be, like, well I'm gonna chose – so in this case, right, it always – well the subsets will always be printed as kind of a subsequence. So – and let's say if the input was the alphabet, just easy way to describe it, all of the subsets I'm choosing will always be in alphabetical order because I'm always choosing A in or not, B in or not.

If I really wanted B Z to be distinct from Z B, then really what I want to be doing at each step is picking the next character to go, and not always assuming the next one had to be the next one in sequence, so I would do more like a permute kind of loop that's like pick the next one that goes, remove it from what remains and recur, and that I need that separate step we talked about of – and in addition to kind of picking, we also have to leave open the opportunity that we didn't pick anything and we just kind of left the subject as is, right, so we could [inaudible] or not.

And so permute always assumes we have to have picked everything. The subset code would also allow for some of them just being entirely skipped. So we pick the next one, right, and then eventually stopped picking.

Student:[Inaudible] just in your wrapper function then [inaudible] put a for loop or something like that, right? When you're through changing your string?

Instructor (Julie Zelenski):Yeah, you can certainly do that, like in a permute from the outside too, right. So there's often very, you know, multiple different ways you can get it the same thing whether you want to put it in the recursion or outside the recursion, have the set help you do it or not, that can get you the same result.

So let me try to identify what's the same about these to try to kind of back away from it and kind of move just to see how these are more similar than they are different, right, even though the code ends up kind of being – having some details in it that – if you kind of look at it from far away these are both problems that are about choice. That the recursive calls that we see have kind of a branching structure and a depth factor that actually relates to the problem if you think about it in terms of decisions, that in making a permutation your decision is what character goes next, right? In the subset it's like, well, whether this character goes in or not that the recursive tree that I was drawing that it shows all those calls, the depths of it represents the number of choices you make.

Each recursive call makes a choice, right? A yes, no, or a next letter to go, and then recurs from there. So I make, you know, one of those calls, and then there's N minus 1 beneath it that represent the further decisions I make that builds on that. And so, in the case of permutation, once I've picked the, you know, N minus 1 things to go, there's only one thing left, right? And so in some sense the tree is N because I have to pick N things that go in the permutation and then I'm done.

In the subsets, it's also N, and that's because for each of the characters I'm deciding whether it's in or out. So I looked at A and decided in or out. I looked at B and decided in or out. And I did that all the way down. That was the life of the input. The branching represents how many recursive calls were made at each level of the recursion. In the case of subsets, it's always exactly two. In, out, all the way down, restarts at 1, branches to 2, goes to 4, goes to 8, 16, and so on.

In the permute case, right, there are N calls at the beginning. I have N different letters to choose from, so it's a very wide spread there, and that at that next level, it's N minus 1. Still very wide. And N minus 2. And so the overall tree has kind of an N times N minus 1 times N minus 2 all the way down to the bottom, which the factorial function – which grows very, very quickly.

Even for small inputs, right, the number of permutations is enormous. The number of subsets is to the end in or out, right, all the way across. Also, a very, you know, resource intensive problem to solve, not nearly as bad as permutation, but both of them, even for small sizes of N, start to become pretty quickly intractable. This is not the fault of recursion, right, these problems are not hard to solve because we're solving them in the wrong way.

It's because there are N factorial permutations. There are [inaudible] different subsets, right? Anything that's going to print to the N things, or N factorial things is going to do N factorial work to do so. You can't avoid it. There's a big amount of work to be done in there. But what we're trying to look at here is this idea that those trees, right, represent decisions. There's some decisions that are made, you know, a decision is made at each level of recursion, which then is kind of a little bit closer to having no more decisions to make. You have so many decisions to make, which is the depth of the recursion. Once you've made all those decisions, you hit your base case and you're done.

The tree being very wide and very deep makes for expensive exploration. What we're gonna look at is a way that we can take the same structure of the problem, one that fundamentally could be exhaustive, exhaustive meaning tried every possible combination, every possible rearrangement and option, and only explore some part of the tree. So only look around in some region, and as soon as we find something that's good enough.

So in the case, for example, of a search problem, it might be that we're searching this space that could potentially cause us to look at every option, but we're also willing to say if we make some decisions that turn out good enough, that get us to our goal, or whatever it is we're looking for, we won't have to keep working any farther. So I'm gonna show you how we can take an exhaustive recursion problem and turn it into what's called a recursive backtracker.

So there's a lot of text on this slide but let me just tell you in English what we're trying to get at. That the idea behind permutations or subsets is that at every level there's all these choices and we're gonna try every single one of them. Now imagine we were gonna make a little bit less a guarantee about that. Let's design the function to return some sort of state that's like I succeeded or I failed. Did I find what I was looking for? At each call, I still have the possibility of multiple calls of in out, or a choice from what's there. I'm gonna go ahead and make the choice, make the recursive call, and then catch the result from that recursive call, and see whether it succeeded.

Was that a good choice? Did that choice get me to where I wanted to be? If it did, then I'm done. I won't try anything else. So I'll stop early, quite going around the for loop, quit making other recursive calls, and just immediately [inaudible] say I'm done. If it didn't – it came back with a failure, some sort of code that said it didn't get where I want to do, then I'll try a different choice. And, again, I'll be optimistic. It's a very optimistic way of doing stuff. It says make a choice, assume it's a good one, and go with it. Only when you learn it didn't work out do you revisit that decision and back up and try again. So let me show you the code. I think it's gonna make more sense, actually, if I do it in terms of looking at what the code looks like. This is the pseudo code at which all recursive backtrackers at their heart come down to this pattern.

So what I – I tried to be abstract [inaudible] so what does it mean to solve a problem, and what's the configuration? That depends on the domain and the problem we're trying to solve. But the structure of them all looks the same in that sense that if there are choices to be made – so the idea is that we cast the problem as a decision problem. There are a bunch of choices to be made. Each [inaudible] will make one choice and then attempt to recursively solve it.

So there's some available choices, in or out, or one of next, or where to place a tile on a board, or whether to take a left or right turn and go straight in a maze. Any of these things could be the choices here. We make a choice, we feel good about it, we commit to it, and we say, well, if we can solve from here – so we kind of update our statement so we've made that term, or, you know, chosen that letter, whatever it is we're doing.

If that recursive call returned true then we return true, so we don't do any unwinding. We don't try all the other choices. We stop that for loop early. We say that worked. That was good enough. If the solve came back with a negative result, that causes us to unmake that choice, and then we come back around here and we try another one. Again, we're optimistic. Okay, left didn't work, go straight. If straight doesn't work, okay, go right. If right didn't work and we don't have any more choices, then we return false, and this false is the one that then causes some earlier decision to get unmade which allows us to revisit some earlier optimistic choice, undo it, and try again.

The base case is hit when we run out of choices where we've – whatever configuration we're at is, like, okay, we're at a dead end, or we're at the goal, or we've run out of letters to try. Whatever it is, right, that tells us, okay, we didn't – there's nothing more to decide. Is this where we wanted to be? Did it solve the problem? And I'm being kind of very deliberate today about what does it mean [inaudible] update the configuration, or what does it mean for it to be the goal state because for different problems it definitely means different things. But the code for them all looks the same kind of in its skeletal form.

So let me take a piece of code and turn it into a recursive backtracker with you. So I've got recursive permute up here. So as it is, recursive permute tries every possible permutation, prints them all. What I'm interested in is is this sequence a letters an anagram. Meaning, it is – can be rearranged into something that's an English word.

Okay. So what I'm gonna do is I'm gonna go in and edit it. The first thing I'm gonna do is I'm gonna change it to where it's gonna return some information. That information's gonna be yes it works, no it didn't. Okay? Now I'm gonna do this. I'm gonna add a parameter to this because I – in order to tell that it's a word I have to have someplace to look it up. I'm gonna use the lexicon that actually we're using on this assignment.

And so when I get to the bottom and I have no more choices, I've got some permutation I've assembled here in – so far. And I'm going to check and see if it's in the dictionary. If the dictionary says that that's a word then I'm gonna say this was good. That permutation was good. You don't need to look at any more permutations. Otherwise I'll return false which will say, well, this thing isn't a word. Why don't you try again?

I'm gonna change the name of the function while I'm at it to be a little more descriptive, and we'll call it is anagram. And then when I make the call here [inaudible] third argument. I'm not just gonna make the call and let it go away. I really want to know did that work, so I'm gonna say, well, if it's an anagram then return true. So if given the choice I made – I've got these letters X J Q P A B C, right? It'll pick a letter and it'll say, well if, you know, put that letter in the front and then go for it.

If you can make a word out of having made that choice, out of what remains, then you're done. You don't need to try anything else. Otherwise we'll come back around and make some of the further calls in that loop to see if it worked out. At the bottom, I also need another failure case here, and that comes when the earlier choices, right – so I got, let's say somebody has given me X J, and then it says, given X J, can you permute A and B to make a word?

Well, it turns out that you can – you know this ahead of time. It doesn’t have the same vision you do. But it says X J? A B? There’s just nothing you can do but it’ll try them all dutifully. Tried A in the front and then B behind, and then tried B in the front and A behind it, and after it says that, it says, you know what, okay, that just isn’t working, right? It must be some earlier decision that was really at fault. This returned false is going to cause the, you know, sort of stacked up previous anagram calls to say, oh yeah, that choice of X for the first letter was really questionable. So imagine I’ve had, like, E X, you know, T I. I’m trying to spell the word exit – is a possibility of it. That once I have that X in the front it turns out nothing is going to work out, but it’s gonna go through and try X E I T, X E T I, and so on. Well, after all those things have been tried it says, you know what, X in the front wasn’t it, right? Let’s try again with I in the front. Well after it does that it won’t get anywhere either. Eventually it’ll try E in the front and then it won’t have to try anything else after that because that will eventually work out. So if I put this guy in like that, and I build myself a lexicon, and then I change this to anagram word. I can’t spell. I’d better pass my lexicon because I’m gonna need that to do my word lookups. [Inaudible]. And down here. Whoops. Okay. I think that looks like it’s okay. Well, no – finish this thing off here. And so if I type in, you know, a word that I know is a word to begin with, like boat, I happen to know the way the permutations work [inaudible] try that right away and find that. What if I get it toab, you know, which is a rearrangement of that, it eventually did find them. What if I give it something like this, which there’s just no way you can get that in there. So it seems to [inaudible] it’s not telling us where the word is. I can actually go and change it. Maybe that’d be a nice thing to say. Why don’t I print the word when it finds it? If lex dot contains – words so far – then print it. That way I can find out what word it thinks it made out of it. So if I type toab – now look at that, bota. Who would know? That dictionary’s full of some really ridiculous words. Now I’ll get back with exit. Let’s type some other words. Query. So it’s finding some words, and then if I give it some word that I know is just nonsense it won’t print anything [inaudible] false. And so in this case, what it’s doing is its come to a partial exploration, right, of the permutation tree based on this notion of being able to stop early on success. So in the case of this one, right, even six nonsense characters, it really did do the full permutation, in this case, the six factorial permutations, and discover that none of them worked. But in the case of exit or the boat that, you know, early in the process it may have kind of made a decision, okay so [inaudible] in this case it will try all the permutations with Q in the front, right? Which means, okay, we’ll go with it, and then it’ll do them in order to start with, but it’ll start kind of rearranging and jumbling them up, and eventually, right, it will find something that did work with putting in the front, and it will never unmake that decision about Q. It will just sort of have – made that decision early, committed to it, and worked out, and then it covers a whole space of the tree that never got explored of R in front, and Y in the front, and E in the front because it had a notion of what is a satisfactory outcome. So the base case that it finally got to in this case met the standard for being a word was all it wanted to find, so it just had to work through the base cases until it found one, and potentially that could be very few of them relative the huge space. All right. I have this code actually on this slide. So it’s permute, and that is turning into is anagram. And so, in blue, trying to highlight the things that changed in the process, right, that the structure of kind of how it’s exploring, and making the recursive calls is exactly the same. But what we’re using now is some return information from the function to tell us how the progress went, and then having our base case return some sense of did we get where we wanted to be, and then when we make the recursive call, if it did succeed, right, we immediately return true and unwind out of the recursion, doing no further exploration, and in the case where all of our choices gave us no success, right, we will return the call that says, well that was unworkable how we got to where we were. So this is the transformation that you want to feel like you could actually sort of apply again and again, taking something that was exhaustive, and looked at a whole space, and then had – change it into a form where it's like, okay, well I wanted to stop early when I get to something that's good enough. A lot of problems, right, that are recursive backtrackers just end up being procedural code that got turned into this based on a goal that you wanted to get to being one of the possibilities of the exploration. Anybody have any questions of what we got there? Okay. I'm gonna show you some more just because they are – there are a million problems in this space, and the more of them you see, I think, the more the patterns will start to emerge. Each of these, right, we're gonna think of as decision problems, right, that we have some number of decisions to make, and we're gonna try to make a decision in each recursive call knowing that that gives us fewer decisions that we have to make in the smaller form of the sub problem that we've built that way, and then the decisions that we have open to us, the options there represent the different recursive calls we can make. Maybe it's a for loop, or maybe a list of the explicit alternatives that we have that will be open to us in any particular call. This is a CS kind of classic problem. It's one that, you know, it doesn't seem like it has a lot of utility but it's still interesting to think about, which is if you have an eight by eight chessboard, which is the standard chessboard size, and you had eight queen pieces, could you place those eight queens on the board in such a way that no queen is threatened by any other? The queen is the most powerful player on the board, right, can move any number of spaces horizontally, vertically, or diagonally on any straight line basically, and so it does seem like, you know, that there's a lot of contention on the board to get them all in there in such a way that they can't go after each other. And so if we think of this as a decision problem, each call's gonna make one decision and recur on the rest. The decisions we have to make are we need to place, you know, eight queens let's say, if the board is an eight by eight board. So at each call we could place one queen, leaving us with M minus 1 to go. The choices for that queen might be that one way to kind of keep our problem – just to manage the logistics of it is to say, well, we know that there's going to be a queen in each column, right, it certainly can't be that there's two in one column. So we can just do the problem column by column and say, well, the first thing we'll do is place a queen in the leftmost column. The next call will make a queen – place a queen in the column to the right of that, and then so on. So we'll work our way across the board from left to right, and then the choices for that queen will be any of the [inaudible] and some of those actually are – we may be able to easily eliminate as possibilities. So, for example, once this queen is down here in the bottommost row, and we move on to this next column, there's no reason to even try placing the queen right next to it because we can see that that immediately threatens. So what we'll try is, is there a spot in this column that works given the previous decisions I've made, and if so, make that decision and move on. And only if we learned that that decision, right, that we just made optimistically isn't successful will we back up and try again. So let me do a little demo with you. Kind of shows this doing its job. Okay. So [inaudible] I'm gonna do it as I said, kind of column by column. [Inaudible] is that I'm placing the queen in the leftmost column to begin, and the question mark here says this is a spot under consideration. I look at the configuration I'm in, and I say, is this a plausible place to put the queen? And there's no reason not to, so I go ahead and let the queen sit there.

Okay, so now I'm going to make my second recursive call. I say I've placed one queen, now there's three more queens to go. Why don't we go ahead and place the queens that remain to the right of this. And so the next recursive call comes in and says, if you can place the queens given the decision I've already made, then tell me yes, and then I'll know all is good.

So it looks at the bottommost row, and it says, oh, no, that won't work, right? There's a queen right there. It looks at the next one and then sees the diagonal attack. Okay. Moves up to here and says, okay, that's good. That'll work, right? Looks at all of them and it's okay. So now it says, okay, well I've made two decision. There's just two to go. I'm feeling good about it. Go ahead and make the recursive call.

The third call comes in, looks at that row, not good, looks at that row, not good, looks at that row, not good, looks at that row, not good. Now this one – there weren't any options at all that were viable. We got here, and given the earlier decisions, and so the idea is that, given our optimism, right, we sort of made the calls and just sort of moved on. And now we've got in the situation where we have tried all the possibilities in this third recursive call and there was no way to make progress. It's gonna hit the return false at the bottom of the backtracking that says, you know what, there was some earlier problem.

There was no way I could have solved it given the two choices – or, you know, whatever – we don't even know what choices were made, but there were some previous choices made, and given the state that was passed to this call, there's no way to solve it from here. And so this is gonna trigger the backtracking. So that backtracking is coming back to an earlier decision that you made and unmaking it. It's a return false coming out of that third call that then causes the second call to try again.

And it goes up and it says okay, well where did I leave off? I tried the first couple of ones. Okay, let's try moving it up a notch and see how that goes. Then, again, optimistic, makes the call and goes for it. Can't do this one. That looks good. And now we're on our way to placing the last queen, feeling really comfortable and confidant, but discovering quickly, right, that there was no possible.

So it turns out this configuration with these three queens, not solvable. Something must be wrong. Back up to the most immediate decision. She knows it doesn't unmake, you know, earlier decisions until it really has been proven that that can't work, so at this point it says, okay, well let's try finding something else in this column. No go. That says, okay, well that one failed so it must be that I made the wrong decision in the second column.

Well, it turns out the second column – that was the last choice it had. So in fact it really was my first decision that got us off to the wrong foot. And now, having tried everything that there was possible, given the queen in the lower left in realizing none of them worked out, it comes back and says, okay, let’s try again, and at this point it actually will go fairly quickly. Making that initial first decision was the magic that got it solved.

And then we have a complete solution to the queens. We put it onto eight, and let it go. You can see it kind of checking the board, backing up, and you notice that it made that lower left decision kind of in – it’s sticking to it, and so the idea is that it always backs up to the most recent decision point when it fails, and only after kind of that one has kind of tried all its options will it actually back up and consider a previous decision as being unworthy and revisiting it.

In this case that first decision did work out, the queen being in the lower left. It turns out there were – you know, you saw the second one had to kind of slowly get inched up in the second row. Right? It wasn’t gonna work with the third row. It tried that for a while. Tried the fourth row for a while. All the possibilities after that, but eventually it was that fifth row that then kind of gave it the breathing room to get those other queens out there.

But it did not end up trying, for example, all the other positions for the queen in the first row, so it actually – it really looked at a much more constrained part of the entire search tree than an exhaustive recursion of the whole thing would have done. The code for that – whoops – looks something like this. And one of the things that I’ll strongly encourage you to do when you’re writing a recursive backtracking routine, something I learned from Stuart Regis, who long ago was my mentor, was the idea that when – trying to make this code look as simple as possible, that one of the things you want to do is try to move away the details of the problem.

For example, like, is safe – given the current placement of queens, and the row you’re trying, and the column you’re at, trying to decide that there’s not some existing conflict on the board with the queen already being threatened by an existing queen just involves us kind of traipsing across the grid and looking at different things. But putting in its own helper function makes this code much easier to read, right?

Similarly, placing the queen in the board, removing the queen from the board, there are things they need to do. Go up to state and draw some things on the graphical display. Put all that code somewhere else so you don’t have to look at it, and then this algorithm can be very easy to read. It’s like for all of the row. So given the column we’re trying to place a queen in, we’ve got this grid of boolean that shows where the queens are so far, that for all of the rows across the board, if, right, it’s safe to place a queen in that row and this column, then place the queen and see if you can solve starting from the column to the right, given this new update to the board.

If it worked out, great, nothing more we need to do. Otherwise we need to take back that queen, unmake that decision, and try again. Try a higher row. Try a higher row, right. Again, assume it’s gonna work out. If it does, great. If it doesn’t, unmake it, try it again. If we tried all the rows that were open to us, and we never got to this case where this returned true, then we return false, which causes some previous one – we’re backing up to a column behind it. So if we were currently trying to put a queen in column two, and we end up returning false, it’s gonna cause column one to unmake a decision and move the queen up a little bit higher.

If all of the options for column one fail, it’ll back up to column zero. The base case here at the end, is if we ever get to where the column is past the number of columns on the board, then that means we placed a queen all the way across the board and we’re in success land. So all this code kind of looks the same kind of standing back from it, right, it’s like, for each choice, if you can make that choice, make it. If you solved it from here, great, otherwise, unmake that choice.

Here’s my return false when I ran out of options. There’s my base case – it says if I have gotten to where there’s no more decisions to make, I’ve placed all the queens, I’ve chosen all the letters, whatever, did I – am I where I wanted to be? There’s no some sort of true or false analysis that comes out there about being in the right state. How do you feel about that? You guys look tired today, and I didn’t even give you an assignment due today, so this can’t be my fault, right?

I got a couple more examples, and I’m probably actually just gonna go ahead and try to spend some time on them on Monday because I really don’t want to – I want to give you a little bit more practice though. So we’ll see. We’ll see. I’ll do at least one or two more of them on Monday before we start talking about pointers and linked lists, and so I will see you then. But having a good weekend. Come and hang out in Turbine with me.

[End of Audio]

Duration: 50 minutes