SRINI DEVADAS: Good morning, everyone. Thanks for coming in on a Friday morning, especially with the long weekend coming up. We're going to pick up where we left off yesterday. And so if you recall, we were looking at an iterative approach to solving the eight queens problem. And in case you've forgotten in the, I guess, intervening 20 hours, the eight queens problem, which is a specific instance of n queens is you have a chess board-- and I won't draw all the squares-- which is, obviously, an 8 by 8 board. And you want to place eight queens on this board such that no two queens attack each other.

And we looked at different data structures to solve this. The naive data structure would be to represent the board as a two-dimensional list. And so you end up having an 8 by 8, if you have a chessboard for the eight queens problem. We looked at a 4 by 4 for starters.

And then after that, we looked at compressing that representation down to a one-dimensional list where let's say you had a 5 by 5 board because we've got five entries here. You exploit the fact that there can be at most one queen on any given column. And you say I'm going to represent the position of that queen using a single number where is from, in this case 0 through 4, if I have a 5 by 5 board, and you put that in here.

So a minus 1 would represent this being empty. A 2 would represent a queen in the second column in this position over here, et cetera, et cetera. So with this more compact representation, it turns out that the conflict check becomes pretty straightforward.

And if you remember, our conflict check is incremental in the sense that what you're going to do is assume that queens have already been placed in some number of columns. I mean, it could be zero. In this case it's 2. And you're going to try and check to see when a third queen is placed on this third column, number two, whether it conflicts with the existing queens.

But you don't have to worry about the checks associated with the existing queens because you've already taken care of that. So that's the incremental nature of this check.

And I won't go over this code again, but the check associated with ensuring that when you place a queen on a given column, it checks the diagonals. That's the most interesting part. And you put a queen here, you need to check this, and you need to check that. And in this case, obviously there's a conflict. And because there are two diagonals, that's why you have the ABS, the absolute value over there, and you're exploiting the fact that the diagonal is a diagonal of a square, and how much you move up should be how much you move to the right or down, left, or what have you.

Another check is simply ensuring that when you look at the particular column and you see the number current is a particular column, the number associated with that particular column shouldn't be repeated in some other column. Because that would imply that if you saw a 2 and a 2 here, obviously, that would imply that you have two queens on the same row, and you can't have that.

So that's the story. And we decided to use iteration because, at this point in time, you know nothing else in terms of control flow. I know that's not the case, but we assume that. And so you end up having this ugly code associated with eight nested loops that does this incremental enumeration. So board 0 being i implies that you're taking the first column and you're iterating the positions of the queen in the rows of the first column, and so on and so forth.

And if you go ahead and run this, which we did last time you end up getting 92 different solutions to the eight queens problem, and as I mentioned, there's only 12 distinct ones. So if you take rotation and reflection into account and each of these is a legitimate solution, that needs to be translated into the picture that you see here by essentially taking this data structure that you see. 7 becomes the queen in the left corner, et cetera.

So we absolutely wouldn't want to publish this code. I guess, in effect, I'm doing it. But you would not want to claim authorship of this code especially if you know about the programming technique that's called recursion.

So that's really what we're going to do today. We're going to look at a couple of different puzzles. We'll finish up with the n queen's puzzle fairly quickly, and we'll look at how we could use recursion, which is an algorithm paradigm as well as a programming paradigm to solve challenging problems.

And there's really two classes, at least, of recursive algorithms, and you're going to see both of those today. The first one is nQueens, and I'll get to the second one when we get to the second one.

So our goal now is to take that ugly code and make it pretty. And before I get into that, I want to show you a simpler example of iteration versus recursion. And this is about as simple as it gets while still being interesting from an algorithmic standpoint.

So the greatest common divisor, Euclid's age-old algorithm to compute the greatest common divisor. And you see the two line, three line, what have you iterative algorithm up there that corresponds to taking the pair of numbers, obviously, m and n that you need to compute the greatest common divisor for. And all you do is assign m to n and n becomes n mod m.

And I'm not going to explain why this algorithm is correct or anything like that. I'm not particularly interested in that. What I'm much more interested in is ensuring that you understand that the iterative algorithm is the same as the recursive one. The recursive one gets you the correct answer if you assume the iterative algorithm is correct and vice versa, and that's really what this is about. So we won't argue if Euclid is correct or not.

But the point here is you could have translated-- well, you did translate this iterative algorithm into a recursive one. And there's a couple of things I want you to keep in mind when you look at recursive code and certainly when you write recursive code. And it's much easier pointing this out in a simple example like this.

So first off, when we say recursion-- and I should have probably said this a little bit earlier-- recursion is something calling itself. So you can say a function, f, calling itself, that's the simple case. We also consider if you have f calling g, which calls f, then that is recursive as well. And see you can have arbitrary structures that correspond to nested recursion. We're really only going to be looking at functions at least in this lecture, where you have a single function, f, calling itself, which is exactly the case for our RGCD, the Recursive GCD that you see here.

You see a call RGCD inside of RGCD, and you know this function is recursive. Sometimes you may look at a function and you don't see that immediate syntactic evidence that the function is recursive, but it may be recursive because of nested recursion, f calling g, more complicated recursion. So this is clearly a recursive function, RGCD. And two things that you need to look for if you want to write correct recursion, I mean this is not sufficient to write correct recursion, but necessary.

And in particular, when I say correct recursion, I want it to be non-terminating, so the problem with something calling itself is, you think of it as there's some sort of cycle and you absolutely need to break out of the cycle, otherwise, you will be in the cycle forever. You could call something which calls something, which calls something that goes on, your program isn't going to terminate. So you always want to be careful when you write recursive code that there's a base case. There's a termination condition.

So there should be some part of the code where if certain conditions are satisfied, you're not making a recursive call. That is what I would consider the base case. That really comes from induction and from the mathematics of recursion. You want to have a base case and then the recursion is really the inductive step.

If that doesn't make too much sense, don't worry about it. You can think of it syntactically. Look for a path in the code where you don't see RGCD inside of RGCD. And you can clearly see that in the if m mod n equals equals 0, return n.

So you want to have what we call the base case where a function returns. That's not enough because you don't quite know if you're going to get to that base case or not. You're going to call this function with some arguments, and you want to get some sense that the arguments are shrinking as you go. And it could even be the other way around. The arguments may increase, and once you grow beyond a certain threshold, you fire off on the return without making the recursive call.

So this notion of the arguments becoming smaller is very common in the case of the algorithmic paradigm that we'll be following in this class where we're going to take large problems like a eight queens or n queens and go to n minus 1 queens and n minus 2 queens. And divide and conquer, which is other paradigm that we look at. As you can see from the name is taking a big problem and making it smaller.

So really for the most part, you're going to essentially have a situation where you're going to say that the arguments to the function should be smaller in some sense. And smaller could mean bigger. I mean, it's just a definitional thing. I mean, the point is when you get to the base case, usually you are saying, as you can see in this base case, you're saying when it comes down to the fact that a particular condition is satisfied, in this case, it's the relationship between m and n. And if that's 0, then we return n.

But in other cases, it just might be when you get to a one queen problem, you're done. I mean, you have no choice. You've got this 1 by 1 board and you've got to put your queen on it, and obviously, that might create a conflict, but you're done. I mean, there's no two things about it if you have one queen problem. There's only one step. You put your queen down.

So that's where we were at with respect to the puzzles that we're going to be doing in this class where you're going to shrink things down. So in that sense, smaller would mean smaller.

It's not necessarily in quotes. So that makes sense? Any questions at all about syntax or anything that you see on the board here? All right good.

So now, I'm ready to show you what the n queens code looks like. One of the other problems, of course, with the eight queens code is that it only works for eight queens, and it doesn't work for nine. And that's annoying, so you'd like to have a procedure that takes n as an argument and works for n equals 4, n equals 8. And eventually, you could run this on n equals 20, and it will finish. I'll show you.

But there is an exponential relationship between run time and n because it's just a complicated problem. There's a lot of combinations. So you do not want to run this code for more than n equals 25.

It hasn't completed. I think at some point I did an analysis on how long it would take for n equals 30, and it wasn't something I wanted to wait for. I mean, my computer would have gotten old and died before the program came back. Which is a reasonable experiment to carry out, but I like my computer. I'm kind of annoyed with it because of the new operating system, but I generally like it.

So you can see that we're going to use exactly the same framework. We're going to use the same no conflicts routine. There's no change there, no change in data structures. There's no change in really even the incremental strategy that we're following. But we can take that intuitive code and turn it into this pretty code that has about five lines, or seven lines, what have you.

And so you can see, again, that if you look at that code, I mean, it's not that much more complicated. There's more going on in there. There's more going on because it's a more interesting problem at some level than GCD, but it has the same characteristics in terms of having a base case, and it's actually an easy one to look at. And it also has a property 2 here where the argument to the nQueens is going to be one less than what the caller function has as its argument because you're going column by column, and you're taking away.

In this case, you're not necessarily going to a 7 by 7, really. It's not that you're going from 8 by 8 to 7 by 7. You're going from 8 by 8 to 7 by 8. You have one less column. You still have to work with all the rows.

So that's really what's going on out here. If current equal size-- so what's happening is current is actually incrementing. So you're moving and what is remaining is becoming smaller. And current starts with-- so if you look at nQueens here, it says nQueens, a size, and rQueens, which is this routine here, it sets current to be 0. And size is, of course, whatever you give nQueens, which could be 8 or 20.

And finally, if you look at this part of the code, you get to the point where you run out of columns when current equals size. And at this point, you put all the queens down, and that things haven't fallen apart-- you haven't returned false-- and therefore, you're done. So when you get to that quote ninth column, you're off the board, and you're done. And all of the existing queens, the eight queens that were put on the board, didn't conflict with each other. That's how it works.

And then over here, you've taken your eight loops that were in the eight queens problem or the for loops in the four queens problem and you've gone off and turned it into one loop. And the reason that works is because rQueens is calling rQueens out here. So that's your recursive call.

And then the argument here is current plus 1, and it's getting closer and closer to size as you go around, which means that it's effectively getting smaller. What you're working with is getting smaller. So if of look at what happens here, you can trace the execution of this, and this is worthwhile doing. And I'll put this in the notes, and you can look at it for a 4 by 4.

But in general, if you're confused about recursion, it's absolutely worthwhile to trace the execution for small arguments and figure out exactly when things are terminating and when things are being called recursively. And this is simply a single call. It looks like it's a single call, because there's only one instance of rQueens. But if you look at the code, how many calls would I possibly make if I were sitting with an eight queens problem and I started with size equals 8?

Then in that very first, I invoke rQueens with current equal 0 and size equals 8. How many rQueens calls will the first rQueens make? How many times would potentially make that? So if I look at the branching here and I say this is rQueens and this has current equals 0 and size equals 8, the way you want to think about this from a tracing standpoint is you're making calls. rQueens is calling rQueens.

And obviously, there is rQueens in the middle of that code over there. And so what is the breadth here in terms of the number of calls? Someone? Trace the code. So i in range 8, which means 0 through 7, and then I'm going to go off, and this is the very first one. So noConflicts is going to constantly return-- it's the first queen. So noConflicts is going to constantly return true.

So I'm going to go inside of the if, and I'm going to make that rQueens call. So I'm going to have eight different calls associated with each of the positions that corresponds to the queen on that first column. And then each of these could potentially make eight different calls, potentially I said because as you get deeper in the recursion, the noConflicts is going to fire false, and it may not make the call. But obviously, the very first one, you're going to make eight calls.

So recursion is scary from a performance standpoint to some people and rightly so because things could explode on you, and this goes back to what I said about running this with n r size equals 25 and so on and so forth. And one of the things that is worth doing and you have to do this type of analysis in classes like 6.006, not a lot of this but because we don't do exponential algorithms.

But you can look at this tree that is being generated by these recursive calls and you do want to get some sense of how large this tree is going to become. And we're going to look at a different class of algorithms. Most of algorithms we're going to look at that are recursive for the rest of this course including the next one that we'll get to in a few minutes, aren't as bad as this one in terms of computational complexity. But n queens is a hard problem. It needs exhaustive search, and there's a lot of combinations.

And so it's not an efficient algorithm. Efficient means polynomial times something like n square quadratic, n cubed, and so on. And this is not efficient, and the reason for that is this explosion in the number of recursive calls.

So it's easy to write five lines of code that takes forever to run. It will terminate. It's not that it had an infinite loop in it, but the number of combinations explodes on you because of eight and then eight again for each of these and then-- and so on. So you can see that gets pretty large pretty quick.

So this code is something that, as I said, is essentially the same. It's sitting there computing nQueens 20. That's why this is not a problem with my laptop, at least, as of this moment, I don't think it's a problem with my laptop. And so there you go.

So nQueens 20 gave you the representation that we chose. You can try and convince yourself that this, in fact, is correct. I haven't. But that code is so straightforward at this point that if there is anything wrong with it, it would be catastrophically wrong. It would just crash or it wouldn't return anything. And so that is one solution.

There's many, many solutions. Unlike the eight queens case, where I ran it to completion and all the 92 solutions got printed out, there's millions of solutions here. So if I just took out a return statement and let it run, you'll get screen fulls of solutions to the 20 queens problem because if you did that, there would be a lot of solutions. But especially given that we are not exploiting rotational symmetry and reflections, there's a lot of solutions. But that's one of them. So there you go. Yeah. Question.

AUDIENCE: Is there an expression for the number of solutions you'll get, that depends on n?

SRINI DEVADAS: No. There's no closed form solution. No. The 12 is a concrete number, and 92 and 12 were enumerated. Now you could obviously write a computer program that counted-- I mean, you didn't have to print this out. You could count the number of solutions to the 20 queens problem, and you can find that out. But then if you want to do rotations and reflections, you'd have to start shaving things off. Yeah, back there.

AUDIENCE: Do you have the run time for the recursive one? Would that be n to the n?

SRINI DEVADAS: No. Because you're doing significant pruning. Great question. So some of these things get truncated because you end up doing this, and then this one you don't have to go any further. So it's not n to the n simply because you never have to put down another queen if the first two queens conflict with each other.

So you likely did, actually, been back on Thursday about this time yesterday, we were going over the 4 by 4. And we didn't even put a queen down. We went over and we said, oh, this clearly can't work because there's a queen on this row already. This clearly can't work because there's a queen one a way. So there's no reason to go deeper.

You would go n raised to n if you only did the conflict check after you put n queens down. And if you put all n queens down, closed your eyes put n queens down and then ran a conflict check, and then did that over and over, over and over, it would be n raised to n. But that's not what we're doing.

What we're doing is if you look at the code, we are saying if-- this is an important statement, actually. Thanks for the question. That is pruning your search, and that is making it significantly less than n raised to n. A nice exercise. Remember what I said about opportunity for homework, when I get questions.

A nice exercise is to go to 8 raised to 8 and then go to the eight queens problem and figure out what that number is in terms of the number of times you're actually checking for no conflicts. So count the number of times, not the number of solutions, but the number of times noConflicts is being called, and you'll find that it's significantly less than 8 raised to 8.

Because you're pruning the search. Technically, it's called pruning the search high up in the tree. That's the technical term for it. All right? Good. Any other questions? So we're done with eight queens and nQueens.