SRINI DEVADAS: So today, we're going to look at that classic puzzle. It's called the eight queens puzzle, and it's very easy to describe. I'll write down the rules, if you happen not to know chess and how a queen moves on a chess board, in a moment. But before I get into that, I just wanted to mention that the programming paradigms that we're going to be looking at today are-- is, I should say the programming paradigm-- is one of exhaustive enumeration.

And what we want to do algorithmically is to look at all possible cases, all possible combinations, possibilities, because we want to exhaust the space to determine if there's a solution to this particular instance of the problem or not. And one of the problems, as I mentioned, is the eight queen's problem. But you could say, I'd like to solve a smaller problem, the five queens problem or the three queens problem.

And you'd like, obviously, to be able to write code that solves this generalized problem, where you can vary the number of rows and columns on a board. It's no longer a chessboard if it's not eight rows and eight columns, but you can still obviously concoct a puzzle associated with an arbitrary-sized board.

And then the other thing that we'll do today is look at data structures, and so this obviously is going to be a data structure that's immediately clear from what I have up there on the board, a matrix data structure to solve the problem. But can you do things more efficiently with a different selection of a data structure? All right?

So the eight queens problem simply says, place eight queens on a chessboard-- and that implies immediately that it's eight by eight, as I've drawn over there-- such that no queen attacks any other queen. Now, if you had no background in solving this problem, perhaps you knew chess, and you might think there's a solution. You might think there's not. I mean, a queen is pretty powerful.

This implies, because a queen can move horizontally. It can move vertically. It can move diagonally, back and forth, in both directions, up and down, left and right, and I guess up and down diagonally. It's a powerful piece. It's the most powerful piece in chess. And first time I heard about this problem, my guess was that there wasn't a solution to the eight queens problem.

And you know, how many of you think there's a solution to this problem? Do you know there's a solution, or do you think there's a solution?

AUDIENCE: For sure, because knights, because they can't move like in the way a knight can, so it's got to be--

SRINI DEVADAS: So Ganatra says there's a solution. How many of you, before you heard him, thought there wasn't a solution? Well, you know, years ago, I was convinced that there was no solution, because I tried very hard and I couldn't find one. I tried very hard for a half hour, OK? And then I decided there wasn't one. But then, you know, now at this point, it got to the point where I could write a computer program in about a half hour or an hour to solve this problem, and I did discover a solution.

So I'm giving it away. It turns out there's many solutions, and there's certainly a slightly different code for finding one solution versus finding all of them, and we'll do both. So without further ado, you want to translate the movement of the queens into a set of rules that tell you if you have found a solution or not.

And there's essentially three rules, because a queen can move in three different ways. No two queens can be on the same column. No two queens can be in the same row. No two queens can be in the same diagonal. And remember, there's two diagonals, one of which that goes this way, and the other one that goes that way.

Right, so there's a lot of combinations here. If you think about this and you look at this board over here, I have eight different rows and eight different columns. And the way that someone might try to solve this problem is go either left or right or top down. And I've written the constraint no two queens can be in the same column first.

So you can imagine, then, this is what we're going to do. You could certainly modify the code and modify our strategy, but we're going to go left to right. So what we're going to do is, we know if we put a queen up here, we can't put a queen in any other row on that first column, obviously, because of the first constraint. So maybe I'll put a queen up here.

And then at that point, I can move to the second column in my manual strategy. And I couldn't put a queen up here because of the second constraint, no two queens can be on the same row. And if I had a queen up here-- let me just write that out. So I can't put one here, I can't put one here. I could put one here, and this goes back to what Ganatra said, about this is how a knight moves in chess.

And so that gets me two queens. And obviously, not a solution to the problem of the eight queens, but maybe I'm a quarter of the way there, right? And it turns out that you have to do what is called backtracking. And backtracking simply means you may have to undo a decision you made with respect to the placement of a particular queen in this exhaustive search strategy, right?

And the important thing when you do a search, as I mentioned early on, you have to convince yourself that the code you've written is going to exhaust all of the possibilities, and that it won't miss solutions. And if you don't exhaust all the possibilities, your code will terminate, and you may throw up your hands, because you had buggy code, or because you had a buggy search strategy, and say, there's no solution to the problem, whereas there is a solution to the problem. All right?

And the way you do this search, this exhaustive search-- because it's quite intense, there's a lot of combinations, and they grow exponentially with the number of rows and columns on our board-- is important. And so you want to be careful that you don't do redundant work, that you don't do things that you don't really have to, because the combinations blow up.

So there's hundreds of millions of combinations with respect to putting queens up here, so if you really think about it, there's like eight different places you can put a queen here, eight, eight, eight et cetera. So you know, that's 8 raised to 8 right? If you think about that. And that's a large number, right? And if you translate that, I don't even know what that is.

But you could do-- you know, this is 2 raised to 3, raised to 8, so that's 2 raised to 24, which is in the millions. And if you had larger board-- it isn't a chessboard-- then it would be much, much larger as you grew the size of the board. But that 2 raised to 24 doesn't scare us these days, because computers are so fast.

But back when I was your age, that was a large number, OK? And you didn't want to write code that took 2 raised to 24 steps to solve problems. Now it takes a matter of fractions of seconds to run through those kinds of combinations, because you're running at a gigahertz on computers. But before we dive into the specifics of a particular strategy, let's look at smaller problems.

You always get intuition about problems by shrinking them and making them simpler. So let's look at the case of a two-by-two board. Can I solve the two queens problem on a two-by-two board? No. So if I put a queen here, well, immediately I cannot put a queen-- it attacks all of these other positions. So the two-by-two does not have a solution.

Let's look at a three-by-three. Right. So let's say I put a queen up here. So the way I'm doing this, by the way, is column by column, and that's kind of going to be our strategy. As I said, you could flip it to row by row, but no reason to do both. Just pick one. And so I put a queen up here. I can't put one here, I can't put one here, I can't put one here, right? So I go ahead and put that over here.

Can I put a queen here? No, because that attacks it. Can I put a queen here? No, because this attacks it. And clearly, I can't do it over here. So am I done? Can I just give up now? What should I do? I don't want to give up now. I don't want to say that a three-by-three-- at this point, I don't know what the answer is in terms of whether a solution exists for a three-by-three or not.

But at this point, given that I put a queen up here, and I went through this process, I cannot give up, right? And why cannot I give up? What should I do next, once I've failed here? I put a queen here and a queen here, and I said, um, you know, I can't put a queen anywhere here. What do I do? Right? Yeah, go ahead, Fadi.

AUDIENCE: You can change the position of the first queen.

SRINI DEVADAS: You can change the position of the first queen. That's exactly right. So I need to backtrack, and so I tried a few different things here. And it was exactly the same thing, where I had a queen here. And let's say I have a very straightforward strategy that is going to put queens down and run a piece of code that is going to check for conflicts.

This is going to be our most important subroutine that we're going to write, and it's going to be dependent on the data structure that we picked, but that is our fundamental check, you know, the conflict checker. So I could put a queen here. And obviously, with one queen, there's no reason to run the conflict checker. When I put a queen here, I run the conflict checker, and it says, conflict. Here, conflict. Here, no conflict. Right?

And then I move on to the next, and I get conflict for all three of them, right? So at this point, I have to have a way in my code-- this is exactly the enumeration that I need in my code, to go back and say, just like I tried different combinations for the second column before I converged on putting a queen down here, I need to go back and change this to do this.

So I put a queen here, then can I put a queen here? Here? Here? No, right? So that doesn't work either. So then I go up and put a queen here, and now you know where this is headed, because of symmetry. The only place I can put a queen is over here. And once I put the queen over there, I can't put a queen in any of these spots.

So it turns out a three-by-three does not have a solution. At this point, because I've gone through all of the combinations with respect to the starting points on the first column, and then with each of those starting points, I've gone through all of the combinations in the second column and the third column, I've effectively done the 3 raised to 3, which was basically what I described to you previously with respect to these different combinations.

Now I don't know if the 8-- we kind of think at this point that there is a solution in those 8 raised to 8 combinations for the eight queens problem. But there was no solution in the 2 raised to 2 combinations for the two queens problem. And there were no solutions in the 3 raised to 3 combinations or placements for the three queens problem. OK?

So let's do four queens. And the reason I'm doing this is I want to point out one thing, which is an efficiency trick that is going to be important to make sure that our code runs in a reasonable amount of time. Right? And which is kind of implicit in what we've done here, but I want to point it out. So in the case of four queens, I'm going to go off and I'm going to put a queen up here. And I'm going to run through this fairly quickly.

No, no, yes. And so I move on. No, no, no, no, no. Right? So oops-- but now, I don't need to go back all the way to the first. I can put a queen here, right? So now, no, yes, so I get a little further. And then I could move to the fourth column. No, no, no, no. I mean, they attacked from here or attack from there.

Oops, all right. Let's just go-- can I put it over here? No, no. I went back to the most recently placed queen, OK? So one thing that is important, which you kind of get a sense for when you increase the size of the board, is when you fail, when check conflicts returns false, when you fail, you go back to the most recent decision, most recent column that you've placed your queen, and you change that, and you enumerate those possibilities. OK?

And when you do that-- the other thing that we've done, by the way, is for the first queen, we put a queen down with impunity. For the second queen, we check that lone, existing queen on the board for a conflict, right? For the third queen, what am I doing here for the third queen?

What conflicts am I checking when I place the third queen on one of the rows of the third column, specifically?

What conflicts am I checking? I mean, who am I checking with? Yeah, go ahead back there.

AUDIENCE: The two queens that you--

SRINI DEVADAS: Yeah, the two queens. And what am I doing for the existing queens that are already on the board? Do I do anything for them? No. So I guess it doesn't seem like much, but it's an important notion. OK? The notion here is I'm doing a certain amount of incremental checking when I place a new queen, and it's only the new relationships that the new queen-- or the conflicts that the new queen would have that need to be checked.

And in general, you know, if I had seven queens up on the board, and I've set it up nicely with these seven queens-- they're all happy, right? They're non-conflicting. No reason to go look at each of those pairs of seven queens when I put an eighth queen down, as I change the position of the eighth queen. All I care about is that the eighth queen doesn't conflict with the first seven queens.

So I have to check, in general, if I'm putting the k-th queen down, and I'm effectively checking for k minus 1 pairs of conflicts, or k minus 1 queens were already put down. The k-th queen is being put down, and I need to check for the ones that already existed. I don't have to do k minus 1 squared, right? You know, because the k minus 1 were in there. There's different pairs over there. I'm all done with that, all right? I'm all done with that.

All right, so let's keep going here. So where was I? I forget exactly where I was at, but we decided that this didn't work, because I failed on column four. But I can put it over here, and I can put it over here, right? So now, at this point, I've gone through and I've said, look, what it means now is that for these two positions for the first two columns, there's no solution to the four queens problem. OK?

And so I exhausted the positions, for this position. I've also exhausted-- so I can say something even stronger than that, because for this position, I looked at the two possibilities, which were this and that, and neither of them work as well, right? So the stronger statement now is that there's no solution to the four queens problem if you put a queen on the top-left corner, right?

But it doesn't mean that there's no solution to the four queens problem. Ganatra you had a question.

AUDIENCE: Can't you also say there's no solution when you put the queen on the bottom one?

SRINI DEVADAS: Symmetry, yes, wonderful. So it turns out that you can do rotations and reflections, and if you are willing to take your code. Remember that computers are dumb. You have to explain to them what a rotation is and what a reflection is. But if you do that, then it turns out you can get significant reductions in 8 raised to 8, or what have you. Right?

And I'll go back and I'll tell you about the number of solutions to the eight queens problem, if you just looked at it this way, without rotating yourself or reflecting, and then tell you how many unique solutions there actually are, if you took into account the symmetries associated with rotation and reflection. And the last part is important, because it can cut down your combinatorial search, if you do take that into account, right?

But as I said, you know, for our purposes, 2 raised to 24 is not a large number anymore, so we can just blast through and not have to do that. All right? But now what I need to do is I've erased everything. You know, I tried that first queen in the corner. It's probably not going to work here, right? And so because of-- well, no probably. It will not work here. Sorry about that.

But you know, this is different, right? I mean, that clearly we have to check, right? So here, no. Here, no. Here, no. Yes. Right? And then here, yes, right? So I could put that over here. And then I move on. Here, no. Here, no. Here, yes. That's right. So we don't know if there's not another solution. And in fact, there is. There's two solutions.

We won't go there, but 4 by 4-- I'm sorry, four queens has a solution. OK? So now we're ready to code, OK? Now we're ready to code. The one thing we need to do is decide on a data structure. But before I show you the code, or even talk about the data structure, I do want you to keep in mind this search strategy that we employed, right?

And the search strategy was a column-by-column search strategy, and it requires enumeration and changing positions of queens. And deciding when to do that is going to be based on whether check conflicts gives you back true or false. And we can set up check conflicts, so it's not checking-- there's two ways you could do this.

You could say, here's a board. I'm going to check all pairs of conflicts associated with all the queens on the board, and I'm going to just use that procedure over and over. That's not what we did in our search strategy. What we did was we have a check conflict strategy that really takes a new queen as an argument and the existing board as the second argument, right?

And it's really checking for the new queen's conflicts with the existing queens, and that cuts down on the amount of computation that you want to do. All right? So given that, what is the most natural data structure that you can think of that would be applicable to the eight queens problem or the four queens problem? What's the most natural data structure that you can think of? Someone? Yeah, go ahead, Ryan.

AUDIENCE: Two-dimensional array.

SRINI DEVADAS: Two-dimensional array. Two-dimensional list, and that's exactly right. So let's say that I decided to do exactly what Ryan suggested, and what I'm going to do is, I'm going to have a two-dimensional array, or two-dimensional list. In Python, you know, we call them lists, which looks like this. And this is going to look exactly like the board that you have there.

Two-dimensional. This is the list of lists, and you have lists in here. And so the important thing, people-- all of you, if you haven't already, will spend some debugging time in your programming careers, figuring out why things don't work, to try and figure out why things don't work. And it'll be because you flipped the I and J indices of a two-dimensional array, or a list, OK?

Because you'll think that some things are rows and some things are columns, and you'll flip them around. So I don't want to do that. I already did that enough times in my life, so I don't want to do that. And so right here, I think of B0 as the first row. Right? So this thing here is B0. So when I say something like B 2, 3, what is the value of B 2, 3 in my list B? 1. All right? And B 3, 2 is 0,

OK? So that's-- I mean, I could obviously have confused myself. There'd be a bug in the code if I'd switched the identities of rows and columns, right? That's exactly the point.

So I can now imagine that check conflicts, given what we said about incremental checking-- but it still has to do, obviously, computation on the B array, or look at the B array, and decide what's going on. I'm going to be putting in 1s and 0s inside of the B array. And the first two-- I mean, they're all straightforward. I mean, it's not complicated code here.

But the first two checks are slightly easier than the third check. No two Queens on the same column implies that I need to actually look at the first index changing, you know, from 0 to 3.

And for some particular-- I could just write this out here. So no two Queens on the same coin them would mean that I want to go 0 through-- let's just call it-- 7, because I have the 8 Queens problem.

And then I need to look at i where i varies-- actually, let me just call that j. I like i for rows and j for columns. And j varies-- j would be 0. And then you check that B 0, 7, 0 only has one 1, all right? That's essentially what I need to check for.

And then I need to check it for j equals 0. I need to check it for j equals 1. I need to check it for j equals 2, et cetera, right? That make sense? And then, no two Queens on the same row, I need to check that B i and i can go from 0 to 7. I'm sorry, no two Queens can be on the same row. Yeah, that's right. So this goes from 0 to 7. And then I fix the value of the column.

This is fixed. So right now, let's just say that I look at the first row. I'd go 0 here. And then I go 0 through 7 here. And then I make sure that, on B 0, there's exactly one 1, right? All right. So we're good here? Yeah? OK.

The diagonal is the i and j both change, all right? And the thing with the diagonal is-- and I'll show you the code for this. I think it's much easier to explain the code as opposed to drawing things now. Because it's about the specifics of things. And so here it is.

So what I have here is-- I call it noConflicts(). noConflicts() has the board, which is our B array here, or B list. In general, we're going to be working incrementally on the current column. So that's what that is. So you move from 0 to all the way down.

And then you have something that we're going to call qindex, which is where you are. So you have two things that you need to worry about. You're working on a column that's current. And then you also have the position that corresponds to the row where you're putting the Queen down. So you obviously need two of these values for-- to point into the 2-dimensional list. So that's your qindex over here.

And this simply is setting n to be 4. I mean, I could-- this is a general procedure. But because I'm calling this in a 4-Queens procedure, we're going to do iteratively now. So I need to know exactly how many rows and columns they are on my board. So I've just set that up to be n equals 4. But that's essentially that dimension.

So as you can, I'm going to check that there's a single 1 in the current column. This is what I was trying to go over over there. But I think this code is probably more evocative. I'm simply checking to see that the number of 1s in the current column is not greater than 1. If it is greater than 1, then, immediately, I'd return False.

And then here, this is a little bit easier. I check that the current Queen-- this is-- the row on which the current Queen is on only has one Queen on it. And for the-- remember, as I'm going, there's a single 1 in the current column. And this thing here says I need to check that, when I put a Queen down here, that this is the-- I need to check this entire row to make sure that this Queen is the only one on it, all right?

And then I also need to check this and that, OK? And so those two checks are over here, this and that. So the diagonal, the left diagonal, is the first one, the first check. And then the one that goes over to the right is the second check, all right? And again-- yeah, go ahead, Fadi.

AUDIENCE: In the first procedure, like, when we were checking for that only one Queen was in one column-- so I see in my mind that is qindex equal to i. Why is that line added? Like here, we're checking that if we add the Queen in that position, that there's no conflict.

SRINI DEVADAS: Are you talking about this line here?

AUDIENCE: Yes, exactly. And also, because our algorithm makes sure that as we progress, we add, at each stack, one Queen per column, why do we have, in the first place, to check for column conflict?

SRINI DEVADAS: Beautiful. That's a great question. So I'm going to answer that in a minute or so, all right? So Fadi was asking about whether the checks are redundant or not. And he was pointing to a particular piece of code. Let me explain this code. And then I'll answer Fadi's question, OK?

So what I have here is our-- this is actually the search algorithm, all right? And this is exactly what we've talked about up until this point. You're going to go over-- you're going to go column by column, right? And then you're going to go row by row. And so that's essentially what the two loops are.

There's lots of loops here, obviously, right? You know, this is horrible code, right? I mean, I'm not saying this is pretty code, right? But this is horrible code.

And what happens here is, over here, I'm going to go ahead and say, this is-- I'm starting with- - I'm going to vary-- I'm starting with the first column. So the column is fixed here, OK?

Remember, I'm going to column by column. So that's why this is a 0. And then I'm going to vary the position of the Queen on the row, OK? And that i varies, all right?

And then the next thing I'm going to do is I'm going to go to the second column. And I'm going to-- so that's why this is a 1. And then I'm going to vary the position of the Queen on each of the rows corresponding to the second column, and so on and so forth, all right? That's it, all right? It's brutal code, OK?

And I'll show you, I wrote code for eight Queens. It goes all the way to there, right? So that's it. I mean, you know, you can see it's simple code. I'm not-- you know, sometimes brutish means simple-- not always. And so this no conflicts thing is essentially something that is calling this noConflicts() procedure that is doing all of the things that I just described to you, right?

And so if I go ahead and run this, and run this with four Queens-- oh yikes, this always happens. After a while, it's going to come back. But when I run this with four Queens-- let me-- let's see if this is what we want. One more time. Ah, sorry. I've been fighting this system bug ever since I installed macOS Sierra on my laptop. There you go.

So this gave me my two solutions, one of which is up there on the board. And this is the second solution for the four Queens problem, all right? So let me go back to answer Fadi's question, right?

So the idea here-- and the way I described this was, look, as I go column by column, I only put one Queen down on this particular column, the current column that I'm working on. And the way I have this code set up, notice that I'm setting this to be 1, and then I'm resetting it down below here, the j, i, 0, whether it's i-- whether it's this line of code and this line of code, they're paired together-- or this line of code and this line of code.

These two things are ensuring that there's exactly one Queen on any given column. Because I'm only putting that Queen down. And then I'm taking it out and putting it in another spot, right? And that's how I described this entire procedure to you manually, right? That's how we worked through it.

So your contention, Fadi, is that this check is redundant, right? And you're right, right? You're absolutely right. And so if I comment out that check, everything works exactly-- it's a little bit faster. But everything works exactly the same, all right? So there was a reason I put the check in there.

You asked the question. I was going to ask you the question. But that is a good question, all right? So that's kind of the summary of four Queens. You can clearly imagine generalizing this to eight Queens. It looks even more ugly. This is what that looks like, OK?

And in fact, it would look worse, except I decided that I'd employ the continue statement, right? So I don't know how many of you know of the continue statement. But the continue statement essentially says, if there's an F predicate followed by continue, and the predicate is true, then you immediately start the next iteration of the loop, OK? So there's two ways that you could say, I don't want to do anything for the rest of the loop, right? There's two ways you could do this.

One is you could say-- so I have a for loop here, dot, dot, dot. And if I say if condition, then you do stuff, right? And so this do is indented inside of the if. Compare that with, for if not condition, continue. And here you would have do stuff, the same do stuff, all right?

Now, if this do stuff has indentations in it, then you're better off-- I mean, just because I have so many indentations-- you're better off doing this strategy, right? Because if do stuff goes on and has a bunch of ifs in it, then you at least started at this level as opposed to starting at this level, which is 2D, all right? So that's absolutely the only reason why I used continue over here. Because it would look even more ugly if I hadn't use continue, all right?

So if we run this-- and the only other thing-- I'm going to get back to no conflicts in a second. But we'll go ahead and run this. And you see that there's 92 different solutions to the eight Queens problem if you don't take into account rotations and reflections, OK? And all of those solutions are being printed out.

And so then, the question is, what does that mean, right? What does that mean? And that comes to our selection of the data structure, all right?

So we did this. This was the data structure that you picked, which is a natural data structure. You can imagine that you could do something that is more compact, much more compact. And I want to show you how you could optimize this code, both for succinctness as well as memory efficiency, by picking a different data structure, all right?

So let's exploit the fact that each column has only one Queen allowed, all right? So we want to exploit the fact that each column only has one Queen allowed, right? So if you think about it, and you say, well, I have this big thing here, but each of the columns has exactly one Queen, then sure, I could use a bit, like I did before, of 0 and 1 to specify it.

But I do know that this thing is going to be all 0s and a 1, followed by all 0s, or 1 followed by all 0s, et cetera, et cetera, all right? So it's very constrained, if I look at that first column, as to what the value of that column is, right? And so I kind of gave it away by running this. But what could I do now with respect to representing the configuration of the board by doing some amount of compaction, all right? What can I do?

Those of you, look at the screen and-- yeah, go ahead.

AUDIENCE: variable for a row

SRINI DEVADAS: I could just have a single variable that corresponds to the row number that this particular Queen is on, right, in that column, right? And that's exactly what's going on out there, all right? So if I just take that, and I take-- I'll do the bottom one, because that's the easiest for me to see.

But the rows go 0 through 7. So what that says is that, on the first column, I have a Queen up here, OK? And then the third one, I have a Queen. So I have a Queen here. And 0-- uh-oh, I messed up. Thank you. Whew, that would have been bad. So 3, and then over here-- and then, what do I have? 2, right? So 2 is over here.

I'm going to have a panic attack if I end up getting a conflict. Because I drew these-- I put these Queens up here. My name will be Mud, OK? So 5 is over here. And then 1 is this one over here, all right? And 6 is out here. And then finally, 4 is out here, OK?

All right, don't tell me there's a conflict. Even if there's one, I refuse to-- I will refuse to admit that, all right? So that's one of the solutions to the eight Queens problem. And as I mentioned, there's really 12 distinct ones. And so it turns out you don't need to muck with a 2-dimensional list, which is confusing. And you start flipping I's and J's, and your life is miserable, right? So let's just go with a single-dimensional list, right?

Let's just say that if I had something like this-- and in fact, I'm not putting a Queen down on this second column. This is not something we would do in this algorithm, but I just want to show you a generality, right? I'm going to take that, and I'm going to represent this, as you can imagine, as a single-dimensional Python list.

And I'm just going to put-- what am I going to put down here according to what I just described to you? I'm going to put 1. And what do you think I should put down here? Should I put 0? No, no, no, a 0 would be bad, right? Because that would imply that there's a Queen here.

I could put minus 1, right? Minus 1 could imply that there's no Queen on that column. And I could start with a board that is empty, which is all minus 1s, right? And then this one would be-- this is minus 1, yep. Sorry, I wrote 1s. Let me write this like that, minus 1. That's 1. And then I would have 2 here, and then 0 there, OK?

So what's cool about this is the overall procedure in terms of the column by column incremental checking, that control flow of the algorithm can stay the same. I mean, if it were four Queens, the code would look slightly different. Because obviously, our board data structure is a little bit different. I'm going to show you the eight Queens code in a minute. But it's effectively the same algorithm, right?

There's eight loops for eight Queens. There's four loops for four Queens. And a little bit of bookkeeping with respect to putting the Queen on a column would imply that I'm changing minus 1 to 0, or I'm changing 2 to 3, or something like that, as opposed to doing what I did previously, which was putting in a 1 for a particular element in the 2-dimensional list, and then making it a 0, and then putting a 1 in a different spot, right?

So even that gets a little simpler, because I'm just overwriting. When I go 2 the 3, it means that I'm moving this Queen from here to there, OK? So there's that, which is fairly minor.

What is much more interesting is our checkConflicts() procedure or our noConflicts() procedure and what that would look like, right? And we had a bunch of code-- some of it was redundant-- for noConflicts(). But let's look at the code for noConflicts() in this case. And that's it. It's four lines of code, OK?

And this is not redundant, right? So if you guys can make this code smaller, A plus right off the bat, OK? Well, it is a pass-fail class, but I'll write you a letter, OK? So this is essentially the code that would take this new data structure and check conflicts in an incremental way, OK-- same thing. You know, I'm going to add a new Queen to the mix. And I'm going to be checking the conflicts of this new Queen versus the existing Queens, all right?

And so let's go through this. Because I have some time, and that's the last thing I want to do.

So what I have here is, I'm not going to be checking anything about the columns, all right?

Because I know that the number is going to-- basically, the invariant here is there's going to be some number here. It it's a negative number, there's nothing on that column. And if there's a positive number, it better be between 0 and the number of rows minus 1. And that is going to tell me where the one Queen is, all right? So I'm done with that.

But I do have to check that there's a row-- if there's a row problem. There could be a row problem, all right? So I could put a Queen up here. And obviously, I have to detect that, correct?

So the way I detect that is by the highlighted code, where I'm just going to check to see whether there's two numbers that are repeated corresponding to-- well, the current-- so I know what the value of current is. It's not going to be minus 1, because current is going to be a real Queen put into a particular position.

I mean, the last thing you want is to check two minus 1s for equality and throw up a conflict. Because I mean, that's just empty versus empty. You know, there's no conflict.

But board current is guaranteed, by the invocation procedure, to be a positive number-- I should say a non-negative number, either is 0 up until 7 in our case of our eight Queens problem. And I just need to check to see that that value-- let's call it 4-- doesn't exist in any of the other columns, correct? So that's it. It's a little loop, but it's three lines of code.

And in that same loop, we're going to be actually checking for, it turns out, both classes of diagonal conflicts, OK? In this line of code here, can someone explain me-- explain to me what this line of code is doing, all right? What am I doing here? I mean, even pictorially would be fine. Or you know, use your arms and wave, right? What is that line of code doing? I mean, roughly, it's checking for diagonal conflicts, but I want more, all right? How is it checking for diagonal conflicts? Yeah, go ahead.

AUDIENCE: I think the difference between the index numbers of that one we had is that the values have the same difference.

SRINI DEVADAS: That's exactly right. So if you think about what's going on, remember, we're talking squares here. You know, no rectangles, right? Maybe we could have a homework assignment where we have rectangular chessboards, and do things, and confuse everybody, right? But the good news here is that it's a square, right? I like squares, OK?

And squares are easy to deal with, because you end up in a situation where you know that how much you move in this direction should be exactly the same that you move in the other direction, all right? That's what a diagonal in a chessboard is, right-- you know, on a square board is. You need to move in the same amount of movement, all right? So you're not talking about diagonals that look like that, right? That's not what you're talking about.

So what this means is-- if you go look at this line of code, what it says is-- my current is telling me-- for this particular column, it's telling me what row I'm on, OK? That's really what the value-- the value-- I'm sorry, this is the column number. Excuse me, I misspoke. Current is telling me what the column number is, OK?

And this board current is telling me what row number there is. So board current gives me the row number for the current column, all right? And i is whatever column that I'm comparing against. I need to enumerate the different columns that I'm comparing against, all right?

So if I look at this and that, you'll find that the difference between-- this is 1, 1-- 2, 1, right? So this column is 2, and the row is 1, right? So I'll just say 2 equals column, and row equals 1 for this thing over here, all right? And then I'll use a different color for this one. Here I have, the column 3-- the column is equal to 3. And the row equals 0, OK? Right?

Clearly, that's a conflict, correct? And the reason there's a conflict, mathematically speaking, is because the difference in the columns is the same as the difference in the rows, OK? 3 minus 2-- well 2 minus 3-- 2 happens to be greater than-- sorry, 2 happens to be less than 3. 3 happens to be greater than 2. But the absolute value of the difference is something that you need to take into account. Because you could have diagonals in both directions, right?

So that ABS-- the absolute value over there is key, because it tells you that it's checking for both this diagonal as well as this diagonal, right? And so that's it. So if you think about this being a square, and you go look at the comparisons between the values on the rows, and if that value, the difference, is the same, in absolute terms, as the difference in the columns, you have a conflict.

Otherwise, you don't have a conflict, right? Because then you're going off. The diagonal is going away, through the board, but a Queen is not on it, all right? So I guess I did run it, so I won't run it again.

But I'll leave you with the thought-- and we'll talk about this next time-- is how in heck can we make this code look prettier, all right? And we will go and at a-- from here on out, we'll be looking at a programming paradigm that's very powerful. I won't give it away. You probably know what it is. But we'll talk about it first thing tomorrow.

And one of the things we will do, we'll do a different puzzle tomorrow as well. But we'll take a look at the eight Queens code, make it prettier, and also get it to the point where it solves the n Queen's problem where n is an input to do the procedure.

The problem with this code is it only solves eight Queens. I showed you code for four Queens that only solved four Queens. If I told you that I needed you to write code that solves n Queens where n is an argument to the procedure, you wouldn't be able to do that with this iterative strategy unless you bounded and wrote n different pieces of code, right, one of which has four loops, and the other one has five loops, and the other one has six loops. And you would still need to know what n is, OK? All right, so see you next time.