SRINI DEVADAS: All right good morning, everyone. Welcome back. I hope you had a good long weekend. So today's puzzle is, I guess, a classic puzzle. It's Sudoku. I've never actually successfully managed to complete a Sudoku puzzle by myself, because they've fallen into two categories for me. Either they're easy, and I get bored and I stop. Or they're too hard, and I get lazy and I stop.

But what I have done is write a computer program that essentially solves any Sudoku puzzle that is put in front of it in seconds. Maybe there exist puzzles for which it would take minutes, but I haven't discovered such puzzles. And what we're going to do today is talk about Sudoku, compare and contrast the human way of solving Sudoku puzzles against a brute force way, and then try and integrate the two together. You know perhaps this is the closest we're going to get to AI in this class, where we're going to try and marry an exhaustive search method with some smarts.

And back-- I think when we were doing the N-queens puzzle-- one of you asked a question about what the number of possibilities were. For an eight queens puzzle, was it eight raised to eight? And I said, well no. You can prune the search by figuring out that particular partial configurations that correspond to perhaps two queens being placed on the eight by eight board already does not correspond to a solution, because the two queens conflict with each other. And you can then shrink this eight raised to eight substantially.

So that's exactly the methodology that we're going to follow here in trying to take our brute force solver, which will work, given enough time, on arbitrary Sudoku puzzles. But we may not want to wait that long. And we're going to take this strategy of pruning the search to try and improve the solver. And one last thing before I get started on the rules of Sudoku, we're going to have to have a way of measuring performance. Just like we can measure eight raised to eight or four raised to four or what have you, we want to have a way of measuring-- outside of the particular machine that's being used, we can obviously measure real time in terms of seconds, but that's not as precise-- you want to measure more precisely what the number of combinations are.

And so we could certainly instrument our code with appropriate counters that will allow us to measure this performance. And so then it won't matter if our code runs on a fast machine or a slow machine. We can compare it with another piece of code or another variant of the code

and say, oh this new variant is slower or the new variant is faster according to this metric. All right?

So without further ado, let's dive into Sudoku. How many of you have never seen Sudoku, never played Sudoku? All right, so that's fine. It's only going to take me about 30 seconds to explain what the rules of Sudoku are. And then we can dive into trying to, at least partially, solve this puzzle. I do not want to completely solve the puzzle because, as I said, it's either too simple or it's too hard. And I'd rather write computer programs.

And so the rules of Sudoku are simple. So this is classic Sudoku, and it's a nine by nine. There's many variants of Sudoku. In fact, a couple of the exercises talk about two variants, diagonal Sudoku and even Sudoku, I think-- there's probably odd Sudokus as well-- that add even more constraints to the basic constraints of Sudoku that I'm going to write up here. And this is nine by nine Sudoku. And the numbers are one through nine.

And the rules are simple. Each row has all the numbers, which means that no numbers could be repeated, because there's nine columns and nine rows. So there's nine numbers on each row. Each column has all the numbers-- same thing. So there's nine rows and nine columns. And then each sector, which is a three by three grid-- and so that's why I have these overhangs here corresponding to pointing out what the nine sectors are in Sudoku. So this is a sector. That's a sector. This middle one, which is completely blank right now is a sector, et cetera. So each sector has all the numbers.

You can grow the size of the puzzle. It gets more difficult. You could add more constraints. As I mentioned, diagonal Sudoku might say something like both of the diagonals, the large diagonals, the full size diagonals, have all nine numbers on them, et cetera. So that makes the puzzle different. You may have a solution to the nine by nine original Sudoku puzzle, but it may not be a solution to the diagonal puzzle. Obviously the other way around works because the diagonal puzzle only has more constraints.

And so what we do here is try and use implications. Right, so we have these rules. And we'll first forget about computer programs and try and solve this the way people do when they just have a paper and pencil and they have the puzzle in front of them. And they try and use these rules to discover empty positions. And it's kind of hard to do anything with this sector here. You could use the row and column constraints, obviously, even for this sector. Because you have constraints on these three based on the fact that you have nine and seven and one and six on

this row, et cetera.

But usually you go with sectors that have a few numbers in them. You go with rows that have a few numbers in them. And you go with columns that have a few numbers in them. And then you can try and shrink the possibilities. All right?

So just because I don't want to go overboard with respect to looking all over the puzzle, let's focus in on eight-- the number eight. And one of you tell me if I can imply something based on the locations of eight on the top third of this puzzle. Yeah, go ahead.

AUDIENCE: Top middle square.

SRINI DEVADAS: Top middle square, what's your name? Kye? So Kye says top middle square should be an eight, right here. Right, and-- oh, top OK. Yeah that clearly can't be an eight, because this is an eight here. But good, so the claim is this is an eight. And Kye, how did you figure that out?

AUDIENCE: You can eliminate the first two rows because.

SRINI DEVADAS: Right you can eliminate this, because eight can't be here because of this eight. Eight can't be here because of this eight. You need to have an 8 in here somewhere, because eight doesn't exist in the sector. So that would imply that I need to put an eight up here. OK?

So this is what's called a horizontal scan. The only thing that Kye did here was scan horizontally. And you can imagine that-- so Kye did not use, in order to imply the eight-- and so this word implication, imply, is something that we're going to use in a more technical sense as well when we write our code, but an implication essentially says these rules imply the location of the eight. Right?

And we didn't do a vertical scan. We did not use the fact that-- in this particular implication, we did not use the fact that a column needs to have all numbers on it, and therefore all of the numbers on a column have to be unique. Take a look at-- take a look at this part here. And let's look at one. And try and use a more sophisticated form of implication corresponding to both horizontal and vertical scans to imply the position of a one somewhere on the puzzle. Can someone do that? Yeah, back there.

AUDIENCE: In the top box-- in the top right box, it's to the right of six.

SRINI DEVADAS: OK, so the one can't be here. The one can't be here. Right? And the one can't be here. So it

has to be over here. What's your name?

AUDIENCE: George.

SRINI DEVADAS: George-- so George says the one has to be here. And he used both vertical scanning as well as horizontal scanning in order to imply the one. So it's a little more sophisticated. OK, on top of that, obviously, sectors are going to give you some implications as well. And there's no end to this, honestly. There's combinations, there's also a little bit of look ahead, where the hardest puzzles are the ones where you run out of the eights and the ones in terms of the examples that we have here where we've just sort of implied-- without guessing, we've implied the location of a number. And then because of that, our puzzle got smaller in the sense that there's fewer blank locations, blank squares. And then that helps us move forward.

So the easy puzzles are the ones where fairly straightforward implications like the ones we did here always exist, are easy to find. Sometimes you have to search a little bit, look at the top, look at the bottom, look at the middle. And then you fill things in. And because you filled things in, something else now is in play, right? It becomes viable in terms of an implication. The fact that I put an eight in there implies that the eight is now taken-- its location. And so now obviously there's only four left here. And the fact that there's an eight here implies that all of these can't have an eight in them. These seven locations underneath can't have an eight in them, right, because of the constraints.

So this shrinking of possibilities is something that a human does. And you can kind of go through this process. It's an iterative process that you go through. And if you get stuck then you can't do an implication that gives you a number. And some of the harder puzzles you have to-- you have a couple of choices, and only one of them is going to be a correct one going forward, but you don't know that at that moment.

So you now have to guess. And perhaps you put an eight over here or an eight over there. And then you say I'm going to go with an eight over here. And then you go a little bit further, and then you realize, wait a minute, there's no way I can solve this puzzle. Because I need to put two sevens into this sector. And then you go back and there's actually a bit of backtracking that happens-- a wrong guess, and you need to go backwards.

And it's very hard to do for us. It's very hard to do for us with pencil and paper, keeping things in our head, or you know writing down notes on the side of the paper. Whereas it's very easy to do that for a computer program. And we kind of did that already in the eight queens. But

we're going to do that in a much more systematic way over here.

So you kind of weigh these two ways of approaching this problem. One of which is I'm just going to blast through the different combinations, having a giant tree structure in my head of where, you know this might imply that particular grid location grid IJ equals eight. This might imply that grid IJ equals seven. And there's obviously a huge number of combinations corresponding to which of these squares that I pick and what value I assign to those squares. And then once I do that there's another set. And this explodes on you very quickly.

And so if you just did this in a completely brutish way, there's no way your program would ever end, even on a simple puzzle. But thanks to these rules, it turns out a fairly straightforward program that's 20 lines of code is going to solve most problems-- at least the ones that I've looked at-- in a reasonable amount of time. And then it's just interesting from an algorithmic standpoint and an efficiency standpoint to look and see how we can take that fairly naive approach which does have some pruning but it's exhaustive and improve that. All right, so that's kind of where we are headed. Make sense? Any questions about what we have so far? All right.

So what we're going to do is go ahead and look at code for Sudoku that-- so all you need to think about now for the next few minutes, because we're going to move into exhaustive search mode and code things up in a computer, is just those rules. So you don't have to really think about horizontal scans and implications or vertical scans. That's going to come a little bit later. And we're going to vary that up as well.

But we're going to do kind of what we did for the N-queens problem, which is set up a recursive search that is going to explore all of these different possibilities. And the equivalent of no conflicts for the eight queens or the N-queens problem, which said there's no conflicts here because none of the queens attack each other, we're going to have something that essentially says this is valid so far, none of the rules of Sudoku corresponding to these three rules that I have up on the board are going to be violated. All right so let's take a look.

And so this structure hopefully, given that we've done N-queens, should be a little bit easier to understand. So when we did eight queens or N-queens, we decided to start column by column. And we could have done row by row, but we decided to start column by column. And it was a fairly straightforward puzzle. There's obviously no real change in an eight queens puzzle. I mean, you're solving the same puzzle as I am.

But if I give you a Sudoku puzzle, there's more variety to it in the sense that depending on what I fill up-- the hard puzzles are the ones that are kind of intermediate in the sense of they're not obviously fully filled and they're not empty. Right if it's completely empty then it's trivial to solve a Sudoku puzzle. You can take any solution to a Sudoku puzzle and present it as a solution to the empty puzzle. And then if everything is full except for two things, I mean it's kind of obvious what those two things are assuming that the puzzle had a valid solution.

So really it's puzzles like this where maybe a third are full that are more difficult. And it's kind of a separate school, a little community that designs puzzles and tries to create hard puzzles. And they try and make the human's problem harder by making this requirement of look ahead, like I mentioned. So good.

So let's take a look at the code here. So what I want to do here-- and the first part here is-- as I said, we went column by column in the case of N-queens. And the question is, where do I start. I want to do something in a fairly naive way. And what I'm going to do is I'm going to do some sort of scan. I'm going to scan like that. And I'm going to find-- as I do the scan, I'm going to find the next empty grid location. And I'm going to say that is going to be something that I'm going to try and fill in.

OK so it's not going to be I discovered this eight in the-- it was, if you count in terms of the empty locations, if I went this way, it was the fourth empty location and decided to fill that up. But here I'm just-- in this code I'm going to try one, two, three, four, five, six, seven, eight, nine here. And the first part of the code here that says what is the name of this procedure. It says find next cell to fill.

Which means what its name is-- find the next cell. Find the next grid location to fill. And it simply goes for X in range zero through nine, for Y in range zero through nine. I'm assuming that since zero is not a valid entry here, I could use zero to signify empty. OK? It's only one through nine, so zero can signify empty. And this just returns X and Y corresponding to the first empty location. So in this case it would just return (0,0). OK?

And then if this were full then it would go-- obviously the X changes. And when X changes, you're going over to the right. And so you would get a (1,0) back. If this were filled, the next time around I'd get this grid location, et cetera. It doesn't matter. Just like I could go column by column or row by row, as long as I have a deterministic way of discovering the empty location-- and usually you want to have the same way of discovering the empty location. But even that

is not a requirement as long as there's an empty location and your find next cell to fill finds that empty location and returns it to you, you're good, and the rest of our code is going to work. But no reason to get more complicated than what I have up there. So find next cell to fill makes sense? We good with that? All right.

And generally with exhaustive search the key procedure is always do you have a valid solution or not? And you may not have a complete solution. Think of it as a partial configuration. So this is a partial configuration that is valid. It's not a complete solution to the Sudoku puzzle. It's a partial configuration that's valid in the sense that it satisfies all of the constraints. You know, if I put another eight in here it would not be a valid partial configuration. It would be partial, but not valid. Right?

I need to grow this. I need to grow this into a solution. And when I say solution I mean all the constraints have to be satisfied. A configuration could be invalid or valid. A solution is always valid. All right? That's just terminology.

And so I want to be able to look high up and be able to truncate the search and say, you know what, grid IJ equaling eight, because I put an eight in that sector which already had an eight in it, is something that should not be explored. And I don't have to worry about any of the branches that come here. Because immediately I've violated the constraint. So in general, I can always check whether partial configurations violate the three constraints I have or not. And that is what this piece of code does. And it's also straightforward.

It's perhaps even more straightforward than diagonal checking in the case of eight queens. But all this does is use the construct that says I'm going to look at-- essentially this is something that is list comprehensions in Python. The for comes after this predicate here. But effectively what you're saying is for X in range nine, check that grid IX is not equal to E. OK, and you're just looking at E.

So is valid, grid IJE takes the grid which looks like this one, let's say, and so it's got zeros in all of the empty places. And it's got a bunch of non-zero entries in all of the places that you see here. And in addition, you have perhaps zero, zero, and let's call it one. And so this is I, and this is J, and that is E. OK. And so let me write that out here, I, J, and E, where I is one-- zero, I'm sorry, I is zero, J is zero, and E is 1. So that would mean putting a one up here and that obviously is going to violate one of our constraints. But that's fine. We're going to check that.

And it's essentially doing incremental checking just like we did. So it's not checking to see that

all of the existing grid IJ values are conflicting or not. It's just saying I have an-- I'm going to be writing something into this grid, into an empty location. It happens to be zero, zero having the value one. And I'm going to check whether the introduction of a one into this square is going to cause problems or not. That's all that it's doing-- incremental, just like we had with eight queens.

And that check is relatively easy to do, because I just need to go and I look at the row corresponding to I, which in this case is the top row. I look at the column corresponding to J, which is the leftmost column, and then I look at the sector corresponding to zero, zero, which is this top left sector. And I check to see for each of those three things, whether there's a problem or not. And the first two-- well actually, I have a problem with the row. And I would also have a problem with the sector. I wouldn't have a problem with the column. But one of them is bad enough. And so I'm going to get a false. So row OK is going to be false. And so I'm going to return false out here. All right, that make sense?

So those are the three things. And there's really not that much here beyond taking those constraints and codifying them. Right. Any questions? Yeah. Fadi.

AUDIENCE: What is the all thing--

SRINI DEVADAS: Ah, the all is essentially a Python built in function that is going to essentially say that-- it's going to-- it's a conjunction that says I'm getting a bunch of Booleans that correspond to the generation of this list comprehension where E not equal to grid IX is going to give me true or false. And I need all of those things to be true. All right?

AUDIENCE: Okay, it's always going to be a Boolean there and that depends on whether all of the elements of the list itself are--

SRINI DEVADAS: It's a conjunction. Yeah, that's right. So and, think of it as an and. Even if one of them is false, the and is false. In order for the and to be true, then all of them need to be true. All right? So it's just a convenient construct which is applicable in sort of-- the perfect application is what you see here. It's not the most sophisticated of applications, but it works very well in this case.

Now for the sector, I can't actually do that. And so there's a little bit more work, because I can't-- this only works when-- and I could put a list comprehension in here like this and generate all the Booleans. For the sector I end up having to do something a little bit different. I mean you could do things more convoluted and use all in here as well, but it's not worth it. OK,

that make sense? Good.

So here's the core routine that corresponds to the search. And ignore this global variable here. I'll explain that in a minute. That's going to be our metric. Backtracks is going to be our metric for computing performance. And it's going to be quite interesting. It's going to produce some interesting results for us when we run this on various different examples.

But this core procedure looks a lot like the n-queens search in the sense that you have a for loop and a recursive call. And in this case the for loop is going to be something that ranges through the different values, that you find a location that you want to put something into, which is the next empty location in your current configuration. And then you need to go put in one through nine in there.

And it's brutish. You're going to put in one and you're going to check conflicts. And then you'll put in two and you're going to check conflicts. If you put in a one and you don't get a conflict, then you get to recur. And you now move into something that is another partial configuration, potentially, but obviously has one location filled from the caller configuration. And then you go and look for the next cell.

So it's certainly possible that I'd go-- when I put in a one here that fails, but if I put in a two here, it's not going to fail. A two is not going to fail here because, if I just look at those constraints, a two OK. All right, so I'm going to put in a two here. And then I'm going to recur. And I'm going to go out here, and I'll try and put in a one here. And a one is going to fail because of this and that. A two is going to fail because of that. A three-- is a three going to fail? No, not immediately. So I could put in a three here. And then I recur and go to the next one, and so on and so forth, right?

And for each of these things obviously I have to do a bunch of search underneath. And you know thank goodness for fast computers, right? Because otherwise, I mean God, I mean can you imagine the amount of paper we'd generate if you were doing this and putting two and three and I want a new sheet of paper for the four, et cetera, et cetera. I mean, we can count the number of backtracks. That's how many sheets of paper you'll need. OK.

So what you see here, again ignore the backtracks, I'll get to that in just a second-- it's just a way of counting the number of calls. And this thing here essentially says I'm going to be returning-- as long as I get through and find a solution I want to return true. So if solve Sudoku grid IJ is true, then I'm going return through. And then I'm going to pop up all the way to the

top, assuming I got-- I go all the way down to the bottom and I get to the point where I have a solution that returns true, which is a completely full configuration that returns true. Right? But if not, then I need to go try the other combinations and I'm only going to make that recursive call if, obviously IJE, corresponding to this, is valid. And that checks the constraints.

And the only other thing I have to worry about is essentially something that says reset your grid location and make sure that you're setting it back to zero after you're done. Right, and so I've just made a choice here, grid IJ equals E. If I look at this line of code here, this is resetting the grid IJ equals E and saying it's empty. Because if I've failed in all of these and I haven't return true in all of these, then obviously I want to change this. And you could argue that the next time around if I and J are exactly the same-- because I and J are set up here-- then I'm going to overwrite the E from a one to a two, et cetera, et cetera. And so that is, in fact, correct.

But I do need to reset this outside of the loop, if not inside of the loop. So it's not like I can get away with this line of code. In general, if you ever backtrack, you have to go back and undo your decision. And you have to erase the tree. And that's essentially what that grid IJ equaling zero is doing. You just need to undo that decision. And you can do this a few different ways.

But the biggest thing to remember when you do recursive search is to get your-- the undoing of your decision, which is what we call backtracking, to be correct. And if you ever leave a mess, then you'd have a problem. That's also true in the case of the N-queens problem.

So I'm going to go ahead and-- and this is just a print routine. So this is not exactly the Sudoku that I have up there, the Sudoku puzzle that I have up there, but it's kind of roughly similar in complexity. And I could go ahead and run the Sudoku program.

And for each of those different Sudoku problems, it's producing solved puzzles. So this is a solved puzzle. You can check this puzzle just real quick and you'll find that all of the constraints are satisfied. And I'm going to explain backtracks in a second. So true says that there's a solution. The number of backtracks was 579. For the second puzzle, which was a little bit harder, the number of backtracks was 6363. I'm sorry, this is just scrolling. And for the fourth one, it was 335,000-- I'm sorry, for the third one. And for the fourth one, was 9949.

These last two puzzles, hard and diff, there was a Finnish guy called-- there is a Finnish guy called Arto Inkala, who designs puzzles. And he claimed that this hard puzzle in 2006 was the

hardest puzzle ever designed in Sudoku. And then in 2010 he came up with this more difficult puzzle, according to him, that required a lot of look ahead from a standpoint of the human being. Like if we went back to what I said you can't quite do this implication. You have to kind of make a guess. And then you have to go further and further down. And I think the claim was that the hard puzzle required like five levels of look ahead, and then the difficult puzzle required six levels of look ahead.

And obviously, given that look ahead, this puzzle has to have an initial configuration that's solvable. So it's not a trivial thing to create puzzles. But now people are using computer programs and doing things like we're doing here to find difficult puzzles. And interestingly enough, the 2006 puzzle, at least for this naive computer program, takes 335,000 backtracks-- the one that was supposedly made more difficult in 2010, which now takes about 10,000 backtracks. So obviously there's a difference between the way this program behaves and how you or I would behave, or rather you would behave if you tried to solve this puzzle.

So let me just explain backtracks, and then I'll stop to see if there's any questions about the code. So when you make recursive calls and you want to count the number of recursive procedure calls-- you want to do something inside each of the recursive procedures and you want to sort of cumulatively or collectively keep some information, one way of certainly doing it is to pass arguments. And then you have to return the argument, because when you pass an argument and you modify it it's not like that is going to be-- that modification, if it's just an integer, if it's not a mutable variable, it's not going to be seen by the caller procedure.

And so when you do recursion and you want to do some counting, the notion of global variables is a convenient construct to have. And global variables essentially say that there's exactly one memory location associated with this variable. And we're going to go ahead and, anytime we are mutating this variable and you're modifying it, you're going to see the effect of that in that memory location.

So what you have up here is, I set backtracks to be zero. OK and that's my global variable. The fact that I put backtracks equals zero here doesn't make this a global variable just yet. The fact that I have global backtracks inside of solve Sudoku now says that there's a single copy of backtracks, and it doesn't matter whether I'm at the top level of recursion or the bottom level of recursion. It's just that memory location corresponding to backtracks-- the name backtracks, that is getting incremented. And this could be 10 levels deep. It could be 40 levels deep, given that I've called things 40 levels in. But it's just the one backtracks.

So as you can see, what backtracks does is anytime you have a valid location and you've gone ahead and-- essentially you've failed. The reason it's out here is solve Sudoku did not return true. When solved Sudoku actually returns false, that's when you come out and you increment backtracks. So it meant that you had to do some undoing. When you set grid IJ to be zero, that's when you're undoing your guess, right?

So backtracks makes sense from a standpoint of I need to backtrack and go in a different fork in the road. And so that's why I have backtracks plus equals one when I'm undoing my decision that I made. So this kind of gives you a sense for how many wrong guesses that this program did. And as you can imagine, the more the number of wrong guesses, the more the computation and the longer it takes.

So it is definitely a proxy for performance. But it's a platform independent proxy that's more algorithm related as opposed to the speed of the computer. Because if this computer were twice as fast, I mean I'd just see things running faster even though the algorithm isn't any better. Right? That make sense? So it's a very simple use of global. You don't want to use global variables except in certain constrained settings. This is a fine use of global variables. Cool, good. So any questions about this code?

So what I've done here is I just have the naive code. And I happen to have different numbers of backtracks because I have different inputs. Unlike the N-queens problem, which is kind of boring in some sense, because once you've solved it there's nothing left, in the case of Sudoku, I could change my input, my starting point, and give you different problems.

And so the reason we had many different kinds of backtracks was simply because-- numbers of backtracks was because we had four different inputs to the Sudoku puzzle. All right, so are we good here? People understand this code? You're going to have to modify it, right? Not necessarily this code, depending on the exercise you do, but this is certainly something that hopefully you feel comfortable with potentially modifying.

All right so what I'm going to do now is first I'm going to go ahead and show you some code that corresponds to something that is the original code, except that I'm going to add some smarts to it. What I'm going to do is, at any given point of time, I'm going to try to do some implications without actually doing any guessing.

So the way I'm going to integrate the human approach into this exhaustive search approach at

top level, is I'm going to take my configuration, and before I do an arbitrary guess, before I call find next cell, or maybe I have a particular location here that I'm eventually going to guess. So I do know that. But before that, I'm going to try and see whether the current grid values imply anything or not by using the rules in exactly the same way or roughly, I should say, the same way that we did right when we began the lecture. All right? So we're going to try and use some implications and maybe imply the eight or imply something different associated with some other location.

So this is not a backtrack, in the sense that this is going to be-- I can take this to the bank assuming I haven't done any guessing up until this point, and assuming that the initial configuration that was given to me corresponds to a valid solution. But I'm actually going to do this at different points in the search. So it might be that I'm just going to arbitrarily choose a two here.

And so I go through and I'm going to take this, for argument's sake, and I'm going to put a two down. And then I have not the initial puzzle that was given to me, but something that I've kind of hacked in the sense that I've stuck a two in there. And that may not correspond to the solution, because I just sort of put the two down there. But now given the two, I'm going to try and do some implications. And I'm going to try and see whether there's things that are valid or not. The important thing is that, because I put a two down in an arbitrary way without using implications, the two could have been incorrect. I mean that's exactly why we have all of these backtracks, correct? Because I've put down incorrect guesses and then I've had to backtrack.

So once I put a two down and then I fill in a bunch of things with implications. You know, I may even put an eight up there. I may put a six out here, et cetera, et cetera. And I go deep in and then I realize, ooh, you know that two was a mistake. The two really shouldn't have been in there. Now I have to clean up everything. I have to clean up all of the guesses that came after two and all of the implications that came after two. All right?

That's the biggest thing that I want you to take away from this integration of implications with exhaustive search. It's clean up your mess, clean up your bad guesses. The fact that-- you say, oh but the implication was something that was deterministic. It was exactly following these rules. No, no, no, no, no. It was deterministic. All of that is true. But you made a wrong guess. And therefore everything that you did from then on out is in question.

And if you, in fact, find a contradiction, you've got to go all the way back and clean up

everything. And then go back and erase everything that you had. And then go take this two and maybe turn it into a three or what have you. All right?

So before I show you the code that does the implications-- and you can kind of imagine that there's many ways that we could do implications, we did that manually. I want to show you this part looks exactly the same as before, no change. Find next cell to grid is exactly the same. Is valid is exactly the same, right? There's a large make implications procedure and an undo implications that I'll get to in a second. But this part here looks almost exactly the same, except that I've replaced grid IJ equals E with make implications.

And this is something that not only is-- what make implications is going to do is it's going to set-- whatever I had up here, it's going to set two up here. And on top of that it's going to go use these things to go fill in a bunch of different values in here. So it's one extra step. This is the integration that I talked about. So the idea is that-- now you can do this for the original as well.

But the point is, once you've made a guess, you always want to check to see whether that guess does certain implications or not. Right? I mean that's the whole purpose of this exercise. Even humans do this in the very difficult puzzles. They make a guess and then they see whether there's some implication or not. And maybe there's a contradiction and they have to go back and undo all of that damage they caused and change the guess. But in general, when you have a configuration and you add to it, it's possible suddenly that there will be other things that are implied by the one change that you made to it.

So grid IJ equals E in the original code got replaced with this procedure that we'll talk about, which I don't want to spend a whole lot of time on, but it's essentially something in terms of details. But it's essentially something that puts in different values in the different locations. And grid IJ equal zero is replaced by undo implications, which is cleaning up all of the incorrect guesses and incorrect implications. And the reason the implications are incorrect-- because it came from an incorrect guess.

And so that's it. Undo implications is trivial. It just sets all of the implications, and I'll tell you what the data structure is in a second, but think of it as making everything zero, going back to a clean slate. I mean clean slate in the sense that all of the incorrect implications and guesses are cleaned up. So that's all there is over here.

Make implications is-- you can do anything you want. You can do vertical scans. You can do

horizontal scans. You can-- if you go look at Sudoku literature and you look at ways of playing Sudoku, there's books written on how you can become a better Sudoku puzzle solver. And you could take that, and you could code that in. And you could replace make implications with those fancy techniques that are up there, right? But we've established I'm lazy. And so I only write a certain amount of code, and then I get tired. And so I wrote about 20 lines of code corresponding to a fairly straightforward implication just to give you a sense of how this would work.

But the most important thing in here is not the details of make implications. And I'll give you some sense of that before we're done. But it's really the structure that is the most important. The fact that I've done make implications here and undo implications here is the correctness requirement that is important to exhaustive search.

So if I do this and I do kind of the implications that we had right at the beginning of lecture and I go ahead and run it, just take a look. I won't write this out, but remember what the backtracks are for these things, roughly speaking, for the original Sudoku. Oh, I'm sorry, I need to go to the shell. And it was 335,000-- what is it-- 579, 6363, 335,000, and 9949. So if I go off and I run Sudoku optimized, which is doing these implications like I describe, and I go ahead and run that.

The first one goes from 579 to 33 backtracks. OK so that's pretty good. Because it's done a bunch of implications. It's still-- it's not super smart. I mean that is a simple enough puzzle that a human being would not backtrack. I mean a human being would not backtrack in that first puzzle, right? And you should check that. And-- oh, this thing finished in the middle. So it went to 33. Oh, only had three of them? What do I have here in Sudoku Opt?

Oh I see. I only ran-- oh wow. OK so I ran inp2, hard, and difficult. So it really went from 6363 to 33. It went from 335,000 to 24,000. And then it went to-- 7-- went from 9949 to 726. The details aren't-- the numbers aren't super important. Don't hang your hat on them. Obviously if I change the code those numbers change. But you can see that there are substantial gains to be had in terms of implications not making these dumb guesses that clearly are incorrect. And you can fill in-- if you take away some of these empty squares, then the depth of the recursion that you have to go through becomes substantially smaller. And that's why your backtracking is simpler.

So I want to leave you with a couple of things. I want to give you some sense for what

particular implication that-- a strategy that we used. And so I'll just put up make implications and give you some sense for how this works. So the basic idea is that what I'm doing here is I'm looking at a particular sector. And I've created a data structure that says the missing elements here-- if I put a two in here-- let's just say I go ahead and put a two in here. The missing elements here are-- the set is three, four, five, six, seven, and nine.

So this could be three, four, five, six, seven, eight, nine. This could be three, four, five, six, seven, eight, nine. This is quite dumb right now. But each of these different squares could be three, four, five, six, seven, eight, nine. OK? Possibly, all right. And then I say-- so that's the first part of the code. And then I say I'm going to attach, essentially, a copy of the set to each of the missing squares. And then I'm going to go through and find the missing elements.

So this thing here can't be a nine because I see a nine here. It can't be a three, right? And so I can take this thing here. And I take away the nine. And I take away the three. And I can do the same thing with that. Obviously I can also take away the-- the eight isn't there, but I could take away the seven, and I could away the three, the six, and the one. So I go ahead and I take away the six. And the three was already taken out. And I keep doing this. And I try and shrink the possibilities corresponding to this particular square that has the set of different possibilities.

And if I ever-- so when can I make an implication? What is the condition that is going to let me make an implication when I take this set of numbers and I start shrinking them down using these rules that I have over on the right hand side there? What is an implication? What does that correspond to in relation to the size-- in relation to the set? Right, yeah, behind you, Ryan.

AUDIENCE: So if you only have one element.

SRINI DEVADAS: That's exactly right. If you have one element in the set, then that's an implication. If I have two elements in the set, it's not an implication, because I don't quite know what to do there. But if I had one element in the set, that's an implication. And that's it. That's-- you know this code is not complicated. Check if the vset is a singleton, which is a single element. And I'm going to go ahead and append to this implication, which is a very straightforward data structure that says this is the grid location I, grid location J, and this is the value that was implied by that.

So not only do I have IJE, which is the original guess that I have, I also have kind of a bunch of other tuples corresponding to different coordinates, you know, KL coordinates and the value, call it V, associated with that. And these are all the different implications that I can collect together in this list. And I can just add those things into make implications. And then I keep

going. And then if I ever realize I've made a bad guess, I have to undo everything by zeroing them all out, which is making them all empty.

So one thing that this code does, and you can take a look at it. And I would encourage you to do the first exercise, which is taking these implications and making them a little more powerful by adding three or four lines of code to this code. And exactly what you have to do in this exercise, and I'll show you what the results should be in just a minute. But let me just spend 30 seconds explaining to you how you could do a little bit better than what this code does.

So what I've described to you really is get this set, imply, get a singleton, et cetera. And then you can do this, obviously, for each of these sectors. And that's what this does. You had a for loop up there that does it for each of the sectors. Grab a sector and go ahead and do an implication for that sector. Now this code just runs through the sectors, you know, One, two, three, four, five, six, seven, eight, nine and then discovers the implications if they exist, adds them to the imply list, and then throws up its hands and says I'm tired, I'm done, I don't want to do any more.

What could you do that's an improvement, given what we have described and what I've told you so far. What is an incremental improvement over going over these sectors once and doing these implications and storing them and moving on? What is an incremental improvement? Ganatra?

AUDIENCE: Look, once we get all the singletons, we can set those as-- since those are determined, like, deterministic, I think that we could set those into the original grid and say that's our new base grid and run through it again.

SRINI DEVADAS: Run through it again, exactly. You don't have to stop. There's no reason to stop if you're implying. Once you've put something in here and you've gone through one, two, three, four, five, six, seven, eight, nine, got the implications, you can put them into the grid and then start over again. One, two, three, four, that's what humans do. Right? When humans put something in, then they don't stop. They just keep going until they get to the end.

Now of course all of these implications could be incorrect if that first guess was incorrect. There's no change there. But there's nothing that's stopping you from turning this little thing-- there's a loop here that simply corresponds to making a pass over the sectors, but you can put this whole thing into a loop. And you keep going through the loop until you basically have no change that happens in your grid. OK so that's four lines of code. And I'm not going to show

you what those four lines of code look like, so close your eyes in case you--

And this is the solution to that code. And I'm going to go ahead and run it. And you saw what those numbers were with respect to the backtracks. But if you do those extra implications, the 33 went down to two for that example. So this is not optimal, because I wanted one. So if I wanted to be a human being that took this easy puzzle and just sort of went all the way without making any incorrect guesses, I would be doing implications. And that would go all the way. And I got close with two. And I didn't print out the intermediate ones, but the 24,000 went down to 11,000. And I forget what the last one was. It went down.

So with four lines of code and with the optimized code that I'll put up you should be able to get those numbers in your first exercise. Or you could solve diagonal Sudoku or even Sudoku. Or you could spend the rest of the day coding whatever you want, whatever. All right, see you next time.