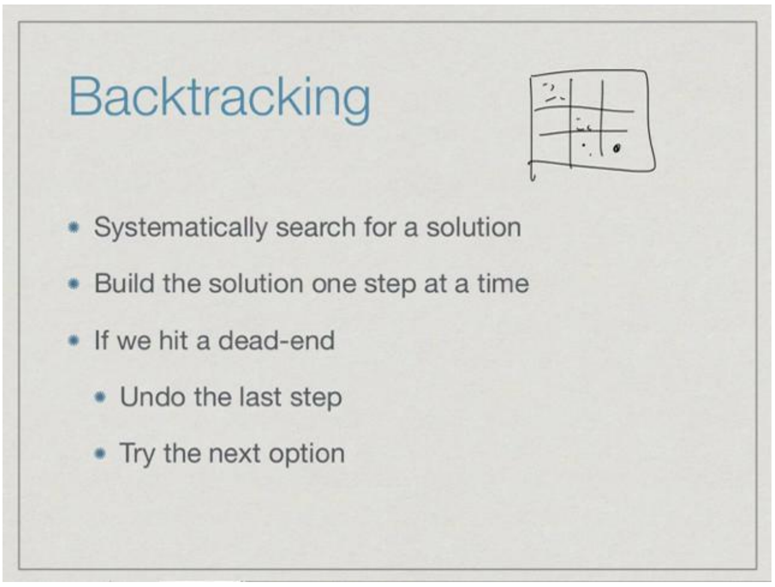
Programming, Data Structures and Algorithms in Python

Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 06
Lecture - 01
Backtracking, N Queens

For many problems, we have to search through a set of possibilities in order to find the solution.

(Refer Slide Time: 00:02)



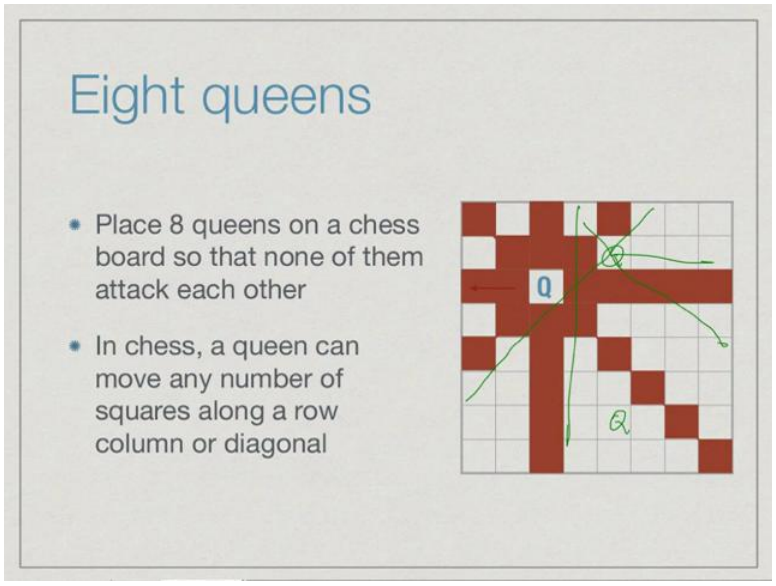
Backtracking

- Systematically search for a solution
- Build the solution one step at a time
- If we hit a dead-end
  - Undo the last step
  - Try the next option

There is no clear solution that we can directly reach. So, we have to systematically search for it. We keep building candidate solutions one step at a time. Now it might be that the solution that we are trying to get does not work. So, we hit a dead end, and then we undo the last step and try the next option. Imagine for instance if you are solving a Sudoku. So, you have a grid and then you start filling up things and there are some points you realize that there is nothing you can put here.

Then you go back and you have to change something you did before. So, we have to backtrack, we have to go forwards trying to solve the problem; and at some point when we realize that we are stuck we cannot solve the problem again, we have to go back and change something we have done before and try something else.

(Refer Slide Time: 00:53)



Eight queens

- Place 8 queens on a chess board so that none of them attack each other
- In chess, a queen can move any number of squares along a row column or diagonal

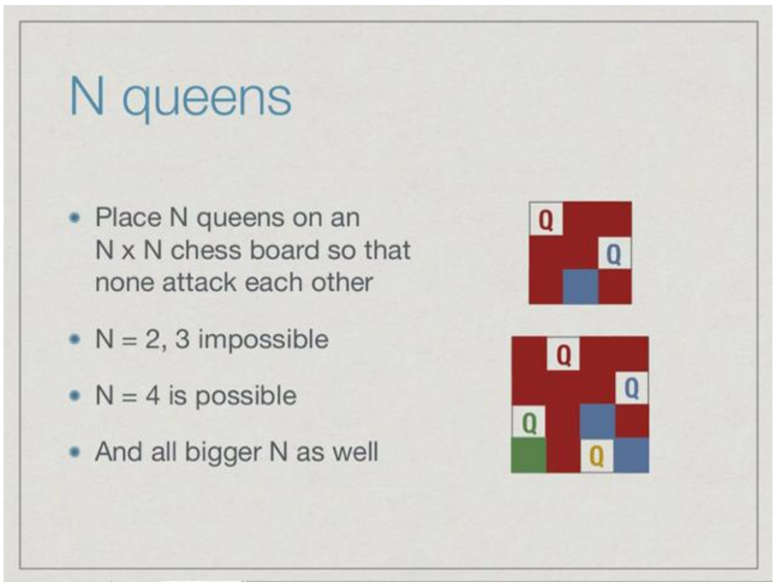
One of the classic problems of this kind is called Eight queens problem. The problem is to place 8 queens on a chess board so that none of them attack each other.

Now, if you have ever played chess, you would know that the queen is a very special piece it can move any number of squares along a row, column or diagonal - for instance, if we place the queen here, in the third row and the third column, then it could move anywhere upward down the third column anywhere left or right on the third row, and along the two diagonals on which the square three comma three lies.

Since it can move along these columns it can also capture any piece that lies along these rows. The queen is said to attack all these squares. The squares to which the queen can move are said to be attacked by the queen. So, our goal is to place queens so that, they do not attack each others, so if we have a queen here then we cannot put another queen in any of the red squares, we have to put it somewhere else. For instance we could put a new queen; say for instance here this would be ok; or here.

And then, it if I put a queen here in turn it will attack more pieces like, it will attack these squares and you rule out some more options so I will not be able to place queens there and so on. So, we want to see if we can place 8 queens. Now we cannot place more than 8 queens; because, there are only 8 rows if you place 9 queens, 2 will be in the same row or the same column and the same column. They will have to attack each other. So, 8 is clearly the limit, the question is whether we can actually put 8?

(Refer Slide Time: 02:28)



N queens

- Place N queens on an N x N chess board so that none attack each other
- N = 2, 3 impossible
- N = 4 is possible
- And all bigger N as well

So, we can generalize this question and ask not for 8, but N. Supposing, I have a chessboard in which there are N rows and N columns. Can I place N queens on such a chessboard? Now for N equal to one the question is trivial, because you only have to put 1 queen on 1 square. Now, it is easy to see that N equal to 2 is impossible because, if I have 2 squares and wherever, I put a queen say here it will attack all the remaining squares. No matter where I put the queen, every other square will be on the row, column or diagonal of that queen.

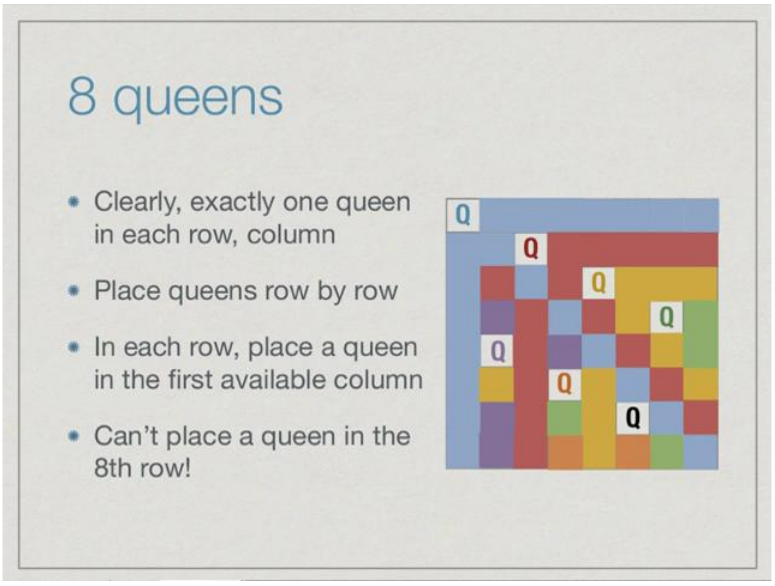
And so there is no possibility of putting a second queen. It turns out that three is also impossible. Supposing we start by putting a queen in the top left corner then we will see that it blocks out the first column, the first row and the main diagonal. This leaves two slots open for the second queen, but wherever we put, whichever of the two we put, it will block the other one.

Once we put a queen in one of those slots the other one is on the same diagonal and there is no free slot for the third queen. So, just by exhaustive analysis we can show that, n equal to three is actually impossible. For N equal to 4 for 4 a 4 by 4 board, it does turn out to be possible. We should not start at the corner, but one of the corners. Supposing we put it in the second column, then we get this pattern of block squares.

Then we can find an empty slot on the second row right at the end. So, we put a queen there it blocks of certain of some more squares in the last column and in that diagonal, but this still leaves one slot in the third row, unfortunately the third queen does not block the last two slot on the fourth row and we have this kind of symmetric pattern where everything is one of the corner in which none of the queens attack each other.

Now, it turns out that once we cross N equal to 4, for 5, 6, 7, 8, you can show that there is always a solution possible. Our task is to find such a solution. How do we find a solution for N greater than or equal to 4?

(Refer Slide Time: 04:24)



8 queens

- Clearly, exactly one queen in each row, column
- Place queens row by row
- In each row, place a queen in the first available column
- Can't place a queen in the 8th row!

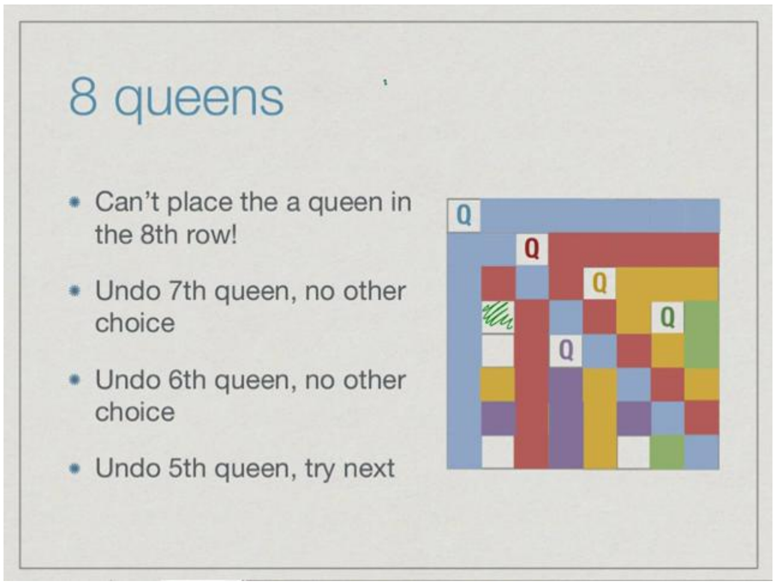
So, as you observed, the first of first thing you know is that there can be exactly one queen in each row and in each column because queens attack the column and row on which they lie. If we have two queens on the same row or the same column they will necessarily attack each other. Since 8 is the classical size of a chessboard let us look at specifically our example for 8 queens. So, we want to place the queens now row by row. We know that there is exactly one queen in each row.

Let us first put a queen in the first row, then based on that put a queen in the second row and so on exactly as we did for the 4 by 4 case that we saw in the previous slide. So, in each row we will place the queen in the first available column, given the queens that I have already been placed so, far by available we mean a square which is not attacked so far. So, we start with an 8 by 8 board and in the first row now everything is available. By our analysis we are going to put a queen in the first available column, namely the top left once we do this; it blocks out the first row and column and the main diagonal. So, all the shaded squares are now under attack. We move to the second row and we try to put a queen in the first available column this is the third one and this in turn will attack another set of rows, columns and diagonal squares.

Now, we move to the third row and in the 5th column we can place a queen. And this one again attacks some squares. So, we have added some colors to indicate, as each new queen is placed which squares are newly under attack by the new queen, some of them are attacked by multiple queens. For instance the yellow queen attacks the blue square on the diagonal which was already attacked by the first queen.

So, we will leave it blue for now. In this way we can proceed. So, we put a 4th queen on the 4th row, and then this is a mistake this should be already attacked by this queen and then we will place a 5th queen and then a 6th one and then a 7th one and now we find that all the squares in the 8th row are actually blocked. There is no way to extend this solution to put the 8th queen. So, we have to do something about this, we cannot place a queen in the 8th row.

(Refer Slide Time: 06:36)



8 queens

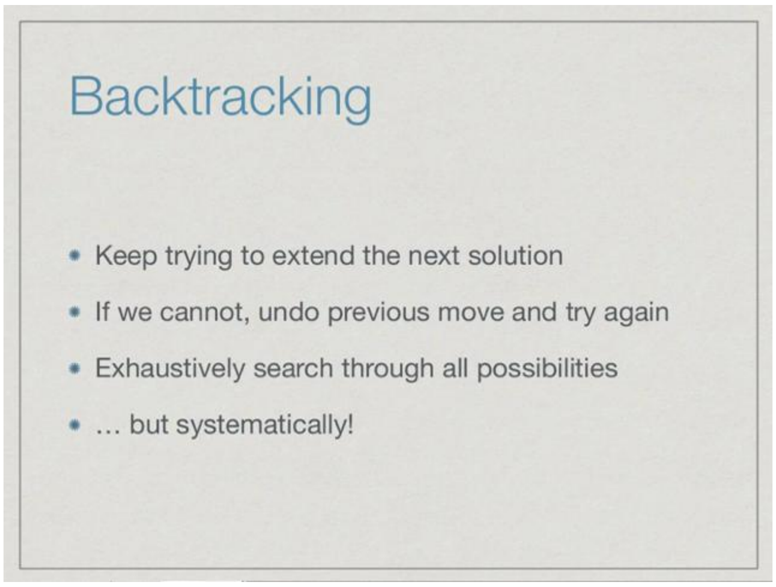
- Can't place the a queen in the 8th row!
- Undo 7th queen, no other choice
- Undo 6th queen, no other choice
- Undo 5th queen, try next

Since we cannot put a place with queen in the 8th row we have to go back and change something we did before now. The last thing we did was to put the 7th queen right. So, we do that and we find that unfortunately for the 7th queen, we had only one choice. So, we have no other choice for the 7th queen. Though the 7th queen could not lead to a solution, it was not the choice of the 7th queen, which actually made a problem, but it was something earlier.

Then we go back and try to move the 6th queen. So, once again if you remove the 6th queen then this unblocks a few squares, but at the same time there was no other place to place the 6th queen on the 6th row. So, again this was a unique choice that we had made. Now if we go back to the 5th queen then we find that there is a way to place the 5th queen. In a different place namely it move it to this slot. So, we can move this 5th queen to one slot to the right and try again.

So, having gone back from the 8th square and, so 8th row which is completely blocked, to the 7th row which had only one choice, to the 6th row which had only one choice we come back to the 5th row and now we try the next choice for the 5th row. If we try the next choice for the 5th row - then we get this pattern of squares and now we see for example, that we cannot put a 6th thing. So, both the choices for the 5th row actually turn out to be bad. So, you would now have to go back and try a different choice for the 4th row and so on.

(Refer Slide Time: 07:55)



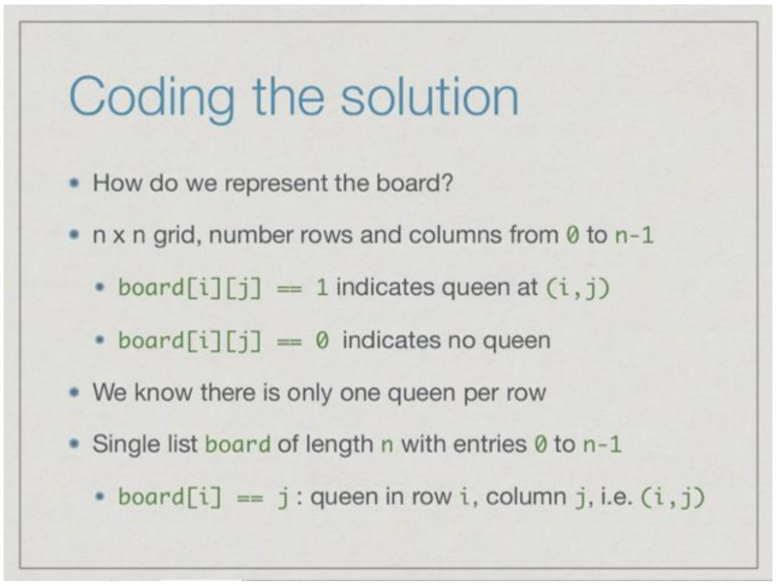
Backtracking

- Keep trying to extend the next solution
- If we cannot, undo previous move and try again
- Exhaustively search through all possibilities
- … but systematically!

This is what backtracking is all about, we keep trying to extend the solution to the next step if we cannot we undo the previous move and try again, and in this way we exhaustively search through all the possible solutions, but we do it in a systematic way we do not go back and randomly reshuffle some of the choices we made before we go back precisely one step and undo the previous steps.

So, at each step we have a number of choices we go through them systematically, for each choice we try to extend the solution if the solution does not get extended we come back we try the next choice and when we exhaust all choices at this level we report back to the previous level that we have failed then they will try their next choice and so on. The key to backtracking is to do a systematic search through all the possibilities by going forwards and backwards one level at a time.

(Refer Slide Time: 08:47)



Coding the solution

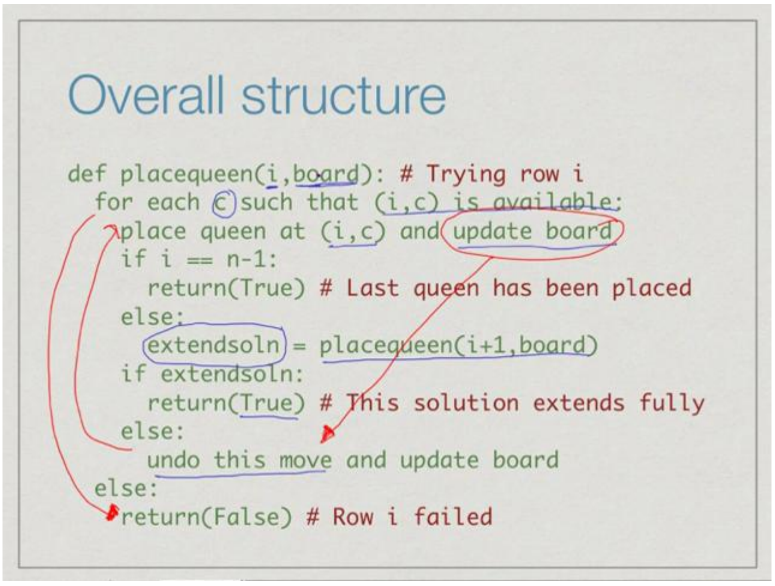
- How do we represent the board?
- n x n grid, number rows and columns from `0` to `n-1`
  - `board[i][j] == 1` indicates queen at `(i,j)`
  - `board[i][j] == 0` indicates no queen
- We know there is only one queen per row
- Single list `board` of length `n` with entries `0` to `n-1`
  - `board[i] == j` : queen in row `i`, column `j`, i.e. `(i,j)`

So, how would we actually encode this kind of an approach? Specifically, for the 8 queens problem, so our first question is how to represent the board because a board is what keeps changing as we make moves and undo them. The most obvious way for an N queen solution is to represent the board literally as an N by N grid. And since python numbers list position from 0 onwards we have an N by N grid and we number the columns not 1 to N, but 0 to N minus 1, so will have rows 0 to N minus 1 and columns 0 to N minus 1. We can now put a value 1 or 0 or true or false to indicate whether or not there is a queen at the square i comma j; i is the row, j is the column.

So, we can have a two dimensional list, board or list of lists, which has N minus 1 by N minus 1, 0 to N minus 1 and 0 to N minus 1 as a valid indices and we say that board i j is 1 to indicate that the queen is at i comma j.

And therefore, if it is 0 it indicates there is no queen. There are two possible values for every square. Of course, we also know that there is only one queen per row. This particular thing though it has N minus N into N N square entries it will only have actually N ones at any given time. So, we can optimize this slightly by just having a single list with the entries 0 to N minus 1 where we say that the ith entry corresponds to the ith row and we record the column number. So, if board of i is equal to j it means that in row i the queen is at column j. The queen is at position i comma j.

(Refer Slide Time: 10:36)



Overall structure

```
def placequeen(i,board): # Trying row i
  for each c such that (i,c) is available:
    place queen at (i,c) and update board
    if i == n-1:
      return(True) # Last queen has been placed
    else:
      extendsoln = placequeen(i+1,board)
    if extendsoln:
      return(True) # This solution extends fully
    else:
      undo this move and update board
  else:
    return(False) # Row i failed
```

So, with such a data structure this is the outline of how our strategy works. So, what we have to do is place each queen one at a time. So, we are just writing a function which tries to place a queen in row i given the current state of the board. So, we pass it the current state of the board as one argument and we pass it the row number i that we are going to do. So, we would initially start with an empty board and with row 0. Now we run through each column and check whether the row column position that is the square i comma c is available, if it is available we then put a queen there and we of course, have to update the board. So, we will come back in a minute, but in our case updating our board just means setting board i equal to c if we have the one dimensional representation.

Now, if we have actually put the last queen, if I was N minus 1 then this is the last queen right. So, if it is an 8 queen problem then when we have put queen number 7 starting from 0 then we are done. So, we can return true; however, if this is not the last queen then we have to continue. So, what we need to do is now with the new board we have to place one more queen. So, we recursively call this function incrementing the row to i plus one with the updated board which we have just put and this will return true or false depending on whether it succeeds or not. So, we record it is return value in the name extend solution. Depending on whether it succeeds or not we check if extend solution is true that is the current position reaches the end.

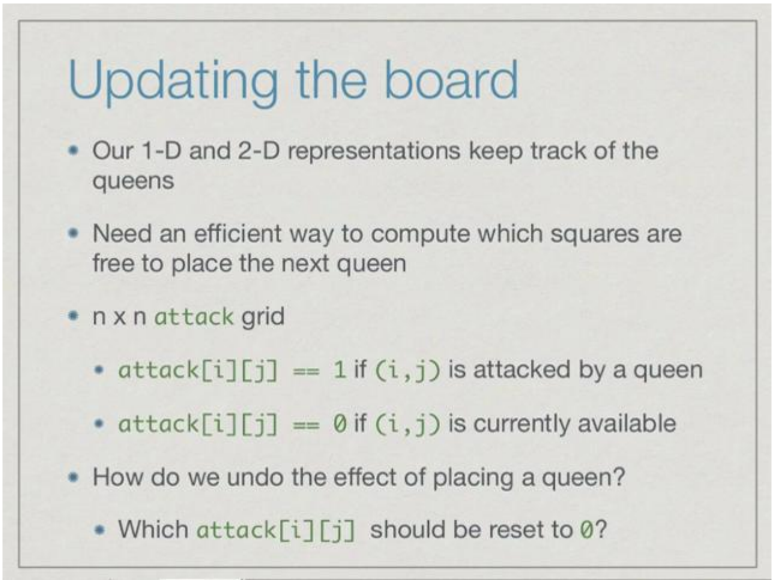
Now, when would it be true; if it succeeded in going all the way to level N minus 1 and N minus 1 returns true. So, when N minus 1 returns true then N minus 2 will return true and so on. Then our level I will also get the value true. Then we can also return true. So, if extend solution returns true we also return true saying that, so far I am good. On the other hand if extend solution returns false it means that given the current position that I chose for row I, nothing more could be done to extend this to a full solution. This position must be undone. So, we have to undo this move. So, we have to whatever we did earlier to update the board.

This update has to be reversed at this point. So, we have to reverse the effect of putting at i c and then, when we do this we will go back and we will try the next c and now if we have actually run through all the c's and we have not returned true at any point, then python has this else which says that the for loop terminated without coming out in between.

The for loop terminates normally it means we have run through every possible c that was available and for none of them did we return true; that means, that there is no way to currently put a queen on row i given the board that we have. So, we should return false saying that the board that we got is not a good one, then the previous row will now get a false and we try the next position and so on. This is a recursive solution that we get we will see an actual python implementation, but we have to do a little bit more work to figure out how to actually implement this.

The crucial thing in the implementation that we saw the previous one is, that we have to update the board when we place the queen and update the board when we undo it and we also have to check whether i c is available.

(Refer Slide Time: 14:01)



Updating the board

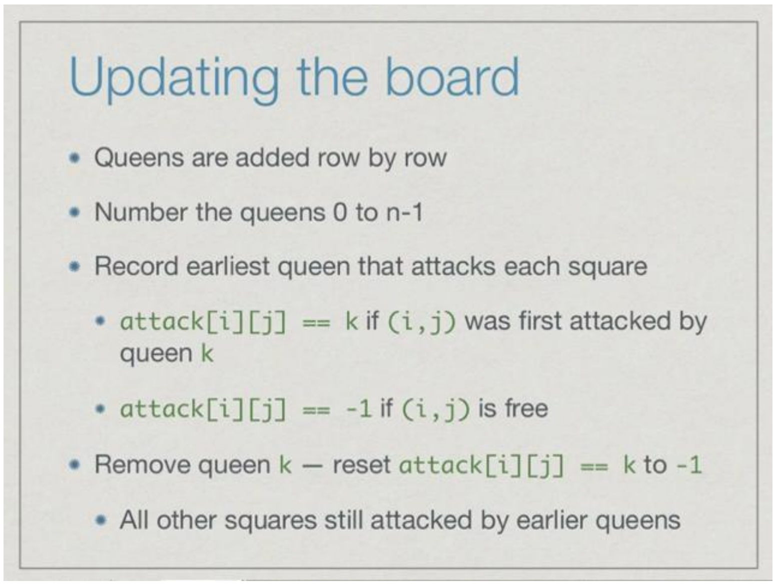
- Our 1-D and 2-D representations keep track of the queens
- Need an efficient way to compute which squares are free to place the next queen
- n x n `attack` grid
  - `attack[i][j] == 1` if `(i,j)` is attacked by a queen
  - `attack[i][j] == 0` if `(i,j)` is currently available
- How do we undo the effect of placing a queen?
  - Which `attack[i][j]` should be reset to `0`?

So, we had two representations, a two dimensional representation with 0s and ones and a one dimensional representation which gives us the column position for each row to keep track of the queens in the board, but in order to determine whether a square is free or not, we need to have a better way to compute how the squares are attacked by queens.

A simple way would be to just say that along with a two dimensional representation of the board we denote like we are done pictorially in the example we worked out we denote by what we have called this kind of colored square whether or not an attack a square is attacked. So, we say attack i j is 1, if it is attacked by queen otherwise it is 0. Now the problem with this is that a given square i j could be attacked by more than one queen right. So, when we undo a queen it will obviously, attack many squares, but not all those squares become free by removing that queen because, some of the squares are also attacked by other queens which we had placed earlier.

So, we need to be careful, when we remove a queen in order to mark squares which were attacked as being free. Well, one way to do this is to actually, number the queens and record the earliest queen that attacks each square.

(Refer Slide Time: 15:28)



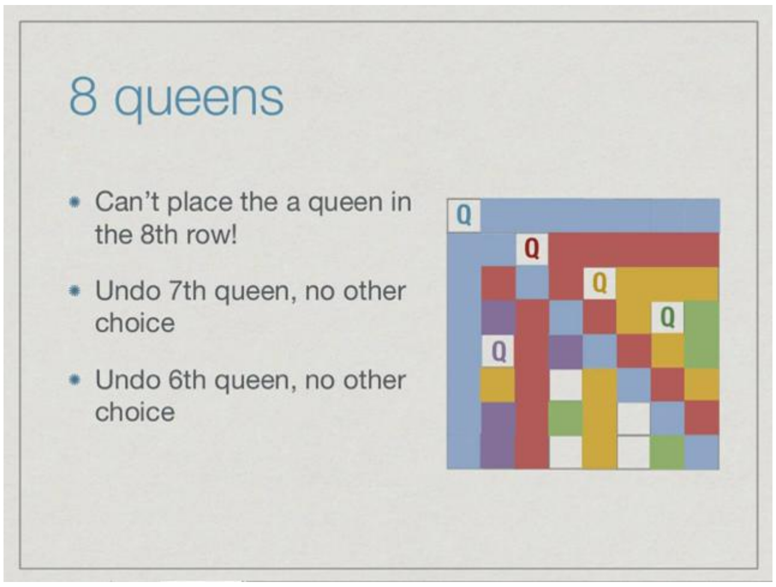
Updating the board

- Queens are added row by row
- Number the queens 0 to n-1
- Record earliest queen that attacks each square
  - `attack[i][j] == k` if `(i,j)` was first attacked by queen `k`
  - `attack[i][j] == -1` if `(i,j)` is free
- Remove queen `k` — reset `attack[i][j] == k` to `-1`
  - All other squares still attacked by earlier queens

So, we say attack i j is k if i j was first attacked by queen k and attack i j is minus one if i j is free. So, when we remove queen k we reset attack i j with value k to minus 1 and all other squares are still attacked by earlier queens.

So, we can explain this very easily with the picture that we had before.

(Refer Slide Time: 15:58)



8 queens

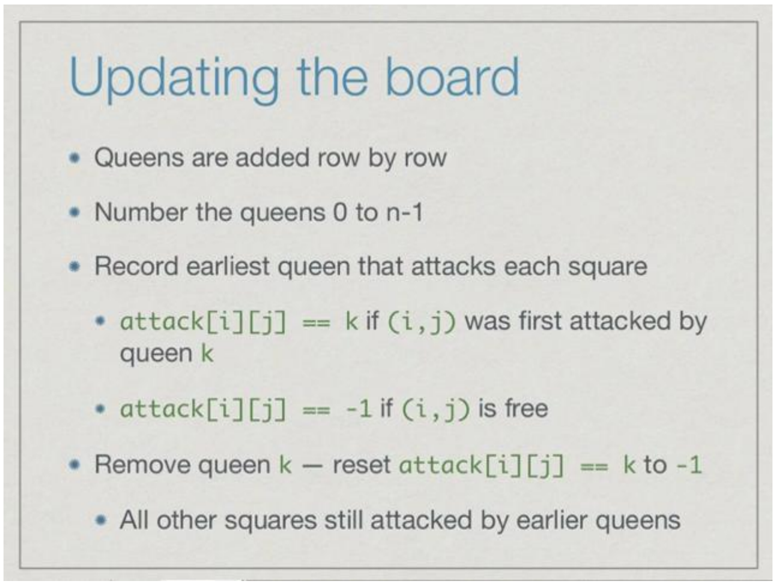
- Can't place the a queen in the 8th row!
- Undo 7th queen, no other choice
- Undo 6th queen, no other choice

Here is how we had represented our board when we put the blue queen we marked all squares of the blue queen attacked with blue as blue solid squares then, when we put the red queen we only attack when we mark with red those squares new squares which are attacked for example, this particular square, which is attacked by both red and blue was already attacked by blue. So, we did not mark it. So, in this way with each new queen i that we put we only mark the squares which are attacked by queen i.

The colors here represent the queen numbers. The blue squares are queen 0, the red squares are queen one, the yellow squares are queen two and so on. So, when it comes to undoing it for a instance, now we want to undo this particular thing now this when we put it had only one white square, there was no free squares other than this. So, we did not add any new attack. So, removing it does not actually change anything regarding the attack position only makes that particular square itself free does not unattack any of the other squares.

Now, when we remove this orange queen, then we have to remove all the orange squares which were placed under attack only after adding this queen and that turns out to be these two on the bottom row. So, when we undo this one, we will find those two get undone. Similarly when we undo the purple. So, what we are done actually was precisely this more efficient implementation of how to keep things how to record what is under attack.

(Refer Slide Time: 17:22)



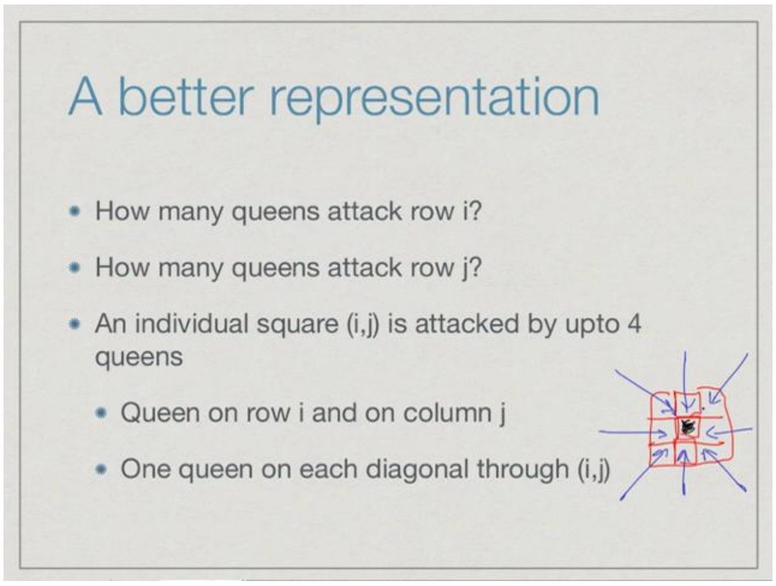
Updating the board

- Queens are added row by row
- Number the queens 0 to n-1
- Record earliest queen that attacks each square
  - `attack[i][j] == k` if `(i,j)` was first attacked by queen `k`
  - `attack[i][j] == -1` if `(i,j)` is free
- Remove queen `k` — reset `attack[i][j] == k` to `-1`
  - All other squares still attacked by earlier queens

So, we are going to now keep an attack array which says that attack i j is k, if it is first attacked by queen k and when we remove queen k we reset to minus one saying that, that square is free precisely if the value is currently k. Now this would work the only difficulty is that it requires N square space, we saw that we could replace the board by a linear thing from a N by N array with 0s and ones, we could replace it by a single array which had board i equal to be j.

The question is can we replace attack by a linear array now one thing to remember is that though attack itself is an N squared array attack, undoing the attack does not require as to actually look at all the N squared entries once we fix the queen to undo, we only have to look along it is row, column and diagonal and remove all entries with the value equal to that queen on that row column and diagonal. The updates are not a problem the updates are linear, adding and removing a queen only requires us to look at a linear number of cells in this array, but the array itself is quadratic, so can we improve our representation to use only order N space.

(Refer Slide Time: 18:35)



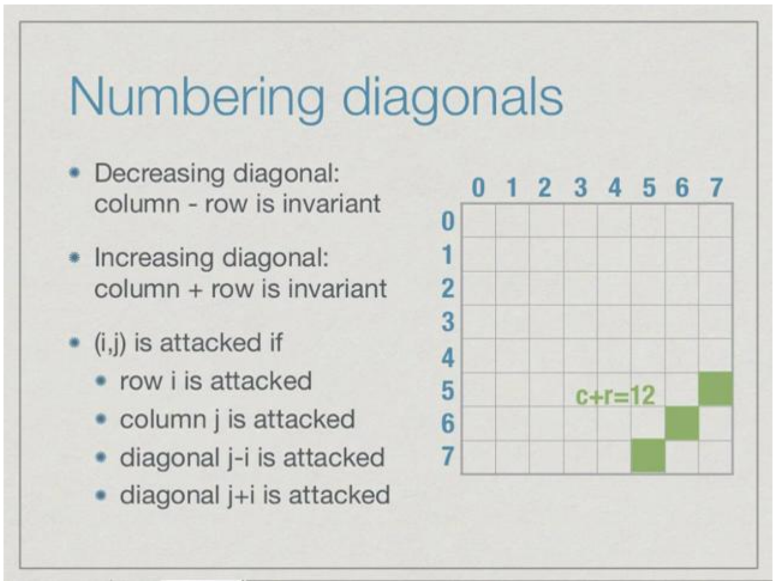
A better representation

- How many queens attack row i?
- How many queens attack row j?
- An individual square (i,j) is attacked by upto 4 queens
  - Queen on row i and on column j
  - One queen on each diagonal through (i,j)

To do this we just have to look a little closer at the problem. So, how many queens attack row i now if we look at the row as a whole remember we place only one queen in each row and in each column. So, only the queen on row i actually attacks row i similarly only one queen is in column j. Therefore, there is only the queen in column j which attacks that column. If we look at an individual square then, if we are in the center of this for instance then this particular square can be attacked from 4 directions, can be attacked from the column in which it is or the row in which it is or it can be attacked from this main diagonal or the off diagonal.

The main diagonal is the one from top left which is called north west and the one, the off diagonal is the one from the south west. There are 4 possible queens that could be attacking this square. There are 4 directions in which a square could be under attack. It might be better to represent these 4 directions rather than the squares itself the representation we have now is to say that this particular square is attacked by queen k, but it does not tell us from which direction queen k is attacking right it does not tell us whether queen k is attacking it from the row or the column or the diagonal.

(Refer Slide Time: 20:05)



Numbering diagonals

- Decreasing diagonal: column - row is invariant
- Increasing diagonal: column + row is invariant
- (i,j) is attacked if
  - row i is attacked
  - column j is attacked
  - diagonal j-i is attacked
  - diagonal j+i is attacked

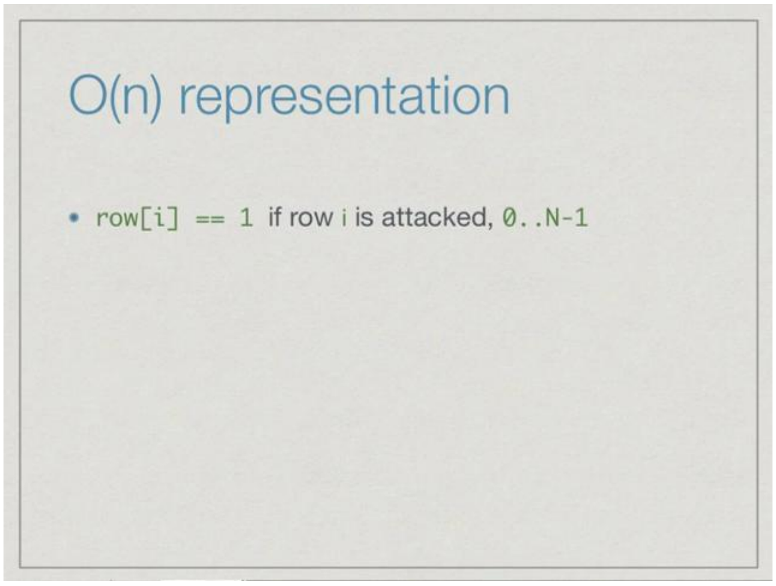
So, rows and columns are naturally numbered from 0 to 7, but how about diagonals. Now if we look at a diagonal from the north west. Let us call these directions north west, south west, north east and south east. If you look at a decrease in diagonal a diagonal that goes from top to bottom like this, then what we find as that this difference the column minus the row is something that will be the same along every square on that diagonal, for instance look at this diagonal it starts here.

Here the column number is 2 and the row is 0. 2 minus 0 is 2, if we go to the next item of the diagonal is 3 minus 1 which is again 2 then 4 minus 2 is again 2 and so on. So, if we go along this diagonal for all these squares, c minus r where c is the column number and r is a row number the difference is exactly 2 and you can check that nowhere else on the square on this grid is this true, as another example if you look at this particular thing. We have 0 minus four. The difference is minus 4 and similarly 3 minus 7 is also minus 4. So, everything along this particular diagonal has a difference minus 4.

Now, if we look at the diagonals going the other way then we find that the sum is an invariant here for instance we have either 6 plus 0 or 5 plus 1 or 4 plus 2 and 2 plus 3, 3 plus 3 and so on. So, along this purple diagonal c plus r is equal to 6 everywhere, and along this green diagonal we have 7 plus 5, 6 plus 6 and 5 plus 7. So, c plus r is equal to

12. So, we can now conclude that the square at position i j is attacked, if it is attacked by queen in row i or in column j or if it is along the diagonal whose difference is j minus i or if is along the diagonal whose difference is j plus i whose sum is j plus i.

(Refer Slide Time: 22:00)



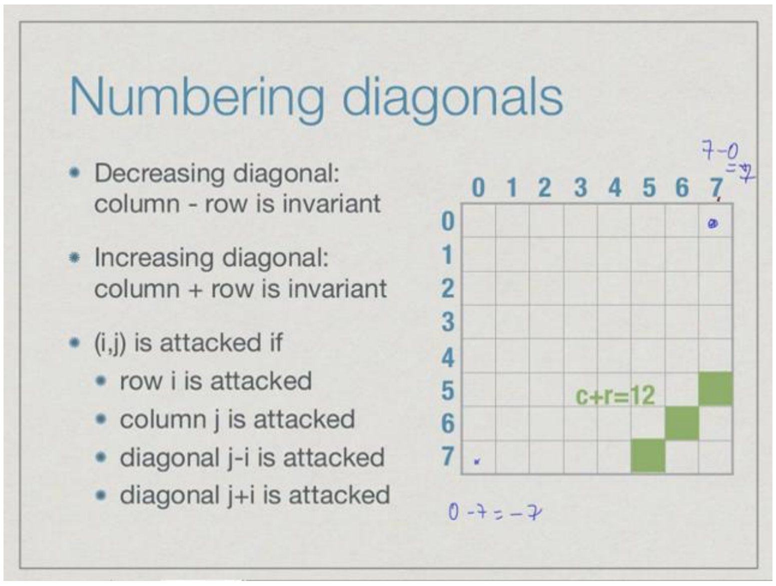
O(n) representation

- `row[i] == 1` if row i is attacked, `0..N-1`

So, we can now come up with a representation which only keeps track of rows, columns and diagonals which are under attack and from that we can deduce, whether a square is under attack. So, we say that row i is 1, if row i is under attack where i ranges from 0 to N minus 1 similarly; we can have a an array which says column i is attacked and then column i is set to 1 provided column i is attacked for again i between 0 and N minus 1. Now when we look at the diagonals we have these two types of diagonals.

The north west to south east diagonal is the one where the difference is the same and if you look at the differences, if you go back then you see the differences at this diagonal here, the difference is 7 minus 0 is 7 and here the difference is 0 minus 7 is minus 7.

(Refer Slide Time: 22:48)



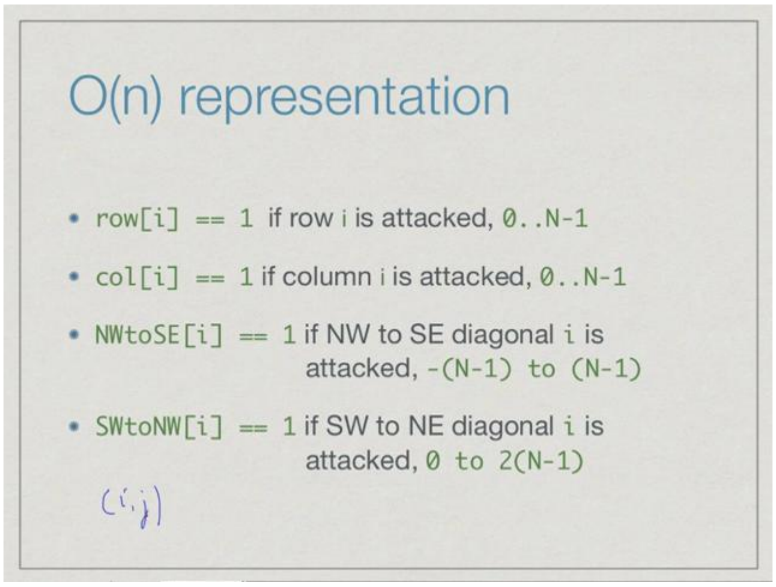
Numbering diagonals

- Decreasing diagonal: column - row is invariant
- Increasing diagonal: column + row is invariant
- (i,j) is attacked if
  - row i is attacked
  - column j is attacked
  - diagonal j-i is attacked
  - diagonal j+i is attacked

0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
c+r=12
7-0 =7
0-7=-7

It goes from plus N minus one to minus N minus 1.On the other hand, if you go the other way then the sum at this point is 0 plus 0 is 0, and the sum over here is 7 plus 7 is 14. The sum along these diagonals are 0, 1, 2, 3, 4 and so on. This is one this is 2 this is 3 and so on.

(Refer Slide Time: 23:11)



O(n) representation

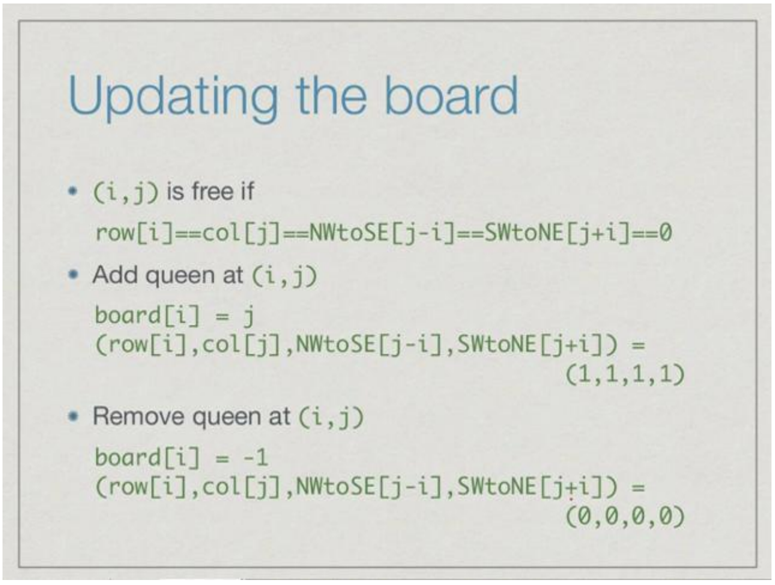
- `row[i] == 1` if row i is attacked, `0..N-1`
- `col[i] == 1` if column i is attacked, `0..N-1`
- `NWtoSE[i] == 1` if NW to SE diagonal i is attacked, `-(N-1) to (N-1)`
- `SWtoNW[i] == 1` if SW to NE diagonal i is attacked, `0 to 2(N-1)`

(i,j)

So, we have these north west to south east diagonals running from minus N minus 1 to N minus 1 this gives me the number if at. This is the difference if the difference is say 6 I know which squares are there if the difference is minus 3. I know which squares are there and the possible range of values is from minus 7 to plus 7 minus N minus 1 to plus N minus 1 and for the other direction it is from 0 to 2 times N minus 1 in our case two times N minus 1 is two times 7 which is 14.

So, 0 to 14, but if we have an N by N thing we have two times N minus 1. This gives us an order N representation of the squares under attack. Therefore, we look for if we want to see if i j squares under attack we check whether it is row i is one or column j is 1 or j minus 1, diagonal is 1 or i plus j diagonal is 1. If any of these is 1, then it is under attack if all of these are 0 then is not under attack right.

(Refer Slide Time: 24:07)



Updating the board

- (i,j) is free if

  `row[i]==col[j]==NWtoSE[j-i]==SWtoNE[j+i]==0`
- Add queen at (i,j)

```
board[i] = j
(row[i],col[j],NWtoSE[j-i],SWtoNE[j+i]) =
                                  (1,1,1,1)
```
- Remove queen at (i,j)

```
board[i] = -1
(row[i],col[j],NWtoSE[j-i],SWtoNE[j+i]) =
                                  (0,0,0,0)
```

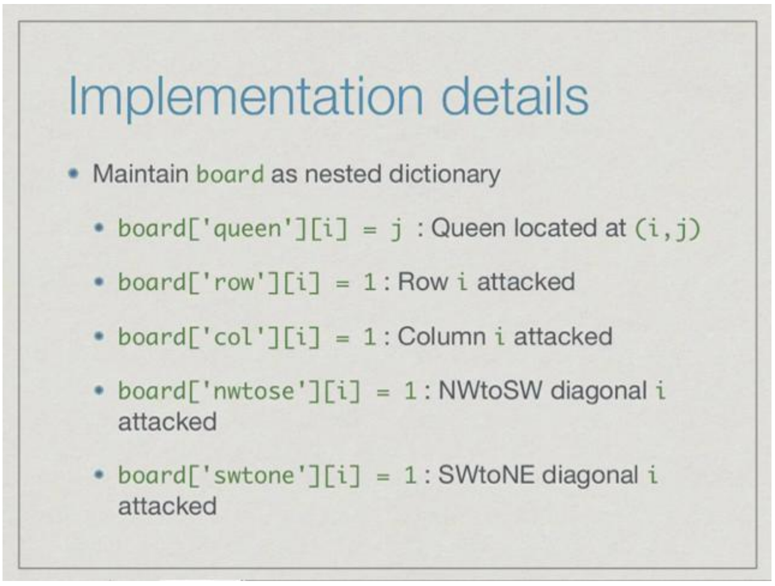
So, i j is free provided row i column j the north west to south east diagonal j minus i and the south west to north east diagonal j plus i are all equal to 0. When we add a queen at i j first we update the board representation to tell us that there is, now the ith row is set to the jth column and for the appropriate row, column and diagonal corresponding to this square we have to set all of them to be under attack.

So, row i becomes under attack, column j becomes under attack the j minus one th diagonal on the decreasing diagonal and j plus i th diagonal on the increasing diagonal all get set to one; And undo is similarly, easy we have to first reset the board value to say that the ith queen is not placed. So, we could say minus one this is not a valid value because the values are 0 to N minus 1. So, minus 1 indicates that the ith queen is not placed at this moment and we reset this row and this column to be equal to 0 because, this row and this column are attacked only by this queen.

Remember we cannot have two queens on the same diagonal because, they would attack each other. So, at any given point each one of these rows columns and diagonals is attacked by a single queen and it must be attacked by the queen at i comma j. So, only the queen at i comma j can attack all of these because, if it was under attacked by another queen we could not placed a queen at i comma j.

The fact that this free before indicates that all of these got attacked only by the current queen. So, when we remove the current queen we must reset them back to 0.

(Refer Slide Time: 25:35)



Implementation details

- Maintain board as nested dictionary
  - board['queen'][i] = j : Queen located at (i,j)
  - board['row'][i] = 1 : Row i attacked
  - board['col'][i] = 1 : Column i attacked
  - board['nwtose'][i] = 1 : NWtoSW diagonal i attacked
  - board['swtone'][i] = 1 : SWtoNE diagonal i attacked

One implementation detail for python is that instead of keeping these 5 different data structures, we have a board and a row and a column and all that we keep it as a single nested dictionary. So, it is convenient to call it board and we will have at the top 5 key values indicating the 5 sub dictionaries. The queen position we will call the key queen. So, instead of saying board i is j, we will say board with queen as the key at position i is j then we will say instead of row i is 1 or minus 1 we will say that the board at key row is one similarly board at key column board at north west to south east and board at south west to north east.

So, we have just converted it. So, we do not have to pass around 5 different parts to each function we just have to pass a single board which is a dictionary which contains everything of interest.

Remember that this is how we try to give our solution. So, we wanted to place the queen in row i and for each column that is available we would try to update the board and so on. Now, we have now better ways to do these things right. So, we have shown that using these dictionary or these 5 different representations we can check whether a row and column is available, how to update the board when we place a queen and we undo the queen.

Here we have an actual python implementation of what we discussed. So, we have this this function here which is called place queen. Place queen we said takes the row i and the board and the first thing it does it has to determine. What is the value of n? So, we just take.

Remember that board is now a dictionary. So, board of queen will tell us how many rows there are in the thing. If we take the length of the keys of board of queen we get n. This is just way of recovering N without passing it around. Now, what we do is for every possible value from 0 to N minus 1 that is for j, for all column values we check if i j is free in the current board. If it is free then we add a queen this is exactly the code that pseudo code we had if, i is N minus 1 we return true otherwise we try to extend the solution by placing a queen at i plus 1th row. If the solution does extend we return true otherwise we undo the queen.

So, undoing the queen will remove this queen and also update the board and finally, if this loop goes all the way through for every possible column and does not return true then we means it means we cannot place the queen on the ith row. So, we return false. Now the main function that we have the main code will start off by initializing board to be an empty dictionary, it will ask the user how many queens what kind of board we have N by N. So, remember we take the input it will be a string we convert it using int and we record this as N.

So, it asks for us number converts it to an int and passes it as N then we will initialize the board with the number N. We need N because, we need to know how to set up that remember that the indices run from 0 to N minus 1 or N minus N minus 1 plus N. N is required in order to initialize the dictionary and finally, we try to place the queen. So, initialization will setup an empty board where nothing is under attack then we try to place a queen in the 0th row on this board, if it succeeds then we have a function which prints the board.

Let us see how these other functions work. Let us first look at the function which initializes the board. Initializing the board says that first of all for every key for each of these sub dictionaries queen row column north west south east south west or north east we first set up a dictionary with that key. So, this says create an empty dictionary; Now for three of these things for queen, row and column right the indices are 0 to N minus 1. For i in range we just set up the key value i to point to minus 1 in case of queen this says that the queen in row i has not been placed and for row and column these are the attacked ones which says that they are 0 if they are under not under attack and one if they are under attack. The initial thing is to say 0.

Now, similarly for the north west to south east the range goes from minus N minus minus N minus 1 to plus N minus 1. So, from the range function since we give the upper bound as N. We set every key in this to 0 similarly for 0 to 2 N 2 into N minus 1 we want to set the south west to north east diagonals to be 0 this is one reason here why we are using a dictionary because for the other things of course, we could use a list right 0 to N minus one is the natural list index, but here we have the strange indices which go from minus N minus one to plus N minus 1 and so on.

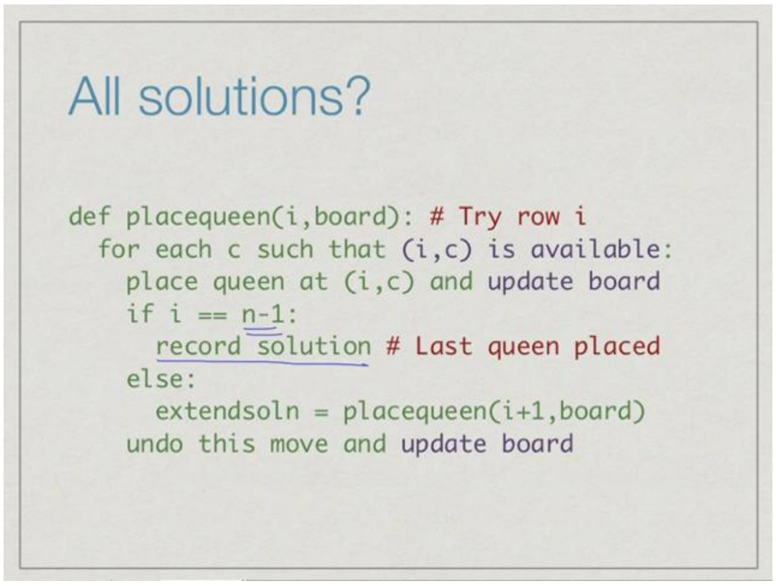
That is why we use a two level nested dictionary. This initializes the board; what how do we print a board well for every row we sort the rows. So, we take board dot queen dot keys will give us 0 1 upto N minus 1 in some random order we sort them and for each such row we print the row and the column number for that row. This happens when we have a successful solution. When is a position free well we check whether board the row entry is 0 the column entry is 0 the diagonal entry j minus i is 0 and the diagonal entry j plus i is 0 this is exactly as we said before and finally, what happens when we add a queen right when we add a queen we have to place it.

So, we set the queen entry for row i to j and then we mark the corresponding row column and diagonal to be one and when we undo a queen we set the queen entry to be minus 1 and the row column and diagonal entries to be 0; these are all exactly what we wrote in the pseudo code that has been formalized in python code. Now we can run this code and verify that it works. So, here we have this code 8 queens dot py which is the code that we just saw in the editor.

Now if I run this code as python 3.5 8 queens dot py. This is by the way if you have a python program you can run it directly without first invoking it and then importing it if you do this it will ask us how many queens we want. For instance if we give it the number 4 then we will get the solution that we saw in the earlier example it is not very printed out very neatly. So, if we give the number 8 then we will get one solution like this the it turns out that you can actually change that print board function i would not show you the code, but to print it out in a more user friendly way. So, I have another function which is called pretty if I do this then it shows me the 4 queen solution in a more readable form right.

So, you see exactly the kind of off diagonal positions and if I do for 8 queens then you see there is an extra column. There is some mistake in that, but there is an extra column, but basically you can see that if you ignore the last column which is showing the position of the queens in the first 8 queen solution. So, it is fairly straight forward once we have got the representation worked out and the structure of the code worked out, it is very easy to transform it into actual python code.

(Refer Slide Time: 32:54)



All solutions?

```
def placequeen(i,board): # Try row i
  for each c such that (i,c) is available:
    place queen at (i,c) and update board
    if i == n-1:
      record solution # Last queen placed
    else:
      extendsoln = placequeen(i+1,board)
    undo this move and update board
```

As a final step suppose we want not one solution, but all solutions right. So, we do not wants the previous thing in the moment it find a solution then it returns true and then every previous level also returns true and eventually it print out the board. Supposing we do not want to stop at the first solution, but keep printing out it is actually much easier; then what we do is we just keep going through all possible positions and whenever we reach the final step if we actually a solution reaches the final step then we record it in our case it might print it otherwise we extend it and go to the next one.

Actually it is much simpler to print all solutions than it is print a single solution because we do not have to remember whether our solution extends or not it is really running through every possible solution. The only thing is that it will not run through every solution to the very end and then decide it does not work. It is not like we are putting all possible queen positions and then trying it out we are trying it out for smaller things, because once we get stuck at say position 5 then it would not try to extend this it will come back and so on, but this is just a much simpler loop which just prints all solutions.

So, here is the code it's exactly the same code otherwise the only thing is the place queen function is much simpler now, we just try for every j and range one to 0 to N minus 1, if it is free we add the queen, if we have reached the last row we print the board, we extend the solution and then we undo the queen and try the next one.

For every j we are going to first add the queen, if it manages to place it extend the solution and finally, we are going to undo it and try the next j; we're just going to blindly try every possible j and we are not going to ever come out complaining that we have not succeeded the rest is pretty much the same, the print board has been changed slightly and slight change in the print board is just that we have changed it so that, it will print the entire thing on a single row. So, we have added this thing which says, end equal to space. So, we print the positions in a single row rather than row by row, that we can see them all.

Now if we look at the function now and we try to print it say for 4 queens, then it prints two solutions, these are essentially two rotated solutions of the same thing. If we do it for 8 queens for instance then it will actually produce a vast number of solutions it turns out there are actually 92 solutions, but even these 92 solutions if you look at rotations and reflections they come out to be much less, but if you just look at a position of the square as it is given to you then, there are 92 different solutions that it prints out. This concludes our discussion of backtracking with respect to the 8 queens' problem.