BackTracking LAB

By Ishaan Bhattacharya

INTRODUCTION:

The Problem was initially that there is a Chessboard of 8x8 in length and we are given 8 queens to place on the board such that no queen is attacking another queen. Queens attack each other if they can be reached diagonally or but shifting the queens left or right on the columns and rows. No queen can be on the same row or column together or be diagonal to another queen. The naïve solution to this would be to put a queen down or every space and see if any queens are under threat. Alternatively, one could just randomly place queens and see if any queens are attack any other and if there are any, randomly place queens again. The first naïve solution may work but will have an insanely large runtime O(N^N) and the second naïve solution has a possibility to never find the solution and run forever. This is where backtracking comes in. Backtracking makes decision and then check if future decisions are valid based on the initial decision. If they are then a solution is found, if they are not then we backtrack to the initial decision and make a different decision.

The subsequent Problems are Finding all solutions for an 8x8 board and then extrapolating that to find all solutions of a NXN board with N queens being available to place.

Method:

The method I used to improve on the Naïve solution was to prune out bad decisions early using the brute force algorithm called Backtracking. Backtracking takes in a decision and then checks if the current decision carried over to the end of the board creates a valid board. If it does not we recursively come back to the wrong decision and correct the state and then continue with a new decision. If they is a valid board at the end this is a solution and we immediately return this solution. Alternatively, If we are looking for multiple solutions we just continue until we find different end states that are valid and record them either by printing the board or incrementing a counter that represents Number of solutions.

Related problems that backtracking have helped with are Binario and Sudoku solving that we did in class. The theoretical benefits of this technique is that it prunes out some number of bad board states we no longer have to test before we even think about testing them. This should reduce the runtime of this algorithm compared to testing every cell for every permutation we go through. From pure brute force O(N^N) to theoretically O(2^N).

What I did was first create a print board function to just iterate through the NxN board and print the contents of the board. This was used to return the board state if a solution was found. It was also very helpful in checking what the board looked like if my recursion messed up.

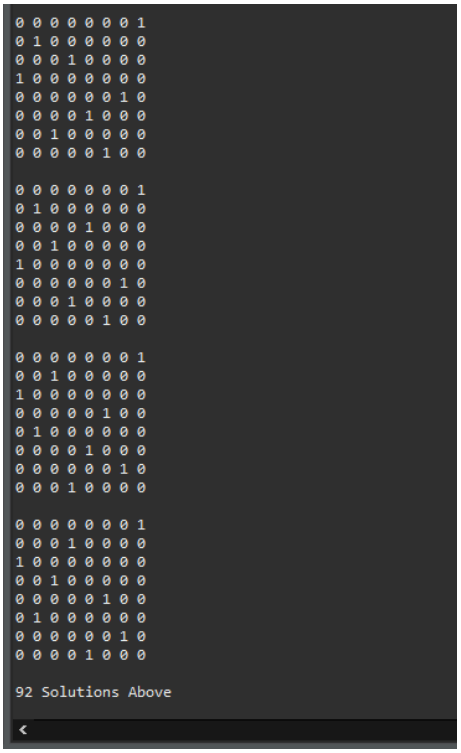
Then I created an isvalid/validity function to check if the current board state had no queen attacking another queen. First I had to check rows and columns. I created 2 counters for this iterating through the row and column of my current index looking for in another queen was subtly threatening my current location. If the counters were 1 or above (not counting my current index) I would return false and then my recursive function would have to backtrack. Next was diagonals, I found out that the absolute value of adding or subtracting my current row and column would be the same as the rows diagonal to the current index. I used this to check the board without going out of bounds. I used a counter to check for validity here too.

Next was the recursive function. In the recursive function I took in the Board, value of N (dimensions) and row and column. I would first do the base case of if the row index exceeded the value of N return. This means I was off the board and all states were valid. After that I created a look to loop through all column indices in a row to check if a queen is valid to insert. Then after an insertion I would validity check the board and if valid I would go to the next row and do the same thing. Until all rows had a queen that was not attacking other row's queens. Then I would be at row==n and I would return the current state of the board as the answer.

The recursive function for finding all the solutions on a board too much more brain power. If a valid state were found my board would just be returned and my function be halted. Now I had to change the function type from Boolean to void or perhaps I could have done int instead. I would now check if the state were valid first then not check the next value as well but immediately recursively call the next row down. Then instead of leaving out the backtracking I kept it in forcing all board states to be considered. If the Row == N I would return the board and increment a counter as a running tally of number of solutions I found. END I was finally done with my methods.

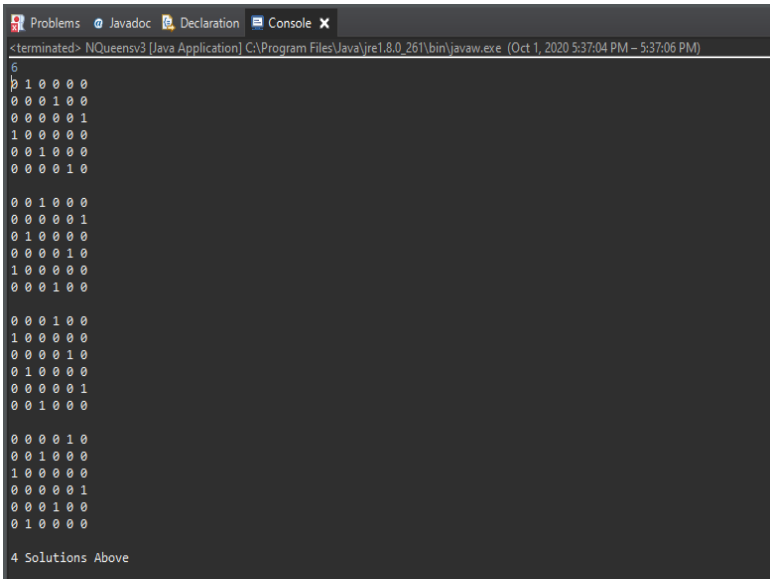
Results:

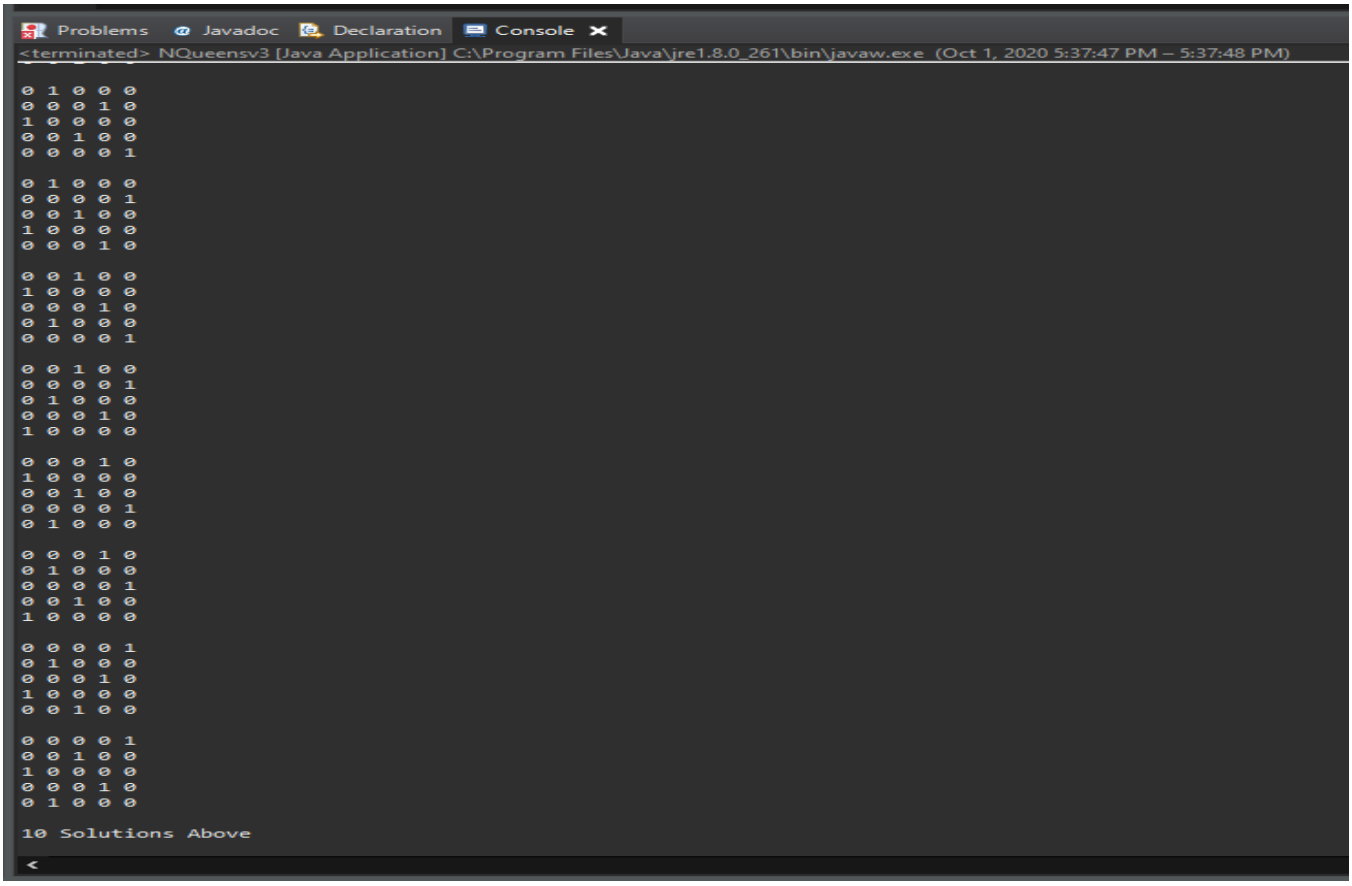
I will show my results just in the IDE I was working in which was eclipse. First was the N=8. Results in pictures below.



I just inputted 8 and it returned 92 boards all of which had no queens attack one another. Also at the end I printed a running tally of the number of solution this board had.

Something interesting I found was with the N=5 and N=6 solution shown below in pictures.





N at 5 had 6 more solutions than N at 6. This caused me some confusion and I was convinced my code was wrong. But after messing with some chessboards and queens and googling online I found out that this was the correct case. The problem for finding a solution to the board did not take very long to solve as I just had to check diagonals and rows and columns and backtrack. The finding all solutions for NXN and N queens took a very long time for me.(6 hours) As I would have to find a way to stop my code from terminate after any number of solutions were printed to console. But I pretty much just copied the Binario and Sudoku solvers u made in class for my validity check and recursive functions.

I could post all the solutions I got and their runtimes but I will be submitting the code and you can check to see if they are correct or at least mostly correct. I did this in java 1.8 on windows 10. I typed and ran it in eclipse just by clicking the run button at the top. I do not know how to give the jvm any memory restrictions. The Printing and tally for my code took a pretty decent amount of time (>5 minutes for the 15 = N case). But I was able to go to N=17 if did not print the board solutions as I went along. So 13 of the top N=30 I was unable to do in 5 minutes or less. The N queens for all others can be seen in my code which is commented to help in reading.

Conclusion:

The runtimes of my code were lightning fast until about N=11,12 where I saw some latency. In class u said java can run millions of operations a second(maybe) 100 million? But as my number of solutions reached the millions or hundreds of thousands and I was printing the board and doing the running tally of number of solutions my code became quite slow. This is mostly like not due to any of these wholly but contributing factors might be, for low runtime test cases jvm overhead might be an issue. For longer runtimes printing the boards to console took up some time. I might have been able to optimize my pruning or used another faster data structure to hold my contents like hashmaps or used hashfunction in some way like all elite coders do.

Some other puzzles that can be solved using this method are puzzles that I really like. Logic puzzles from red Baron. These puzzles have a number of open and filled in boards and then some logic-based hints on which item goes where in the puzzle. The constraints could be the validity check and the different number of monies, locations, people could be the options used in the decisions for the blank spaces. Also, in class u mentioned binario and sudoku as problems that could be solved.

With more time and resources I could most likely make the isvalid function in my code a little more efficient by taking out all the counters and using some Boolean value instead or something like that. I could also store the board in a more efficient data structure that a 2-d array. Instead of printing using I and j as counter in an array maybe I could use a hashmap for easy retrieval and printing.

Appendix:

Installation guide- I assume this for the ide I used. I used eclipse so just download java 1.8 as stated in the beginning of class and then download the eclipse ide and run it in there.

How to run without downloading and IDE- in terminal just do the following

1. go the directory u saved my .java file

2. do javac NQueensv3.java or alternatively javac NQueensv2.java

3. do java NQueensv3 or java NQueensv2 respectively.

What the format of the code is like. U input an integer N preferably 0-30(30 lol) and the program will take that in as N and return all the Boards that are solution to the NxN board with N queens. And after that a line will be printed saying the number of solutions.

Bonus- comment out 127 and 128 if u want only the number of solutions and not the board.