Solution: N-Queens

Let's solve the N-Queens problem using the Backtracking pattern.

We'll cover the following

Statement

Given a chessboard of size $n \times n$, determine how many ways n queens can be placed on the board, such that no two queens attack each other.

A queen can move horizontally, vertically, and diagonally on a chessboard. One queen can be attacked by another queen if both share t same row, column, or diagonal.

Constraints:

- $1 \leq n \leq 9$

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive solution

In order to find the optimal placement of the queens on a chessboard, we could find all configurations with all possible placements of n queens and then determine for every configuration if it is valid or not.

However, this would be very expensive, since there would be a very large number of possible placements and only a handful of valid ones. For example, when trying to place 6 queens on a 6×6 chessboard using brute force, suppose we place the first two queens side by side on the first two squares, there still remain $34 \times 33 \times 32 \times 31 = 1113024$ possible ways to place the remaining 4 queens. All of these ways are invalid since placing two queens next to each other is invalid. The time complexity of this solution would be $O(\binom{n^2}{n})$ and the space complexity would be $O(n)$, where n is the dimension of the chessboard.

Optimized solution using backtracking

A better and more optimized approach would be to only check for valid placements. This can be achieved by backtracking to a previously valid state in case there is no safe move left. Through backtracking, we avoid an exhaustive search and thus improve the algorithm's performance. Therefore, this problem is a great fit for the backtracking pattern.

Problem analysis

As stated in the problem statement, we have an $n \times n$ chessboard to place the queens such that no two queens attack each other. Let's understand the implications of these two conditions:

1. Once we place the first queen anywhere in the first row, no other queen may be placed in that row. Therefore, the search for a safe position for the next queen starts from the next row. This gives us a simple means to store a solution: we simply need to store the column for each row, that is, a list of n column indices, each representing a safe placement.
2. As there are n rows in the given board, and each row may safely hold just one queen, in any valid solution, all n rows would be used, each holding exactly one queen. This gives us both a condition to check the validity of a solution, as well as a way to check whether a solution is complete.

Solution 1

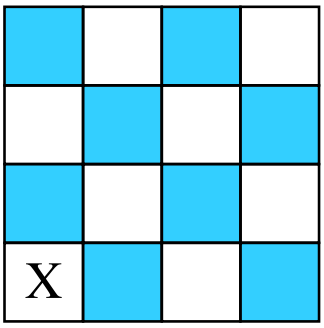
The backtracking algorithm to solve the $n-$queens problem is very similar to a depth-first search of a tree.

There are two conditions that cause us to backtrack, but for two different purposes:

- When we find that we cannot place the current queen in a particular row, we have to backtrack and alter the position of the queen whose position was decided before the current one. Next, we move forward again to find a safe position for the current queen.
- Once we find a valid solution, we still have to identify all the other valid solutions. So, we backtrack by removing the last queen placed on the board and resuming our search for solutions from that point. In order to be sure to find all possible solutions, we'll need to backtrack, row by row, all the way back to the first queen placed on the board, changing its position and then looking for alternative solutions.

Let's run our example on a 4×4 board with 4 queens:

We will consider the bottom row as the first row, and the left most column as the first column. We'll place the first queen in the first column of the first row.



1 of 12

> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step solution construction

The algorithm proceeds through the following steps:

1. Place a queen in the first column of the first row.
2. Now place a queen in the first such column of the second row where placement is permissible. This means that the current queen is not attacked by any queen already on the board.
3. If no such column is found, we'll backtrack to the previous row and try to place the queen in the next column of that row.
4. We continue this until we reach the last row of the board.
5. When we are able to successfully place the n^{th} queen in the n^{th} row we'll have found one valid solution.

6. After we find a solution, we backtrack to the previous row to find the next solution. Next, we try to find another column in the previous row where placement is permissible.

We'll start by declaring a `results` array that will store all possible solutions for n queens, and a `solution` array that will keep track of the current solution. We'll also implement a `isValidMove()` function that checks whether the desired move can place a queen at a safe position. A move is valid if the queen is not vulnerable to attack from other queens on the board.

Java

```
import java.util.*;

class NQueen {
        public static int tab = 2;
        public static String repeat(String str, int count) {
                String result = "";
                for (int i = 0; i<count; i++) {
                        result += str;
                }
                return result;
        }

        public static void printSolutiononBoard(int n, List<Integer>
                if (solution.size() != n) return;
                indent += tab;
                for (int i = 0; i<n; i++) {
                        if (i == 0) System.out.println(repeat(" ", in
                        System.out.print(repeat(" ", indent) + "|");
                        if (solution.get(n - (i + 1)) != -1) {
                                for (int j = 0; j<n; j++) {
                                        if (solution.get(n - (i + 1))
                                        else System.out.print("___|")
                                }
                        } else if (n - (i + 1) == proposedRow) {
                                for (int j = 0; j<n; j++) {
                                        if (proposedCol == j) System.
                                        else System.out.print("___
                                }
```

Check if a move is valid

Next, we'll call a recursive function called `solveNQueensRec()` to place the queens on the chessboard. We'll start by placing the queen in the first column of the first row. For every placement, we'll check if a move is valid. If yes, we'll add it to our solution.

Java

```
import java.util.*;

class NQueen {
        public static int tab = 2;
        public static String repeat(String str, int count) {
                String result = "";
                for (int i = 0; i<count; i++) {
                        result += str;
                }
                return result;
        }

        public static void printBoardState(int n, List<Integer> solut
                if (solution.size() != n) return;
                indent += tab;
                for (int i = 0; i<n; i++) {
                        if (i == 0) System.out.println(repeat(" ", in
                        System.out.print(repeat(" ", indent) + "|");
                        if (solution.get(n - (i + 1)) != -1) {
                                for (int j = 0; j<n; j++) {
                                        if (solution.get(n - (i + 1))
                                        else System.out.print("___|")
                                }
                        } else {
                                System.out.print(repeat("___|", n));
                        }
                        System.out.println();
                }
```

Placing the queens

Once our queen is placed at the correct position, we'll recursively check if another queen can be safely placed in the next row. If yes, we'll add the

position to our solution and move to the next queen. Else, we'll backtrack to the previous valid position and try a different configuration.

If we successfully reach the last row of the chessboard, we'll save that solution in the `results` array and backtrack to find the alternatives.

Java

```
import java.util.*;

class NQueen {
        public static int tab = 2;
        public static boolean printVar = true;
        public static String repeat(String str, int count) {
                String result = "";
                for (int i = 0; i<count; i++) {
                        result += str;
                }
                return result;
        }

        public static void printBoardState(int n, List<Integer> solut
                if (solution.size() != n) return;
                indent += tab;
                for (int i = 0; i<n; i++) {
                        if (i == 0) System.out.println(repeat(" ", in
                        System.out.print(repeat(" ", indent) + "|");
                        if (solution.get(n - (i + 1)) != -1) {
                                for (int j = 0; j<n; j++) {
                                        if (solution.get(n - (i + 1))
                                        else System.out.print("___|")
                                }
                        } else {
                                System.out.print(repeat("___|", n));
                        }
                        System.out.println();
```

N-Queens

Just the code

Here's the complete solution to this problem:

Java

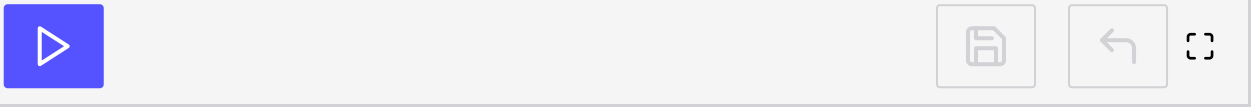
```
import java.util.*;

class NQueen {
        public static String repeat(String str, int count) {
                String result = "";
                for (int i = 0; i<count; i++) {
                        result += str;
                }
                return result;
        }

        public static void solveNQueensRec(int n, List<Integer> solut
                if (row == n) {
                        results.add(solution);
                        return;
                }

                for (int i = 0; i<n; i++) {
                        boolean valid = isValidMove(row, i, solution)
                        if (valid) {
                                solution.set(row, i);
                                solveNQueensRec(n, solution, row + 1,
                        }
                }
        }

        // Function to solve N-Queens problem
        public static int solveNQueens(int n) {
```



N-Queens

Solution summary

To recap, the solution to this problem can be divided into the following parts:

1. Place a queen in the first column of the first row.
2. Place a queen wherever permissible in the next row.
3. Backtrack if no safe configuration exists.
4. Once a solution is found, backtrack to find other possible configurations.

Time complexity

The recurrence relation for the time complexity of this solution is:

$$T(n) = nT(n-1) + O(n^2).$$

Deriving a precise upper bound for this recurrence relation is beyond the scope of this course. However, we can prove by induction that the time complexity, given this recurrence relation, is no worse than $O(n^n)$.

Space complexity

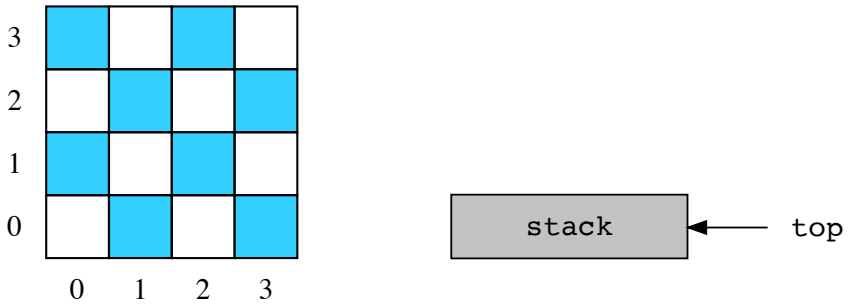
The space complexity of this solution is $O(n)$, where n is the dimension of the chessboard. This is because the maximum number of calls to the recursive worker function is n, one for each row, and each call takes up $O(1)$ space on the call stack.

Solution 2

We may also implement an iterative form of the backtracking search function, using a stack to keep track of the current solution. The stack holds only the column values and one solution is stored in the stack at a time. When one solution is complete, we save the solution present in the stack to a separate list of valid solutions. We then backtrack by popping from the stack and resuming the search for the next solution.

Let's run the above example using this solution and see how backtracking is achieved using the stack:

Initial state of the chessboard and stack.



1 of 28

Java

```
import java.util.*;

class NQueen {
        // This solution uses a stack to store the solution.
        // Stack will hold only the column values and one solution
        // will be stored in the stack at a time.
        static boolean isValidMove(int proposedRow, int proposedCol,
                // we need to check with all queens
                // in current solution
                int oldRow =0, oldCol = 0, diagonalOffset = 0;
                for (int i = 0; i<proposedRow; ++i) {
                        oldRow = i;
                        oldCol = solution.get(i);
                        diagonalOffset = proposedRow - oldRow;
                        // oldCol == proposedCol --> Checks if there
                        // oldCol == proposedCol - diagonalOffset -->
                        // oldCol == proposedCol + diagonalOffset -->
                        if (oldCol == proposedCol || oldCol == propos
                                return false;
                        }
                }
                return true;
        }

        // This solution uses stack to store the solution.
        // Stack will hold only the column values and one solution
        // will be stored in the stack at a time.
```

N-Queens

Solution summary

To recap, the solution to this problem can be divided into the following parts:

1. Place a queen in the first column of the first row.
2. Use a stack to keep track of the current solution.
3. Place a queen wherever permissible in the next row.
4. Backtrack by popping from the stack to find the next solution.

Time complexity

The time complexity of this solution is $O(n^n)$, where n is the dimension of the chessboard.

Space complexity

The space complexity of this solution is $O(n)$, where n is the dimension of the chessboard.

Back

Next