Lab Goals

- Implement an exhaustive search algorithm to solve the Eight Queens problem
- Upgrade your algorithm to handle more generic cases
- Answer some more questions relating to course content

Assignment Details

One topic that we likely will not have time to discuss in this class is the idea of Exhaustive Search. In exhaustive search, we find a solution to a problem by considering (possibly) all potential solutions, and selecting the correct solution from the list of all potential solutions. The run time of an exhaustive search problem is bounded by the number of possible solutions to the problem. That means that if the number of potential solutions is exponential, then the run time will also be exponential. Exhaustive search is often also commonly called Brute Force.

Exhaustive search algorithms can often be effectively implemented using recursion. Each recursive call progresses one node down the recursive tree. When a recursive call terminates, control reverts back to the calling parent, which then resumes execution. This allows us to easily implement Backtracking. In backtracking, an algorithm proceeds forward towards a solution until it becomes apparent that no solution can be achieved along the current path. At that point, we undo the current solution attempt (backtrack) to a point where we can again proceed forward.

In this lab, we will use exhaustive search to find solutions to the Eight Queens problem (defined in Part One). After discovering an algorithm to find one solution, the algorithm will be modified to display all solutions.

Part One: Implementing Basic Eight Queens (25 points)

The Queen is the most powerful piece on a chessboard. Queens have the ability to move horizontally in a row, vertically in a column, or diagonally across both a row and column. Queens can also move any number of spaces, ranging from one space outward to the edge of the board. This means that it is possible for a single Queen to “attack” as many as 27 of the 64 squares on a standard 8x8 chessboard.

The goal of the Eight Queens problem is to identify a way to position 8 different Queens on a single chessboard such that no Queen can directly attack another Queen. A simpler way to think
Here are some bad ways to solve Eight Queens: A naïve brute force algorithm will blindly place 8 Queens on a chessboard, then check to see if it is a valid solution. This blind placement of 8 Queens will consider all $64^8$ possible blind placements of 8 Queens, yielding 281,474,976,710,656 possible solutions. Clearly this is inefficient. (Wolfram Alpha notes that this is 14 times the number of red blood cells in the average human body.) We can greatly reduce this number of potential solutions by filtering all solutions that place multiple Queens on the same square of the board. This results in $\binom{64}{8}$, or 4,426,165,368 possible solutions. Better, but still a large search space. We can limit this solution space even further by including the restriction of a single Queen on every row. Now we have reduced our solution space to $8^8 = 16,777,216$ possible solutions.

This is where we will begin our recursive, backtracking exhaustive search. We can design a recursive function `placeQueen()` which will cycle across a row, trying to place a Queen in each column. If the placement is legal, then a recursive call is made to the next row. If the placement is not legal, then we iterate to the next column and attempt to place the Queen there. If we make it to the end of the row without finding a valid Queen placement, we backtrack out of the current recursive call and into the previous, to try to place the Queen in a different column in the previous row. If we find a valid location for Queen placement in the last row, then we have found a solution. A framework for the recursive function is below:

```java
placeQueen(int row) {
    for (int i = 0; i < numColumns; i++) {
        if (isLegalPlacement(board, row, i) {
            addQueen(board, row, i);
            if (row == numRows) {
                printSolution();
            } else {
                placeQueen(row+1); //if-else
            } //if
        } //if
    } //for
} //placeQueen
```

In this section, you will implement this recursive algorithm, and display the first solution that the algorithm generates.
Part Two: Implementing Advanced Eight Queens (15 points)

From here, you will make two small tweaks to your Eight Queens algorithm.

1. Rather than printing a single solution to the Eight Queens problem, you should modify your program to print all 92 of them. That is, rather than exiting after a solution is generated, you should remove the most recently placed Queen, and continue through the recursive calls until a second solution is found. Print that second solution, remove the most recently placed Queen, and continue the recursive backtracking process.

2. Modify your Eight Queens code to solve the N-Queens problem – print one (or all) solutions for placing \(N\) Queens on an \(N\times N\)-sized chessboard. This should hopefully be a simple code modification as well – simply alter the number of recursive steps and number of columns iterated across. You can verify that your code is working correctly by creating a counter recording the number of solutions found. The table at the top of the page lists the number of solutions for various board sizes.

Part Three: While You Have Some Downtime (10 points)

These Eight Queens runs should not take nearly as long to complete as the FourSum or SortCompare data collection runs. However, you have some additional questions to answer anyway, to test your knowledge of class content:

1. Trace the process of inserting the keys EIGHTQUEENS into a binary search symbol table. Each key is associated with the value corresponding to the index of the letter in the string. List the final set of Key-Value pairs after all letters are inserted.

2. Which symbol table implementation (Sequential or Binary Search) would you choose for an application that runs \(10^3\) put() operations and \(10^6\) get() operations?

3. Trace the process of inserting the keys SYMBOLTABLE into a binary search tree. Each key is associated with the value corresponding to the index of the letter in the string. List the final set of Key-Value pairs after all letters are inserted.
4. Our Binary Search Tree code from the book was presented using recursive operations for `get()` and `put()`. Is it possible to implement them with iterative operations? How would you do so?

**Submission Details**

For this lab, please submit a paper copy of everything listed below. Additionally, please upload all of your files to a folder in your BitBucket repository clearly labeled as “Lab 5.”

1. Your source code for `EightQueens.java` (or whatever you decide to name your file).

2. To show that you have completed Part One, show an output for the first solution to Eight Queens.

3. To show that you have completed Part Two, show the first five solutions for Eight Queens, and show the first solution for both Five-Queens and Eleven-Queens.

4. The answers to the questions in Part Three.

5. An Assignment Information Sheet filled out for your source file.

Please remember that all files that you submit should be your own work, though you are welcome to discuss high-level topics and algorithms with classmates.