N-Queens Problem

Mitch Maxfield, Koya Mitchell

Project ID AA_05_08

Contributions:

Mitch Maxfield: Everything that includes Branch & Bound

Koya Mitchell: Everything that includes Back Tracking

Table of Contents

Motivation:

The N queens' problem is described as a follow. Imagine a chess board of size n by n, how would one place an n number of queens on the board so that they are not attacking each other. In chess the queen is the most powerful piece on the board because it can attack infinitely far in each of the eight-direction surrounding it.

The N queens problem possess an interesting mathematical challenge. Primarily because of the sheer number of potential outcomes to sort through. Even on an 8x8 board there are 64 squares meaning there are a 64 choose 8 number of potential arrangements 8 queens can make on an 8x8 board. Which is roughly 4.4 billion though are only 40320 permutations. This is still a massive number of solutions to search through but is much smaller than 4.4 billion. on an 32x32 board the possibilism become a whooping 2.36x10^35. Still a very large number, it is very interested in discovering the best way to so, we thought it would be interesting to analyze an algorithm that could potentially sort through all these permutations and solve this problem.

Our motivation is to learn more about methods of backtracking and other more optimized solutions such as branch and bound. Understanding these methods will be the most interesting and benefit us both. Sometimes it may seem that a brute force approach is the only option, but that is rarely the case. implementing potential solutions such as a backtracking approach or branch and bound will show how us to better recognize how to improve and optimize brute force approaches.

Objective:

The objective of the project is to use the two algorithms, backtracking and branch and bound, to gauge which of the two is the more efficient one. Both in algorithm efficiency and execution run time. For the project there will be two different source codes implemented with each implementation using one of the two selected algorithms. In each implementation the code will allow the user to input the value of “n” queens and the code will make a matrix that is “n” x “n” large. For example, the user will enter 8 for the value of “n” queens. The code will then make an empty n x m matrix for the problem. Both source codes will print out a “n” x ”n” matrix with the solution and the time taken to execute the problem.

For the source code that utilizes the backtrack algorithm to place the queens in different columns one by one. When a queen is being placed in a column there will be a method that will check and see that if the path the queen could take would be in line with another already existing queen. If there are no existing queens on any of the paths of the queen being placed, then the coordinates of that location will be marked for the output.

For the source code that utilizes the brand and bound algorithm, like the backtrack algorithm, it will check every matrix coordinate to check for an existing queen. However, what the branch and bound algorithm will do differently is that it will value the best solution determined by the algorithm itself. The code will check the diagonals of where the queen is to find other placed queens to determine if that location can be occupied by a queen. There is the possibility of multiple different solutions using this algorithm so the output may be different with different run throughs of the code.

Problem Formulation :

The scale of the n queens problem grows exponentially. A common method for solving the n queens' problem is backtracking. Backtracking is a more intelligent approach to brute force; it works as follows. A state tree is created, the root is the initial state, and each level of nodes represents part of the solution. Each node is classified as either promising if it potentially leads to a solution or non-promising if it cannot be a correct solution. So, if a node is promising then the child is the next remaining option. If that next node is not promising, then it backtracks back up a layer and continues to explore other options until a solution is found.

So, in the n queens' problem. The root node is an empty board, the first node of layer 1 is a single queen at position (1,1), the first child from that node is then another queen placed at the next available safe space and so on. This continues until there are no safe spaces left for a queen to be placed, marking that node as non-promising then backtracking so other solutions can be explored. Or until an n number of queens have been placed thus finding a solution.

The second approach is a branch and bound search for a solution. A key difference between the backtracking and the branch and bound approach is that in backtracking only a single nxn array is used. In backtracking only, a single array is used to keep track of safe spaces the placement of additional queens. Just going through that array to check for a safe space and update the queen's position was already O(n^2). In Branch and bound we split that array into its components. The rules of the problem are that no queens can be the same row or column, no queens can share the same positive diagonal and no two queens can share the same negative diagonal. So, 3 Boolean arrays are created to ensure none of these rules are violated as well as 2 preprocessed matrices are created to assist in efficiently keeping track of which queens are placed What differentiates breach and bound from backtracking is that branch and bound it there is a bound on each node of the likelihood that it will lead to a solution and will keep track of the highest likelihood for a solution seen so far. If the current nodes bound value is lower than that of the highest, then that node is now non promising, and we move on to the next.

The hardware requirements are simple, any pc or laptop will complete our goals. For software any IDE that can work with the Java language will work well. To be specific we ran the tests on a 2015 MacBook Pro, and used the IDE Eclipse.

Back Track Implementation:

In the back tracking method the algorithm will check each position of the board and check if there is a queen placed above, below, and, diagonally across the board from the current position. If a queen is found then the algorithm will iterate one position to the right on the board and continue checking. If there was no queen detected from the current position of the board then the current position will be marked to indicate that a queen is placed there. The time complexity of this algorithm is O(N!) which is the same as the Branch & Bound algorithm. However, for Branch & Bound that is the worst case time complexity. So the average time complexity for Back Tracking is O(N!) while the average time complexity is O(N!)/2.

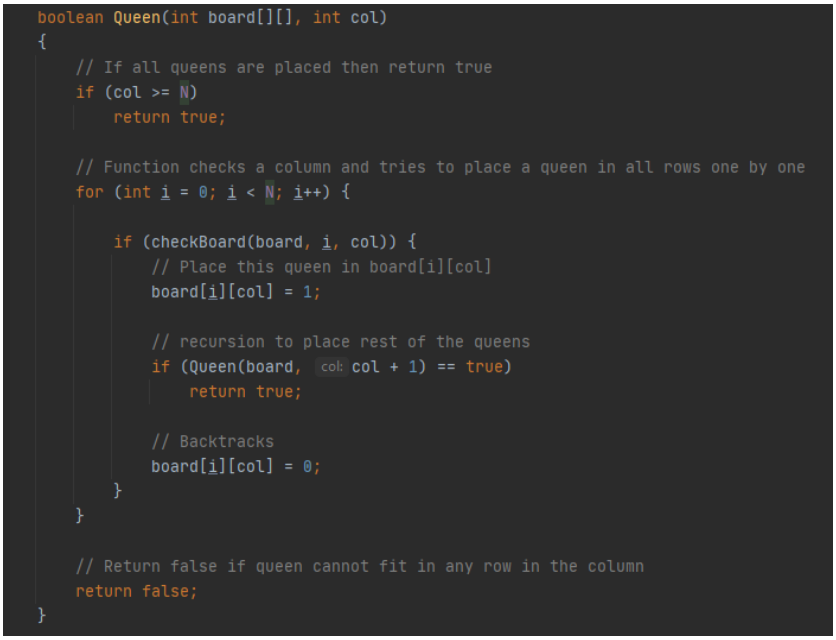
In the program the user is prompted to enter the amount of queens they want and the program will create a board with the size of [amount of queens] x [amount of queens]. Since the time complexity is O(N!) the more queens there are the more exponential the execution time will go up by for this algorithm. It is like this because every time a queen is placed the algorithm has to start from the very first position to check the available placement for the next queen.

The function below is used for checking if there is an existing queen from the current position on the board.

```
// Function to make sure there is an open spot for a queen
boolean checkBoard(int board[][], int row, int col)
{
    int i, j;

    // Check current row on left side
    for (i = 0; i < col; i++)
        if (board[row][i] == 1)
            return false;

    // Check the upper diagonal on left side
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j] == 1)
            return false;

    // Check the lower diagonal on left side
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j] == 1)
            return false;

    return true;
}
```

The function below is used to place the queens on the board. The time complexity is O(N!). in the for loop of the Queen method the range of the for loop is the amount of queens being placed. Inside

the loop the program will use the check board method to place a queen and if there are still room to shift to the right in the columns it will reclusively place the rest of the queens.



```
boolean Queen(int board[][], int col)
{
    // If all queens are placed then return true
    if (col >= N)
        return true;

    // Function checks a column and tries to place a queen in all rows one by one
    for (int i = 0; i < N; i++) {

        if (checkBoard(board, i, col)) {
            // Place this queen in board[i][col]
            board[i][col] = 1;

            // recursion to place rest of the queens
            if (Queen(board,  col: col + 1) == true)
                return true;

            // Backtracks
            board[i][col] = 0;
        }
    }

    // Return false if queen cannot fit in any row in the column
    return false;
}
```

Branch and Bound Implementation:

The backtracking approach to the N Queens Problem was in essence a brute force approach. Branch and Bound is a much more efficient method. Branch & Bound share the same worst-case efficiency as backtracking, O(n!). Although the average case for the branch bound solution is closer to (n!)/2. This is achieved by greatly reducing the cost of checking the board for an open space. In the backtracking solution the program will check each space in each queen's line of fire. In the branch & bound method 3 one dimensional Boolean arrays are mainted to efficiently keep track of whether a given space is safe or not. The arrays rowCheck, rightDiagnolCheck, and leftDiagnolCheck are used.

In the array rowCheck a row is safe and does not contain a queen if the value is false. The arrays rightDiagnolCheck, and leftDiagnolCheck like their name implies check if a given diagonal row is safe to use or not. Because these optimized arrays are all one dimensional a simple isSafe() check is at worst case O(1). This alone is one of the major factors in how much more efficient the branch and bound approach is than a backtracking solution. In the backtracking solution the checkBoard method has a worst-case efficiency of O(n).

The branch and bound technique is put simply an optimization of the backtracking technique. Four conditions must always be met for a solution to be feasible. No row can have more than 1 queen, no column can contain more than one queen and there can be no more than 1 queen on any forward or backward diagonal. Keeping track of rows and columns was straight forward. The area I had difficulty with was keeping track of which diagonals were in use. To solve this, I made two arrays where in which two queens sharing the same diagonal lane would have the same value in that array.

```
for (int r=0; r<n;r++) {
    for (int c=0; c<n;c++) {
        rightDiagnol[r][c]=r+c;
        leftDiagnol[r][c]=r-c+(n-1);
    }
}
```

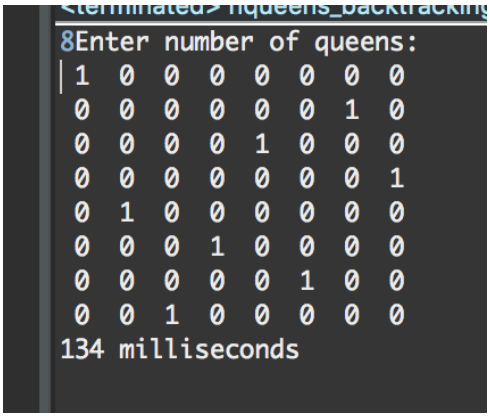
This is the code used to generate those diagonal matrixes. In addition, there are corresponding one dimensional Boolean look up arrays to efficiently keep track of whether a diagonal is used or not.

```
public static boolean solveNQueens(int[][] board, int col, int[][] rightdiagnol, int[][] leftDiagnol
                                   ,boolean[] rowCheck, boolean[] rightDiagnolCheck, boolean[] leftDiagnolCheck, int N) {

    if (col>=N) {
        return true;
    }

    for (int k=0; k<N;k++) {
         /* Check if queen can be placed on
           board[i][col] */
        if (isSafe(k, col, rightdiagnol, leftDiagnol,
                rowCheck, rightDiagnolCheck, leftDiagnolCheck)) {
            board[k][col] = 1;
            rowCheck[k]=true;
            rightDiagnolCheck[rightdiagnol[k][col]]=true;
            leftDiagnolCheck[leftDiagnol[k][col]]=true;
                if (solveNQueens(board, col+1, rightdiagnol, leftDiagnol,
                        rowCheck, rightDiagnolCheck, leftDiagnolCheck, N)) {
                    return true;
                }

        board[k][col]=0;
        rowCheck[k]=false;
        rightDiagnolCheck[rightdiagnol[k][col]]=false;
        leftDiagnolCheck[leftDiagnol[k][col]]=false;




        }
    }
    return false;
}
```

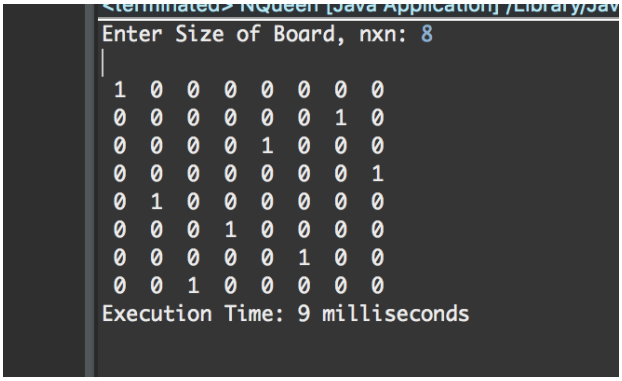
Above is the recursive solveNQueen method, it will place a queen row by row starting at row 0. First it will check is it safe queen placement. If it is then we update the board and all the check arrays then recursively call itself to delve further down this potential solution branch until a queen is placed in every column. If a placement is found to be not safe then the check arrays will be updated to reflect that, and we will test another location for the queen until all n queens are placed or until all options are exhausted and no solutions exist. Although a solution exists for every value of n except for 2, and 3.

Tests and Results:

Backtracking where n=8

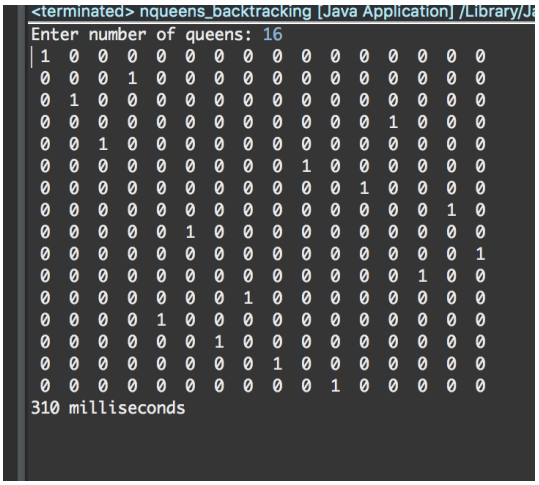


Branch & Bound where n=8

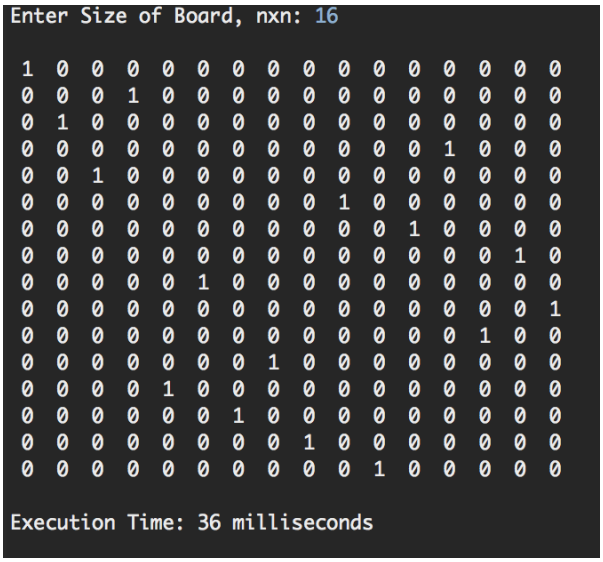


If we compare the execution run time Backtracking is 14.8 times slower than Branch & Bound.

Backtracking where n=16

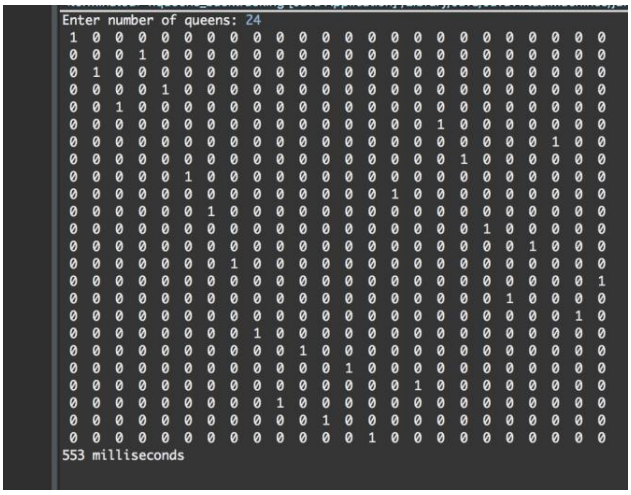


Branch & Bound where n =16

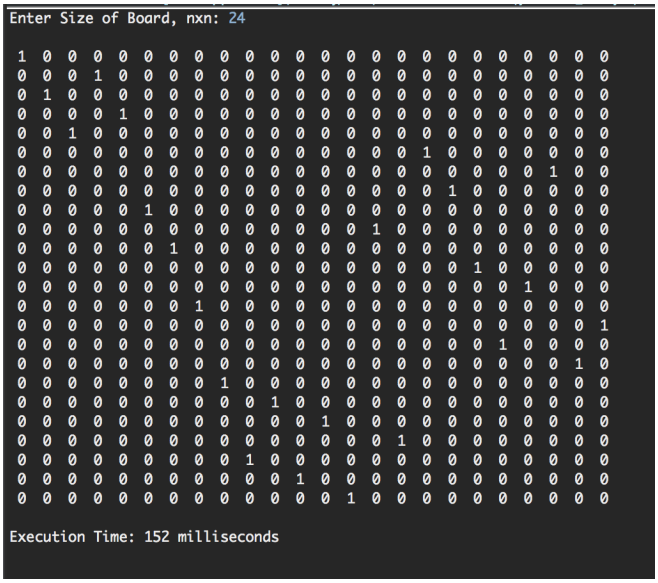


If we compare the execution run time Backtracking is 8.6 times slower than Branch & Bound.

Backtracking where n=24

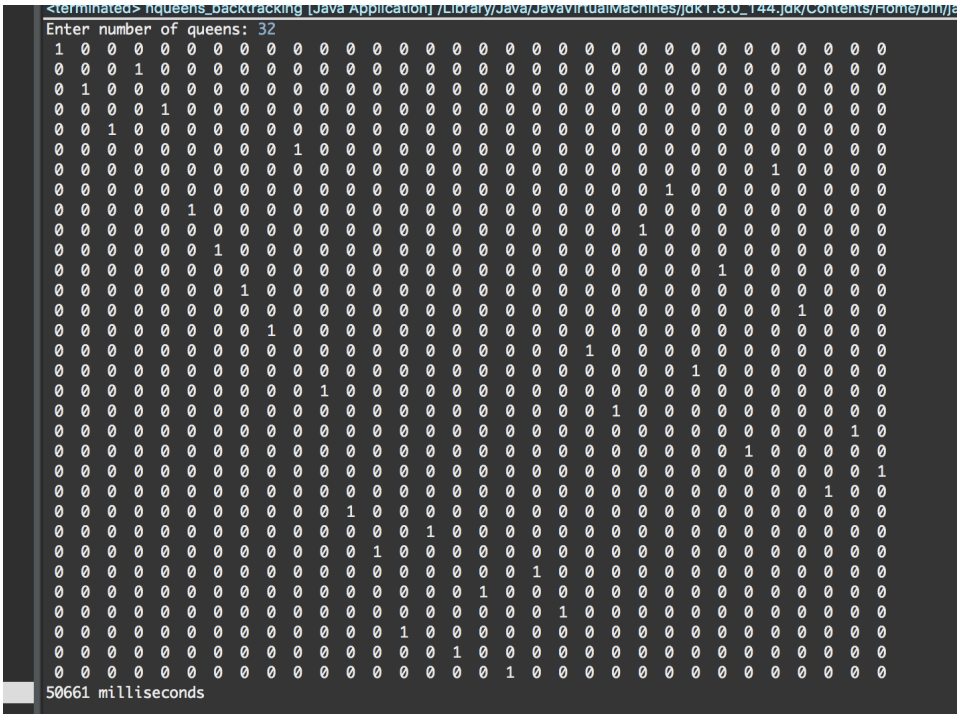


Branch and Bound where n=24

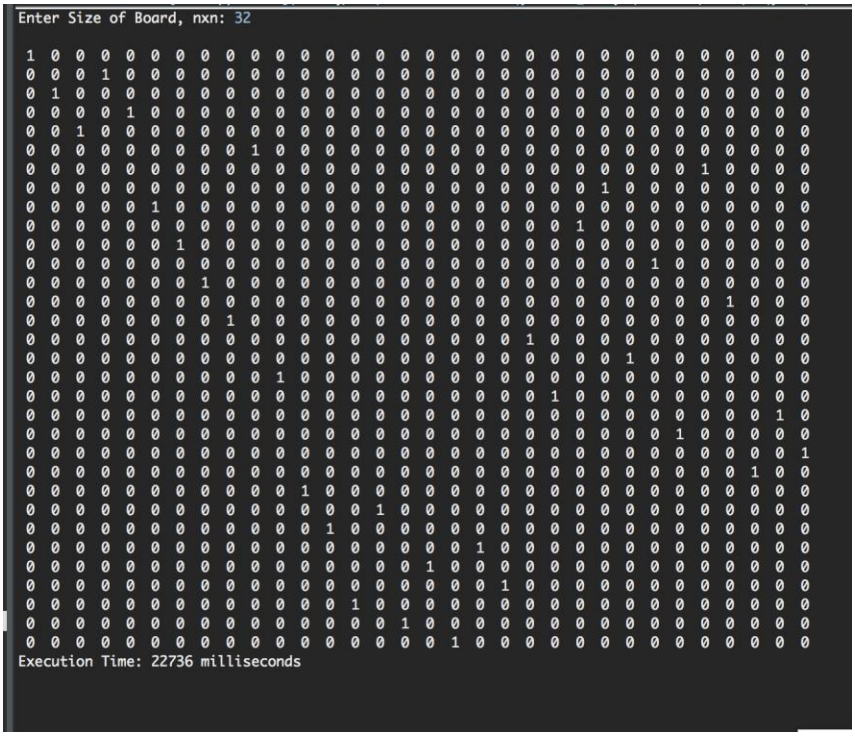


If we compare the execution run time Backtracking is 3.6 times slower than Branch & Bound.

Backtracking where n=32



Branch & Bound where n = 32



If we compare the execution run time Backtracking is 2.2 times slower than Branch & Bound.

Final Reflection:

After comparing the results of both algorithms. The final verdict that was reached is that much more Branch & Bound is more efficient than Back Tracking. Even though they have the same worst case time complexities of O(N!) the execution run time for Branch & Bound is significantly faster than Back Tracking. Based on our calculations the average case for branch and bound is closer to N!/2 while the average case for backtracking Is also n!. Which From here we can conclude that if the sample size is small then either algorithm can work since the difference in execution time is a matter of milliseconds. However, if the sample size is fairly large then the Branch & Bound algorithm is the preferred algorithm. If we take the execution run time of the biggest sample for the NQueens problem (which is 24) the Branch & Bound is 152 milliseconds and the Back Tracking execution run time is 553 milliseconds. That is about 3.6 times slower than the Branch & Bound algorithm. When we tested for a chess board of size 32x32, the branch and bound program found a solution in roughly 24 seconds. When I tested the backtracking program for 32 queens it took several minutes to solve the problem. In conclusion as the value of n increases the difference in execution time increases exponentially.

In completing this project, the importance to adding small and simple optimizations has become clear. In programs that iterate potentially thousands of times through thousands of permutations the small time saving optimizations add up far more than I originally anticipated. What one program did in 24 seconds another program built for the same problem couldn't find a solution in under 6 minutes. All because of a few optimizations and program made to run efficiently.

Works Cited

Levitin, A. (2012). *Introduction to the design & analysis of algorithms*. Boston, MA: Pearson Addison-Wesley.

Cull, P., & Pandey, R. (1994). Isomorphism and the N-Queens problem. *ACM SIGCSE Bulletin, 26*(3), 29-36. doi:10.1145/187387.187400

Source Codes

```java
import java.util.Scanner;

public class nqueens_backtracking {

    // Number of queens
    Scanner scan = new Scanner(System.in);
    final int N = scan.nextInt();

    // driver program
    public static void main(String args[])
    {

        System.out.print("Enter number of queens: ");
        long startTime = System.nanoTime();
        nqueens_backtracking Queen = new nqueens_backtracking();
        Queen.solveProblem();
        long endTime   = System.nanoTime();
        long totalTime = (endTime - startTime) / 1000000;
        System.out.println(totalTime + " milliseconds");
    }

    // Prints board
    void printSoln(int board[][])
    {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++)
                System.out.print(" " + board[i][j] + " ");
            System.out.println();
        }
    }
```

Backtracking Solution 1/3

```
// Function to make sure there is an open spot for a queen
boolean checkBoard(int board[][], int row, int col)
{
    int i, j;

    // Check current row on left side
    for (i = 0; i < col; i++)
        if (board[row][i] == 1)
            return false;

    // Check the upper diagonal on left side
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j] == 1)
            return false;

    // Check the lower diagonal on left side
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j] == 1)
            return false;

    return true;
}

boolean Queen(int board[][], int col)
{
    // If all queens are placed then return true
    if (col >= N)
        return true;

    // Function checks a column and tries to place a queen in all rows one by one
    for (int i = 0; i < N; i++) {

        if (checkBoard(board, i, col)) {
            // Place this queen in board[i][col]
            board[i][col] = 1;

            // recursion to place rest of the queens
            if (Queen(board, col + 1) == true)
                return true;

            // Backtracks
            board[i][col] = 0;
        }
    }

    // Return false if queen cannot fit in any row in the column
    return false;
}
```

Backtracking Solution 2/3

```
// Function to solve the n queens problem
 boolean solveProblem()
 {
     // board
     int board[][] = new int [N][N];

     if (Queen(board, 0) == false) {
         System.out.print("Solution does not exist");
         return false;
     }

     printSoln(board);
     return true;
 }
```

Backtracking Solution 3/3

```
public class NQueen {
    public static void main(String[] args) {
        //long startTime = System.currentTimeMillis();
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter Size of Board, nxn: ");
        int n = scan.nextInt();
        scan.close();
        //System.out.println("Board is Size: "+n);
        long startTime = System.currentTimeMillis();
        int[][] board = new int[n][n];
        int[][] rightDiagnol = new int[n][n];
        int[][] leftDiagnol = new int[n][n];
        boolean[] rowCheck = new boolean[n];
        //Arrays.fill(rowLookUp, false);

        boolean[] rightDiagnolCheck = new boolean[2*n-1];
        boolean[] leftDiagnolCheck = new boolean[2*n-1];

        /*generates the slash matrixes, every queen placed has a forward slash
        ...lane of attack, these matrixes are made in such a way so that any t
        will have the same value, this allows for a quick isSafe check*/
        for (int r=0; r<n;r++) {
            for (int c=0; c<n;c++) {
                rightDiagnol[r][c]=r+c;
                leftDiagnol[r][c]=r-c+(n-1);
            }
        }
        /*
        printSol(slash);
        System.out.println();
        printSol(backSlash);*/


        if (solveNQueens(board, 0, rightDiagnol,leftDiagnol,rowCheck,
                rightDiagnolCheck, leftDiagnolCheck,n)==false) {
            System.out.println("Solution does not exist");
        }

        printSolution(board);

        long endTime = System.currentTimeMillis();
        long timeElapsed = endTime - startTime;

       // printSolution(rightDiagnol);

        //printSolution(leftDiagnol);

        System.out.println();
        System.out.println("Execution Time: "+timeElapsed+" milliseconds");
    }
```

Branch & Bound Code (1/3)

```
public static void printSolution(int[][] board) {
    for (int i=0; i<board.length;i++) {
        System.out.println();
            for (int j=0;j<board.length;j++) {
                System.out.printf("%2d ", board[i][j]);
            }
    }
    System.out.println();

}

//checks if queen placement is safe, in O(1) using optimized boolean arrays
public static boolean isSafe(int row, int col, int[][] rightdiagnol, int[][] leftDiagnol,
        boolean[] rowCheck, boolean[] rightDiagnolCheck, boolean[] leftDiagnolCheck) {

    if(rightDiagnolCheck[rightdiagnol[row][col]] || leftDiagnolCheck[leftDiagnol[row][col]] || rowCheck[row]) {
        return false;
    }
    else
        return true;

}
```

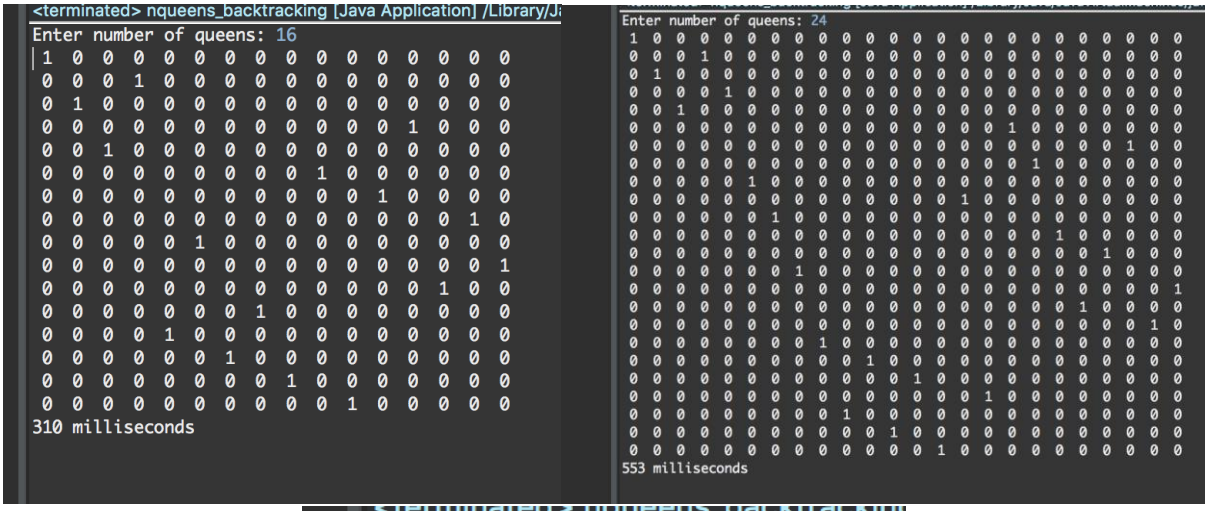
Branch & Bound Code (2/3)

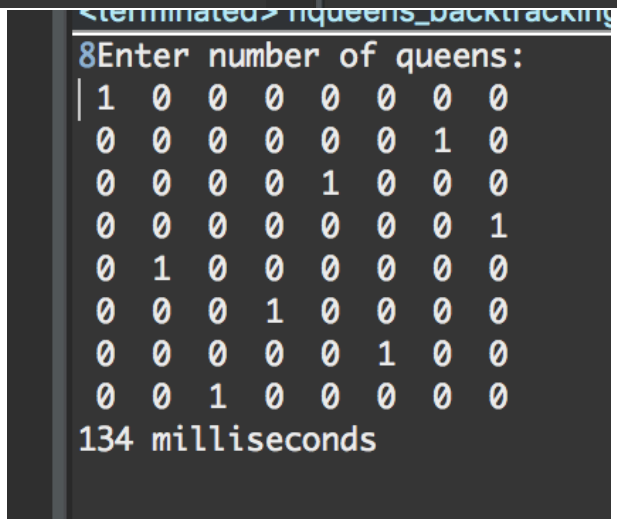
```
public static boolean solveNQueens(int[][] board, int col, int[][] rightdiagnol, int[][] leftDiagnol
                                   ,boolean[] rowCheck, boolean[] rightDiagnolCheck, boolean[] leftDiagnolCheck, int N) {

    if (col>=N) {
        return true;
    }

    for (int k=0; k<N;k++) {
         /* Check if queen can be placed on
           board[i][col] */
        if (isSafe(k, col, rightdiagnol, leftDiagnol,
                rowCheck, rightDiagnolCheck, leftDiagnolCheck)) {
            board[k][col] = 1;
            rowCheck[k]=true;
            rightDiagnolCheck[rightdiagnol[k][col]]=true;
            leftDiagnolCheck[leftDiagnol[k][col]]=true;
                if (solveNQueens(board, col+1, rightdiagnol, leftDiagnol,
                        rowCheck, rightDiagnolCheck, leftDiagnolCheck, N)) {
                    return true;
                }

        board[k][col]=0;
        rowCheck[k]=false;
        rightDiagnolCheck[rightdiagnol[k][col]]=false;
        leftDiagnolCheck[leftDiagnol[k][col]]=false;




        }
    }
    return false;
}
```

Branch & Bound Code (3/3)

Results and Outputs

Results For backtracking n queens' solution

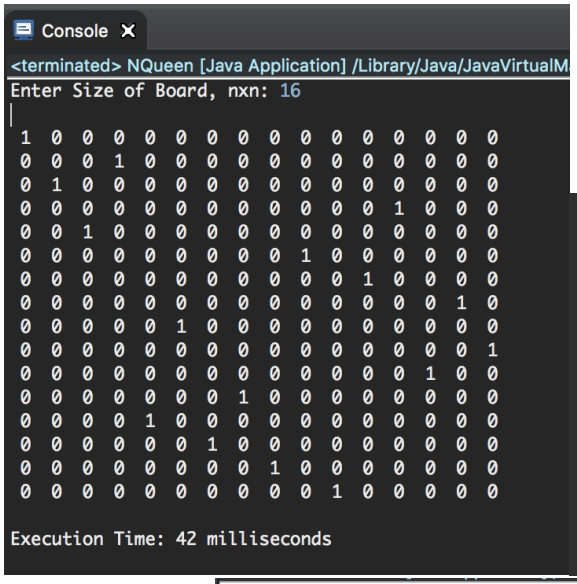


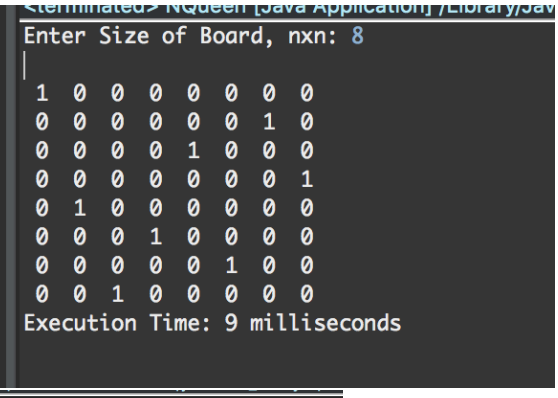


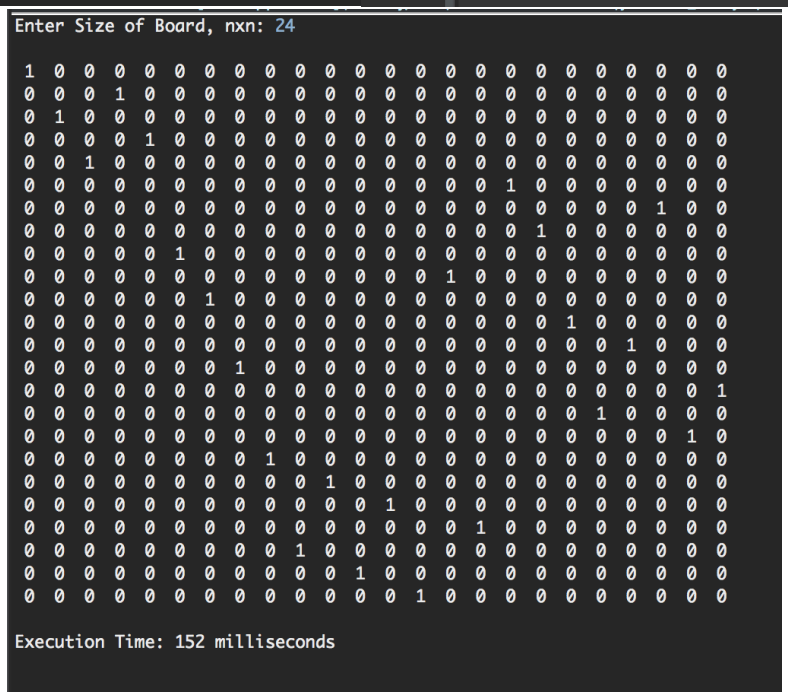
`

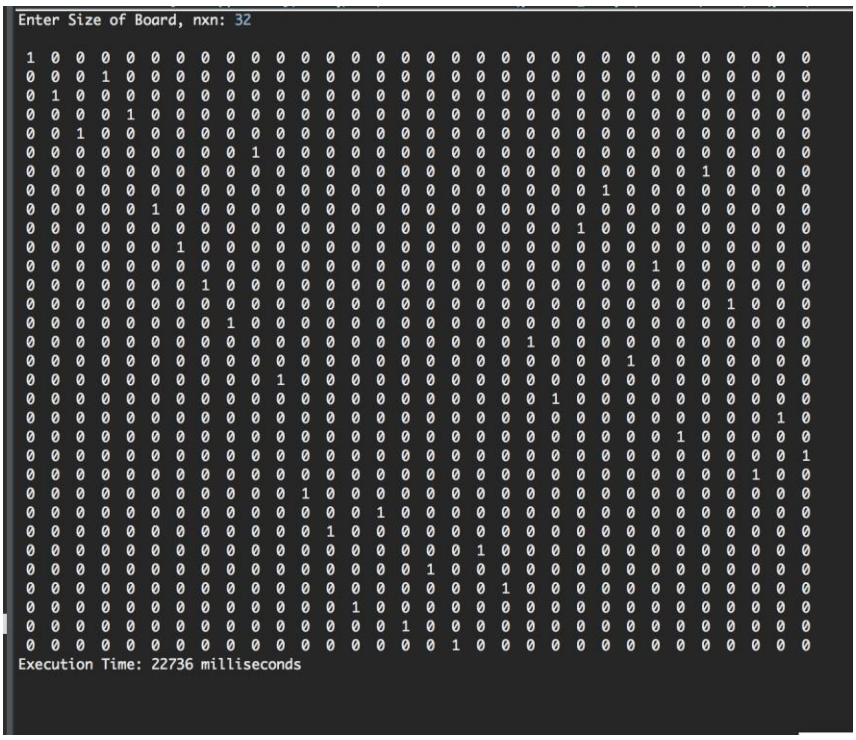

```
<terminated> nqueens_backtracking [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_144.jdk/Contents/Home/bin/ja
Enter number of queens: 32
 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
50661 milliseconds
```

Results and Outputs for Branch & Bound solution


Console
<terminated> NQueen [Java Application] /Library/Java/JavaVirtualM
Enter Size of Board, nxn: 16
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
Execution Time: 42 milliseconds


Enter Size of Board, nxn: 8
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
Execution Time: 9 milliseconds


Enter Size of Board, nxn: 24
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
Execution Time: 152 milliseconds


Enter Size of Board, nxn: 32
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
Execution Time: 22736 milliseconds