L P S - 7

{ Design and Analysis of Algorithms }

Rachit Bhalla
21BAI1869

1. Given the value of ‘n’ as input, write a recursive procedure to find all solutions to place them on a nxn board such that they do not attack each other.

Problem Statement: Design a recursive algorithm to find all solutions of n queens problem.
Logic: For each queen we place on the board, we check row , column and diagonal attacks to correctly determine the place for that queen. Hence we recursively call the nqueen function which uses backtracking logic to further correctly place our queens on the board.
Algorithm:
1.Import all the necessary libraries.
2. Create the 'not_attacked' function, which determines whether a queen placed in a particular 'row' and 'col' location on the 'board' will be attacked by any other queen already existing on the board. It looks for attacks in diagonals, the same row, and the same column.
3.Create the 'n_queens_recursive' function, which uses the recursive backtracking approach to find all N-Queens problem solutions. It requires the "board's" current setup, board size "n," and the "row" that is currently being processed.
4.If "row" equals "n," all rows have been examined, and a solution has been discovered.
5. If not, use the 'not_attacked' function to determine whether placing a queen at each column's location in the current row is valid. If it is valid, call 'n_queens_recursive' recursively for the next row i.e. ('row + 1'), marking that slot as occupied with a value of 1.

6.Set the value at ('row', 'col') back to 0 to remove the queen from the board after the recursive call. This is the retracing process, which enables the investigation of many combinations to uncover every solution.

7.Define the function 'n_queens' that acts as a wrapper. It calls "n_queens_recursive" with the board's initial configuration starting from "row = 0" and generates an empty "board" of size "nxn".
8. Return all the possible solutions of the given problem.

Pseudocode:

```
bool not_attacked(board,row,column)
{
    n = size of board;
    for j=0 to n{
        if (j != col && board[row][j] == 1)
            return false;}
        for i=0 to n {
        if (i != row && board[i][col] == 1)
            return false; }
    for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 1)
            return false;
    }

    for (int i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++) {
        if (board[i][j] == 1)
            return false;
    }
    return true;
}

n_queens_recursive(board,n,row)
{
    if (row == n) {
        print_board(board);
        return;
    }
```

```
    For col=0 to n {
        if (not_attacked(board, row, col)) {
            board[row][col] = 1;
            n_queens_recursive(board, n, row + 1);
            board[row][col] = 0;
        }
    }
}

n_queens(int n)
{
vector<vector<int>> board(n, vector<int>(n, 0));
 n_queens_recursive(board, n, 0);
}
```

Time Complexity: O(N!)

~> This program's time complexity is O(N!), where N is the chessboard's size. Because the second queen cannot be in the same row or column as the first queen, there are N-2 options for the second queen, N-4 for the third queen, and so on. This results in a total of about N!*(N-2)(N-4)...*1 possibilities.

Proof of Correctness:

Initialization:

It means showing that the loop invariant holds true prior to first iteration. With all entries set to 0, the board is first initialised as a 2D vector of size n × n. The initialised board, n, and row = 0 are sent as inputs to the n_queens function, which then invokes the n_queens_recursive function.

Maintenance:

The not_attacked function determines if a queen may be positioned on the board at a specific location without being attacked by any other queens. It determines if a queen is located in the same column, row, or diagonal as the specified place.

Each column in the current row is iterated through by the n_queens_recursive function, which then invokes not_attacked to determine whether a queen may be positioned there. If it can be

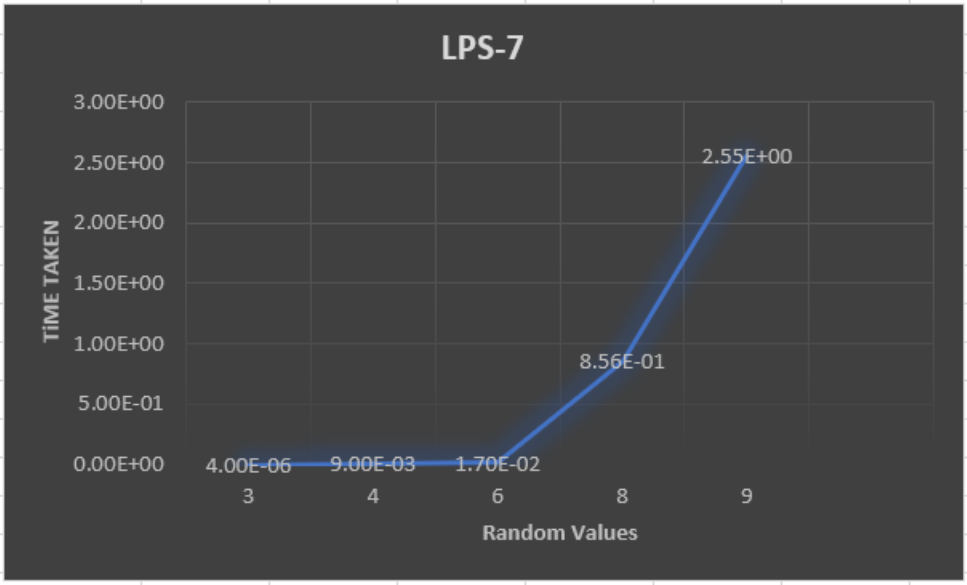
inserted, it does so and then repeatedly calls itself, increasing the row by 1. If it is unsuccessful in being inserted, it goes back by setting the position to 0 and attempting the following column.

Termination:

When row equals n, that is, when all n queens have been put on the board without engaging in mutual combat, the n_queens_recursive function comes to an end. Then, after printing the solution board, it makes another recursive call to the preceding one to look for further solutions. The algorithm comes to an end if there are no more solutions left to be discovered.

Graph:

Queens	Time Taken
3	4.00E-06
4	9.00E-03
6	1.70E-02
8	8.56E-01
9	2.55E+00



2. Given the value of ‘n’ as input, write a recursive procedure to find only one solution to place them on a nxn board such that they do not attack each other.

Problem Statement:

Design a recursive algorithm to find one solution of n queens problem.

Logic:

For each queen we place on the board, we check row , column and diagonal attacks to correctly determine the place for that queen. Hence we recursively call the nqueen function which uses backtracking logic to further correctly place our queens on the board.

Algorithm: 1.Import the necessary libraries

2. initialize empty nxn board.

3. Create a function `not_attacked(board, row, col)` that checks if a queen placed at position (row, col) on the board will not be attacked by any other queen.
   (a)Check if there is a queen in the same row.
   (b)Check if there is a queen in the same column.
   (c)Check if there is a queen on any of the diagonals.
4.Create a recursive function `n_queens_recursive(board, n, row)` that tries to place a queen on each column of the current row, starting from the leftmost column:
5.If `row` is equal to `n`, it means all queens have been placed successfully, so print the board as the solution.
 6.Iterate over each column in the current row.
 7.If placing a queen at position (row, col) is valid (i.e., it's not attacked by any other queen), mark that position on the board as 1.
 8.Call `n_queens_recursive(board, n, row + 1)` recursively to place the queens in the next row.
 9.If a solution is found (i.e., `n_queens_recursive` returns without further recursion), return from the current recursive call.
10. If no solution is found, backtrack by removing the queen from position (row, col) and try the next column.
11. Define a function `n_queens(n)` that initializes the board and starts the recursive backtracking process by calling `n_queens_recursive(board, n, 0)`.
12.Return the board with all the queens placed on it.

Pseudocode:

```
print_board(board)
{
    n = size of board;
    for (i=0 to n-1 {
        for (j=0 to n-1){
            print board
    }
}
bool not_attacked(board, row, col)
{
    n = size of board;
    for j=0 to n-1{
        if (j != col && board[row][j] == 1)
            return false;
```

```
    }
    for i=0 to n-1{
        if (i != row && board[i][col] == 1)
            return false;
    }
    for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 1)
            return false;
    }
    for (int i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++) {
        if (board[i][j] == 1)
            return false;
    }
    return true;
}
n_queens_recursive(board, n, row)
{
    if (row == n) {
        print_board(board);
        return;
    }

    for col=0 to n-1{
        if (not_attacked(board, row, col)) {
            board[row][col] = 1;
            n_queens_recursive(board, n, row + 1);
            board[row][col] = 0;
            return;
    }
}

n_queens(int n)
{
    Initialize board with all values 0
    n_queens_recursive(board, n, 0);
}
```

Time Complexity: O(N!)

~> This program's time complexity is O(N!), where N is the chessboard's size. Because the second queen cannot be in the same row or column as the first queen, there are N-2 options for the second queen, N-4 for the third queen, and so on. This results in a total of about N!*(N-2)(N-4)...*1 possibilities.

Proof of correctness:

Initialization:

We initialise the board vector to a size of n × n and fill it with zeros at the start of the n_queens function. Next, we invoke the n_queens_recursive function with the inputs board, n, and 0. This guarantees that all variables are appropriately initialised before we begin looking for answers.

Maintenance:

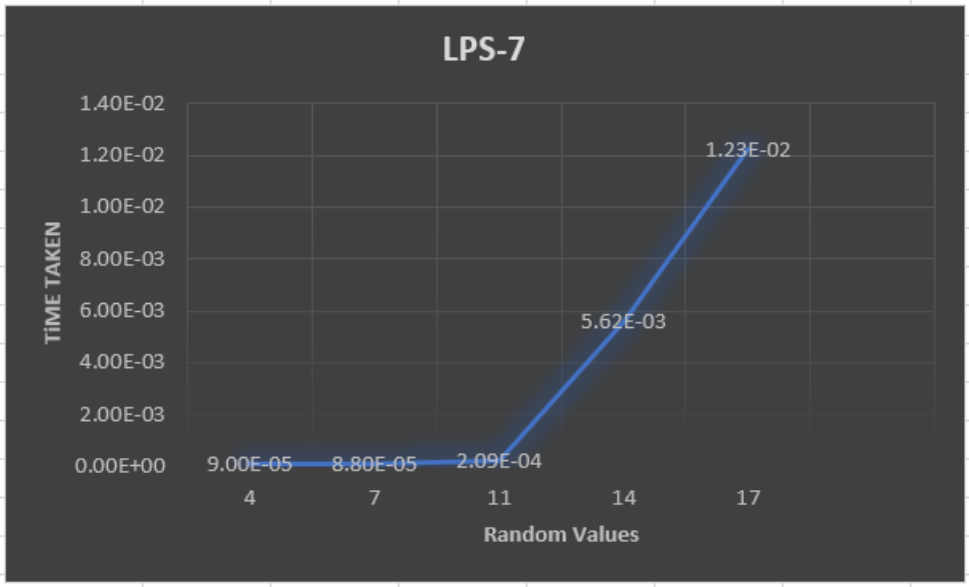
Queens are placed on the board using a backtracking strategy by the n_queens_recursive function. It tries to place a queen in each column for each row and uses the not_attacked function to see if any other queens on the board are attacking it. If it locates a secure location, it sets a queen there and calls itself repeatedly for the following row. When it gets to the final row, it prints the answer and then goes back. If it can't fit a queen in any column, it goes back to the row before and tries to fit the queen in the column after that. This procedure keeps on until either all problems are solved or no queen can be placed safely. This ensures that we maintain the correct state of variables throughout the function.

Termination:

If the n_queens_recursive function cannot fit a queen in any column in the current row or discovers all solutions, it stops. It prints the board once it has the answer and then goes back. If it can't fit a queen in any column, it goes back to the row before and tries to fit the queen in the column after that. When it reaches the first row and is unable to insert a queen into any column, it switches back to the n_queens function. This makes sure the function always exits and returns all legitimate answers.

Graph:

Queens	Time Taken
4	9.00E-05
7	8.80E-05
11	2.09E-04
14	5.62E-03
17	1.23E-02



3. Given the value of ‘n’ as input, write a recursive procedure with an external stack to find only one solution to place them on a nxn board such that they do not attack each other.

Problem Statement:
Design a recursive algorithm to find one solution of n queens problem with an external stack.

Logic:
For each queen we place on the board, we check row , column and diagonal attacks to correctly determine the place for that queen. Hence we recursively call the nqueen function which uses backtracking logic to further correctly place our queens on the board.But this time we take help of an external stack to maintain the position index of the queens.

Algorithm:
1.Import all the necessary libraries.
2. Initialize an empty nxn board.
3. create a function `print_board(board)` that prints the board configuration.
4. create a function `not_attacked(board, row, col)` that checks if a queen placed at position (row, col) on the board will not be attacked by any other queen.
   ->Check if there is a queen in the same row.
   ->Check if there is a queen in the same column.
   ->Check if there is a queen on any of the diagonals.

5. Define a recursive function `n_queens(board, n, curr_row, pos)` that tries to place a queen in each column of the current row, starting from the leftmost column:
   -> If `curr_row` is equal to `n`, it means all queens have been placed successfully, so print the board as the solution and return.
   -> If the stack `pos` does not have an entry for the current row, set `i` to 0 (start from the first column).
   -> Otherwise, set `i` to the next column after the previous attempt in the stack and remove the queen from the previous column.
   -> Iterate over each column `i` from `i` to `n-1`:
    -> If the `i`th column is valid for the current row (not attacked by any other queen):
       -> Store `i` in the stack `pos` for the current row.
       -> Mark position (curr_row, i) on the board as 1.
       -> Call `n_queens(board, n, curr_row+1, pos)` recursively to place the queens in the next row.
       -> Return from the current recursive call.
6.If no valid column is found (queen cannot be placed in the current row), backtrack to the previous row by calling `n_queens(board, n, curr_row-1, pos)`.
7. In the `main` function, prompt the user to enter the value of `n`.
8.Create an empty stack `pos` to store the column positions.
9.Return the solution.

Pseudocode:

```
print_board(board) {
    n = size of board;
    for i=0 to n {
        for j=0 to n{
            cout << board[i][j] << " ";
        cout << endl;
    }
}

Bool not_attacked(board, row, col) {
    int n = board.size();
    for (j=0 to n) {
        if (j != col && board[row][j] == 1)
            return false;
```

```
    }
    for (i=0 to n) {
        if (i != row && board[i][col] == 1)
            return false;
    }
    for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 1)
            return false;
    }
    for (int i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++) {
        if (board[i][j] == 1)
            return false;
    }
    return true;
}

n_queens(board, n, curr_row, pos) {
    if (curr_row == n) {
        print_board(board);
        return;
    }
    start_col = 0;
    if (!pos.empty() && pos.size() > curr_row)
        start_col = pos.top() + 1;

    for col=start_col to n {
        if (not_attacked(board, curr_row, col)) {
            board[curr_row][col] = 1;
            pos.push(col);
            n_queens(board, n, curr_row + 1, pos);
            pos.pop();
            board[curr_row][col] = 0;
        }
    }
}

main() {
    int n;
    initialize board as zero;
```

```
    stack<int> pos;
    n_queens(board, n, 0, pos);
}
```

Time Complexity:O(N!)

~> This program's time complexity is O(N!), where N is the chessboard's size. Because the second queen cannot be in the same row or column as the first queen, there are N-2 options for the second queen, N-4 for the third queen, and so on. This results in a total of about N!*(N-2)(N-4)...*1 possibilities. Though, due to the pruning nature of the algorithm, where we backtrack as soon as we find a conflict, the actual number of recursive calls is significantly smaller than N!. It is bounded by the number of valid solutions, which is often much less than N!.

Proof of correctness:

Initialization:

We initialise the board vector to a size of n × n and fill it with zeros at the start of the n_queens function. To keep track of the column locations of the queens put in each row, we initialise an empty stack pos as well. This guarantees that all variables are appropriately initialised before we begin looking for answers.

Maintenance:

Queens are placed on the board by the n_queens function using a stack-based backtracking technique. If there isn't an entry for the current row in the stack, it begins for each row at the location of the next column in the stack. It tries to put a queen in each column and uses the not_attacked function to see if any other queens on the board are attacking that queen. If it discovers a secure position, it moves a queen there, pushes the column position to the stack, and calls itself repeatedly for the following row. When it gets to the final row, it prints the answer and then goes back. If it is unable to fit a queen into any column, it goes back to the previous row and pops the top entry out of the stack before attempting to fit the queen into the subsequent column. This procedure keeps on until either all problems are solved or no queen can be placed safely. This guarantees that we keep variables in the right state throughout the function.

Termination:

When a queen cannot be put in any column in the current row or when it discovers all solutions, the n_queens function stops working. It prints the board once it has the answer and then goes back. When a queen

cannot be placed in any column, the machine goes back to the previous row and pops the top item out of the stack before attempting to place the queen in the subsequent column. If it goes back to the first row and is unable to insert a queen into any column, it goes back to the main screen and ends the programme. This makes sure the function always exits and returns all legitimate answers.

Graph:

Queens	Time Taken
4	6.20E-05
5	2.76E-04
6	3.90E-04
7	2.03E-03
8	4.25E-03