Batch : T6

Practical No. 10

Title of Assignment : Backtracing\N Queens Problem

Student Name:Parshwa Herwade

Student PRN: 22510064

Problem Statement:

1) **Given an integer n, the task is to find all distinct solutions to the n-queens problem, where n queens are placed on an n×n chessboard such that no two queens can attack each other. Each solution is a unique configuration of n queens, represented as a permutation of [1,2,3,....,n]. The number at the ith position indicates the row of the queen in the ith column. For example, [3,1,4,2] shows one such layout.**

   Input: 4
   Output: [2, 4, 1, 3], [3, 1, 4, 2]
   Explaination: These are the 2 possible solutions.

   Input: 1
   Output: [1]
   Explaination: Only one queen can be placed in the single cell available.

1. Algorithm/Pseudocode

Input: An integer n, the size of the chessboard and the number of queens to place.

Output: All distinct configurations where n queens are placed on an n x n chessboard without any attacking each other.

Steps:

1. **Initialization**: Create a list queens of size n where each index i represents the column number, and the value at each index represents the row number. Initially, no queens are placed.
2. **Backtracking**: We recursively attempt to place queens on the chessboard column by column.
3. **Check for Validity**:
   - Before placing a queen in a particular row and column, check if this placement is safe (i.e., it does not conflict with any previously placed queens).

- Conflicts can occur:
    - In the same row.
    - In the same column.
    - In the diagonals (both major and minor).

4. **Recursive Placement**: Place a queen in each column and move to the next column. If a valid solution is found, save it as a result.
5. **Backtrack**: If placing a queen in the current configuration leads to no solution, backtrack by removing the last queen and trying the next possible position.

Program Code

```
#include <iostream>

#include <vector>

#include <cmath>  // For abs()


using namespace std;


// Function to check if placing a queen at (row, col) is valid

bool is_valid(vector<int>& queens, int row, int col) {

  for (int i = 0; i < col; i++) {

    // Check for row and diagonal conflicts

    if (queens[i] == row || abs(queens[i] - row) == abs(i - col)) {

      return false;

    }

  }

  return true;

}


// Backtracking function to solve N-Queens

void place_queens(int col, vector<int>& queens, vector<vector<int>>& solutions, int n) {
```

```
    if (col == n) {
        // All queens have been placed, add the solution to results
        solutions.push_back(queens);  // Copy the current configuration
        return;
    }

    // Try placing a queen in every row for the current column
    for (int row = 0; row < n; row++) {
        if (is_valid(queens, row, col)) {
            queens[col] = row;  // Place the queen
            place_queens(col + 1, queens, solutions, n);  // Move to the next column
            // No need to explicitly backtrack because we overwrite queens[col] in the next
iteration
        }
    }
}

// Function to solve N-Queens and return all solutions
vector<vector<int>> solveNQueens(int n) {
    vector<vector<int>> solutions;  // To store all the valid solutions
    vector<int> queens(n, -1);  // To store the position of queens
    place_queens(0, queens, solutions, n);  // Start placing queens from column 0
    return solutions;
}

// Function to print the solutions in 1-based index format
void print_solutions(vector<vector<int>>& solutions) {
    for (const auto& sol : solutions) {
```

```
        for (int q : sol) {
            cout << q + 1 << " ";  // Convert 0-based to 1-based indexing
        }
        cout << endl;
    }
}

int main() {
    int n;
    cout << "Enter the value of N: ";
    cin >> n;

    vector<vector<int>> solutions = solveNQueens(n);
    cout << "Solutions for N = " << n << ":\n";
    print_solutions(solutions);

    return 0;
}
```

2. Output with verity of test cases

```
Enter the value of N: 5
Solutions for N = 5:
1 3 5 2 4
1 4 2 5 3
2 4 1 3 5
2 5 3 1 4
3 1 4 2 5
3 5 2 4 1
4 1 3 5 2
4 2 5 3 1
5 2 4 1 3
5 3 1 4 2
```

```
Enter the value of N: 4
Solutions for N = 4:
2 4 1 3
3 1 4 2
```

```
Enter the value of N: 1
Solutions for N = 1:
1
```

```
Enter the value of N: 8
Solutions for N = 8:
1 5 8 6 3 7 2 4
1 6 8 3 7 4 2 5
1 7 4 6 8 2 5 3
1 7 5 8 2 4 6 3
2 4 6 8 3 1 7 5
2 5 7 1 3 8 6 4
2 5 7 4 1 8 6 3
2 6 1 7 4 8 3 5
2 6 8 3 1 4 7 5
2 7 3 6 8 5 1 4
2 7 5 8 1 4 6 3
2 8 6 1 3 5 7 4
3 1 7 5 8 2 4 6
3 5 2 8 1 7 4 6
3 5 2 8 6 4 7 1
3 5 7 1 4 2 8 6
3 5 8 4 1 7 2 6
3 6 2 5 8 1 7 4
3 6 2 7 1 4 8 5
3 6 2 7 5 1 8 4
3 6 4 1 8 5 7 2
3 6 4 2 8 5 7 1
3 6 8 1 4 7 5 2
3 6 8 1 5 7 2 4
3 6 8 2 4 1 7 5
3 7 2 8 5 1 4 6
3 7 2 8 6 4 1 5
3 8 4 7 1 6 2 5
4 1 5 8 2 7 3 6
4 1 5 8 6 3 7 2
4 2 5 8 6 1 3 7
4 2 7 3 6 8 1 5
4 2 7 3 6 8 5 1
4 2 7 5 1 8 6 3
4 2 8 5 7 1 3 6
4 2 8 6 1 3 5 7
4 6 1 5 2 8 3 7
4 6 8 2 7 1 3 5
4 6 8 3 1 7 5 2
```

```
4 7 1 8 5 2 6 3
4 7 3 8 2 5 1 6
4 7 5 2 6 1 3 8
4 7 5 3 1 6 8 2
4 8 1 3 6 2 7 5
4 8 1 5 7 2 6 3
4 8 5 3 1 7 2 6
5 1 4 6 8 2 7 3
5 1 8 4 2 7 3 6
5 1 8 6 3 7 2 4
5 2 4 6 8 3 1 7
5 2 4 7 3 8 6 1
5 2 6 1 7 4 8 3
5 2 8 1 4 7 3 6
5 3 1 6 8 2 4 7
5 3 1 7 2 8 6 4
5 3 8 4 7 1 6 2
5 7 1 3 8 6 4 2
5 7 1 4 2 8 6 3
5 7 2 4 8 1 3 6
5 7 2 6 3 1 4 8
5 7 2 6 3 1 8 4
5 7 4 1 3 8 6 2
5 8 4 1 3 6 2 7
5 8 4 1 7 2 6 3
6 1 5 2 8 3 7 4
6 2 7 1 3 5 8 4
6 2 7 1 4 8 5 3
6 3 1 7 5 8 2 4
6 3 1 8 4 2 7 5
6 3 1 8 5 2 4 7
6 3 5 7 1 4 2 8
6 3 5 8 1 4 2 7
6 3 7 2 4 8 1 5
6 3 7 2 8 5 1 4
6 3 7 4 1 8 2 5
6 4 1 5 8 2 7 3
6 4 2 8 5 7 1 3
6 4 7 1 3 5 2 8
```

```
6 1 5 2 8 3 7 4
6 2 7 1 3 5 8 4
6 2 7 1 4 8 5 3
6 3 1 7 5 8 2 4
6 3 1 8 4 2 7 5
6 3 1 8 5 2 4 7
6 3 5 7 1 4 2 8
6 3 5 8 1 4 2 7
6 3 7 2 4 8 1 5
6 3 7 2 8 5 1 4
6 3 7 4 1 8 2 5
6 4 1 5 8 2 7 3
6 4 2 8 5 7 1 3
6 4 7 1 3 5 2 8
6 4 7 1 8 2 5 3
6 8 2 4 1 7 5 3
7 1 3 8 6 4 2 5
7 2 4 1 8 5 3 6
7 2 6 3 1 4 8 5
7 3 1 6 8 5 2 4
7 3 8 2 5 1 6 4
7 4 2 5 8 1 3 6
7 4 2 8 6 1 3 5
7 5 3 1 6 8 2 4
8 2 4 1 7 5 3 6
8 2 5 3 1 7 4 6
8 3 1 6 2 5 7 4
8 4 1 3 6 2 7 5
```

3. Analysis in terms of complexity wherever applicable.

Time Complexity:

- The time complexity is approximately **O(N!)**. For each column, we try placing the queen in every row, and this is done recursively.
- In practice, the pruning done by checking for conflicts reduces the actual number of recursive calls.

Space Complexity:

- The space complexity is **O(N)** for the recursion stack and the queens array.
- Additionally, the space required to store solutions can go up to **O(N! × N)** since each solution involves storing N integers.

Problem Statement:

2) Given the dimension of a chess board (N x M), determine the minimum number of queens required to cover all the squares of the board. A queen can attack any square along its row, column or diagonals.

```
Input : N = 8, M = 8
Output : 5
Layout : Q X X X X X X X
         X X Q X X X X X
         X X X X Q X X X
         X Q X X X X X X
         X X X Q X X X X
         X X X X X X X X
         X X X X X X X X
         X X X X X X X X

Input : N = 3, M = 5
Output : 2
Layout : Q X X X X
         X X X X X
         X X X Q X
```

Algorithm/Pseudocode

Step 1: Identify Special Cases

- For small grids or edge cases (like 1x1, 1xN, Nx1), a minimal number of queens is required, typically 1.

Step 2: Layout Queens for Larger Grids

- For an N×MN \times MN×M grid, queens are placed to maximize the coverage of rows, columns, and diagonals. The optimal number of queens for certain grids is determined by a greedy approach that places queens such that the board is fully covered while minimizing overlap.

Step 3: Recursive Placement (Optional)

- For large or irregular grids, a recursive backtracking solution can be used to try placing queens, backtracking if the placement does not fully cover the board.

Step 4: Verify Full Coverage

- After placing queens, check if every square is either occupied by a queen or can be attacked by a queen.

Program Code

#include <iostream>

```
#include <vector>
using namespace std;


// Function to check if a position (row, col) is attacked by any queen
bool isAttacked(vector<vector<char>>& board, int row, int col, int N, int M) {
  // Check the current row and column for any queen
  for (int i = 0; i < N; i++) {
    if (board[i][col] == 'Q') return true;
  }
  for (int j = 0; j < M; j++) {
    if (board[row][j] == 'Q') return true;
  }

  // Check diagonals
  for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
    if (board[i][j] == 'Q') return true;
  }
  for (int i = row, j = col; i >= 0 && j < M; i--, j++) {
    if (board[i][j] == 'Q') return true;
  }
  for (int i = row, j = col; i < N && j >= 0; i++, j--) {
    if (board[i][j] == 'Q') return true;
  }
  for (int i = row, j = col; i < N && j < M; i++, j++) {
    if (board[i][j] == 'Q') return true;
  }

  return false;
```

```
}

// Function to place the minimum number of queens on the board
void placeQueens(vector<vector<char>>& board, int N, int M) {
    int queensPlaced = 0;

    // Try placing queens in an alternating pattern
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            if (!isAttacked(board, i, j, N, M)) {
                board[i][j] = 'Q';
                queensPlaced++;
            }
        }
    }

    // Output the board
    cout << "Minimum number of queens required: " << queensPlaced << endl;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
}

int main() {
    int N, M;
```

```
    cout << "Enter dimensions of the chessboard (N M): ";
    cin >> N >> M;

    vector<vector<char>> board(N, vector<char>(M, 'X'));

    // Place queens and print the layout
    placeQueens(board, N, M);

    return 0;
}
```

Output with verity of test cases

```
Enter dimensions of the chessboard (N M): 8 8
Minimum number of queens required: 5
Q X X X X X X X
X X Q X X X X X
X X X X Q X X X
X Q X X X X X X
X X X Q X X X X
X X X X X X X X
X X X X X X X X
X X X X X X X X
```

```
Enter dimensions of the chessboard (N M): 3 5
Minimum number of queens required: 3
Q X X X X
X X Q X X
X X X X Q
```

Analysis in terms of complexity wherever applicable.

Time Complexity:

- **isAttacked()** runs in O(N+M) time, as it checks the row, column, and diagonals for a queen.
- **placeQueens()** iterates through each cell of the board and calls isAttacked() for each, resulting in a time complexity of O(N×M×(N+M))

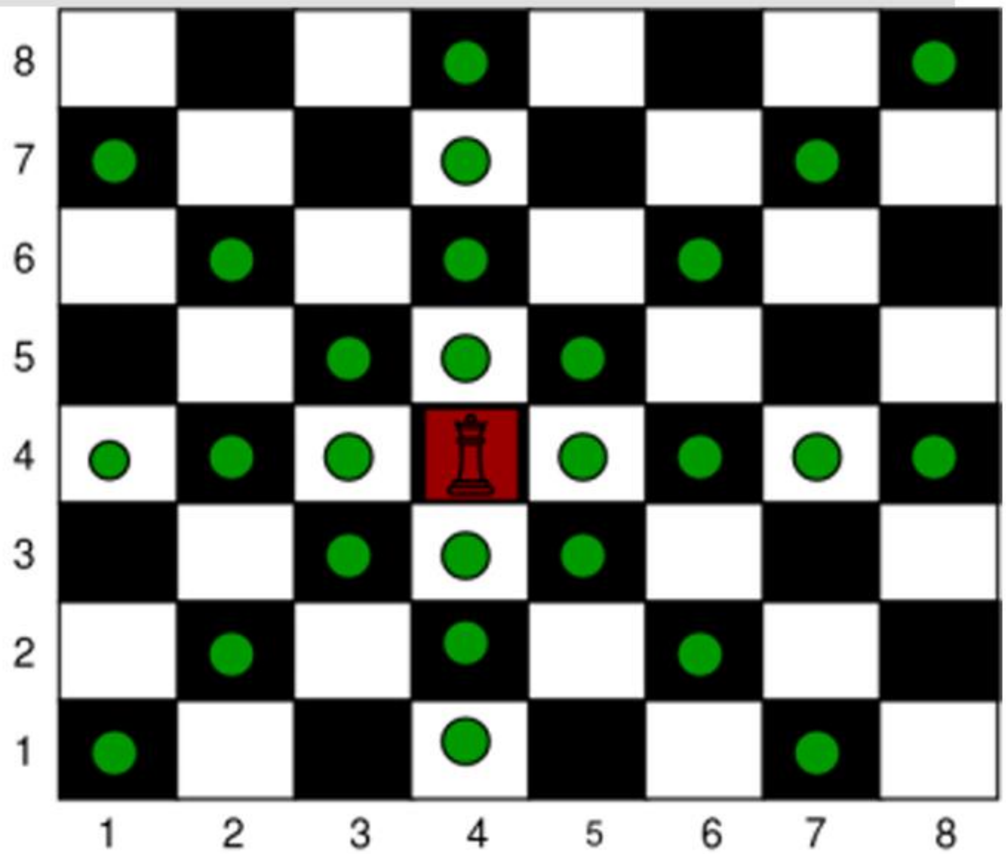
Space Complexity:

- The space complexity is O(N×M), which is required to store the chessboard and track the positions of the queens.

Problem Statement:

3) Consider a N X N chessboard with a Queen and K obstacles. The Queen cannot pass through obstacles. Given the position (x, y) of Queen, the task is to find the number of cells the queen can move.

```
Input : N = 8, x = 4, y = 4,
        K = 0
Output : 27
```



Algorithm/Pseudocode

Initialize the Directions:

- Set initial boundaries for movement in all 8 directions (up, down, left, right, and the 4 diagonals).

Adjust Boundaries for Obstacles:

- For each obstacle, check if it lies in the Queen's path in any of the 8 directions. If it does, update the boundary for that direction accordingly.

Calculate Total Moves:

- For each direction, the number of possible moves is the distance between the Queen's position and the updated boundary.

Output the Total Number of Moves.

Program Code

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Function to calculate the number of cells the Queen can move
int countQueenMoves(int N, int Qx, int Qy, vector<pair<int, int>>& obstacles) {
  // Directions: up, down, left, right, top-left, top-right, bottom-left, bottom-right
  int up = N - Qx;
  int down = Qx - 1;
  int left = Qy - 1;
  int right = N - Qy;
  int topLeft = min(N - Qx, Qy - 1);
  int topRight = min(N - Qx, N - Qy);
  int bottomLeft = min(Qx - 1, Qy - 1);
  int bottomRight = min(Qx - 1, N - Qy);

  // Adjust boundaries based on obstacles
  for (auto& obs : obstacles) {
    int ox = obs.first, oy = obs.second;
```

```
// Same column (vertical movement)
if (oy == Qy) {
  if (ox > Qx) {
    up = min(up, ox - Qx - 1); // obstacle above
  } else {
    down = min(down, Qx - ox - 1); // obstacle below
  }
}

// Same row (horizontal movement)
if (ox == Qx) {
  if (oy > Qy) {
    right = min(right, oy - Qy - 1); // obstacle to the right
  } else {
    left = min(left, Qy - oy - 1); // obstacle to the left
  }
}

// Diagonal movements
if (abs(ox - Qx) == abs(oy - Qy)) {
  if (ox > Qx && oy > Qy) {
    topRight = min(topRight, ox - Qx - 1); // obstacle top-right
  } else if (ox > Qx && oy < Qy) {
    topLeft = min(topLeft, ox - Qx - 1); // obstacle top-left
  } else if (ox < Qx && oy > Qy) {
    bottomRight = min(bottomRight, Qx - ox - 1); // obstacle bottom-right
  } else if (ox < Qx && oy < Qy) {
    bottomLeft = min(bottomLeft, Qx - ox - 1); // obstacle bottom-left
```

```
            }
        }
    }

    // Total possible moves
    return up + down + left + right + topLeft + topRight + bottomLeft + bottomRight;
}

int main() {
    int N, K, Qx, Qy;

    // Inputs
    cout << "Enter the size of the board (N): ";
    cin >> N;

    cout << "Enter the Queen's position (Qx, Qy): ";
    cin >> Qx >> Qy;

    cout << "Enter the number of obstacles (K): ";
    cin >> K;

    vector<pair<int, int>> obstacles(K);
    if (K > 0) {
        cout << "Enter the positions of obstacles (ox oy): " << endl;
        for (int i = 0; i < K; i++) {
            cin >> obstacles[i].first >> obstacles[i].second;
        }
    }
```

```
    int result = countQueenMoves(N, Qx, Qy, obstacles);
    cout << "The number of cells the Queen can move to: " << result << endl;


    return 0;
}
```

Output with verity of test cases

```
Enter the size of the board (N): 8
Enter the Queen's position (Qx, Qy): 4 4
Enter the number of obstacles (K): 0
The number of cells the Queen can move to: 27
```

```
Enter the size of the board (N): 8
Enter the Queen's position (Qx, Qy): 4 4
Enter the number of obstacles (K): 2
Enter the positions of obstacles (ox oy):
5 5
3 2
The number of cells the Queen can move to: 23
```

```
Enter the size of the board (N): 8
Enter the Queen's position (Qx, Qy): 1 1
Enter the number of obstacles (K): 0
The number of cells the Queen can move to: 21
```

```
Enter the size of the board (N): 8
Enter the Queen's position (Qx, Qy): 4 4
Enter the number of obstacles (K): 1
Enter the positions of obstacles (ox oy):
6 4
The number of cells the Queen can move to: 24
```

```
Enter the size of the board (N): 8
Enter the Queen's position (Qx, Qy): 4 4
Enter the number of obstacles (K): 3
Enter the positions of obstacles (ox oy):
5 5
3 3
4 6
The number of cells the Queen can move to: 17
```

Analysis in terms of complexity wherever applicable.

Time Complexity:

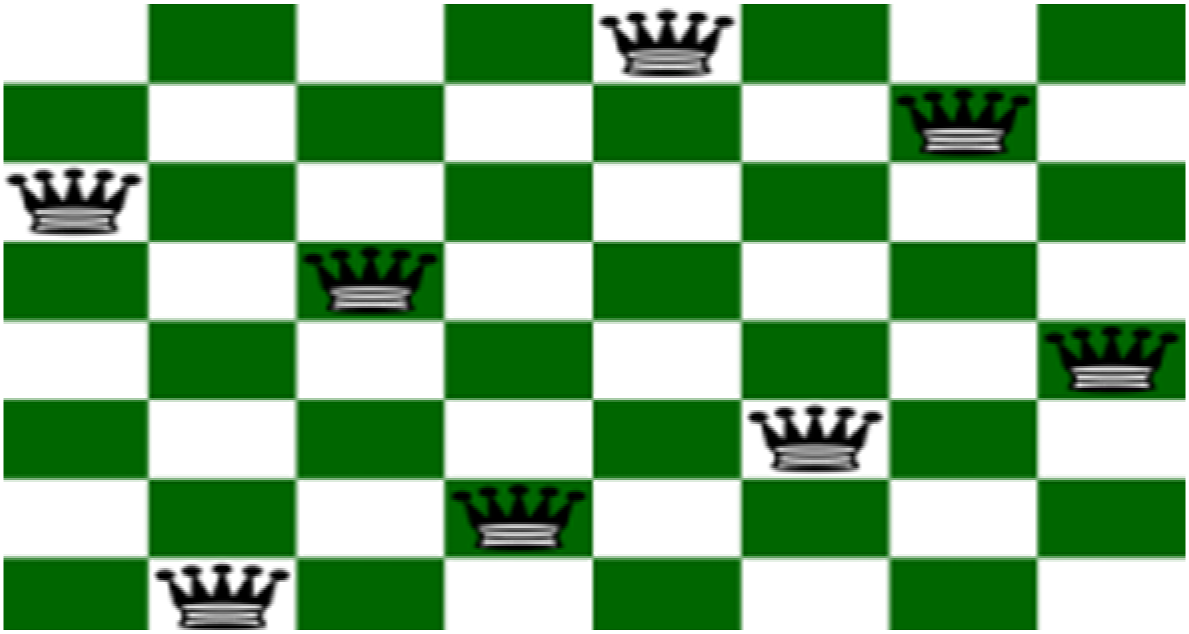
- The algorithm iterates over all obstacles, so the time complexity is O(K) where K is the number of obstacles. For K=0, it simply computes the number of cells the Queen can move to based on the board size, which takes constant time O(1)

Space Complexity:

- The space complexity is O(K) due to the storage of obstacle positions.

Problem Statement:

4) The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, the following is a solution for 8 Queen problem.



Algorithm/Pseudocode

Start with an empty board and place the first queen in the first column.

Check row by row in that column, if a queen can be placed safely.

If a queen can be placed, place it and move to the next column to place another queen.

Backtrack if a queen cannot be placed in any row of the current column, and move the previous queen to a new row in its column.

Repeat until all queens are placed on the board, or backtrack when no solution exists for a column.

Program Code

```
#include <iostream>

#include <vector>

using namespace std;


class NQueens {

private:

    int N;

    vector<vector<string>> solutions;
```

```
// Function to check if placing a queen at (row, col) is safe
bool isSafe(vector<string>& board, int row, int col) {
    // Check this row on left side
    for (int i = 0; i < col; i++) {
        if (board[row][i] == 'Q')
            return false;
    }

    // Check upper diagonal on left side
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 'Q')
            return false;
    }

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

    return true;
}

// Helper function to solve the problem using backtracking
void solveNQueensUtil(vector<string>& board, int col) {
    // If all queens are placed
    if (col == N) {
        solutions.push_back(board);
```

```
            return;
        }

        // Try placing a queen in each row of this column
        for (int i = 0; i < N; i++) {
            if (isSafe(board, i, col)) {
                board[i][col] = 'Q';   // Place the queen
                solveNQueensUtil(board, col + 1); // Recur for next column
                board[i][col] = '.';   // Backtrack and remove the queen
            }
        }
    }

public:
    // Constructor to initialize board size
    NQueens(int n) : N(n) {}

    // Function to solve the N-Queens problem
    vector<vector<string>> solveNQueens() {
        vector<string> board(N, string(N, '.'));
        solveNQueensUtil(board, 0);
        return solutions;
    }
};

int main() {
    int N;
    cout << "Enter the value of N: ";
```

```
    cin >> N;


    NQueens solver(N);
    vector<vector<string>> result = solver.solveNQueens();


    if (result.empty()) {
        cout << "No solution exists for N = " << N << endl;
    } else {
        cout << "Number of solutions: " << result.size() << endl;
        for (const auto& solution : result) {
            for (const string& row : solution) {
                cout << row << endl;
            }
            cout << endl;
        }
    }


    return 0;
}
```

Output with verity of test cases

```
Enter the value of N: 4
Number of solutions: 2
..Q.
Q...
...Q
.Q..

.Q..
...Q
Q...
..Q.
```

```
Enter the value of N: 1
Number of solutions: 1
Q
```

```
Enter the value of N: 5
Number of solutions: 10
Q....
...Q.
.Q...
....Q
..Q..

Q....
..Q..
....Q
.Q...
...Q.

..Q..
Q....
...Q.
.Q...
....Q

...Q.
Q....
..Q..
....Q
.Q...

.Q...
...Q.
Q....
..Q..
....Q

....Q
..Q..
Q....
...Q.
.Q...

.Q...
```

```
...Q.
Q....
..Q..
....Q

....Q
..Q..
Q....
...Q.
.Q...

.Q...
....Q
..Q..
Q....
...Q.

....Q
.Q...
...Q.
Q....
..Q..

...Q.
.Q...
....Q
..Q..
Q....

..Q..
....Q
.Q...
...Q.
Q....
```

Analysis in terms of complexity wherever applicable.

Time Complexity: In the worst case, there are N! possible ways to place the queens (as you place one queen in each column and there are N choices for each queen). The exact time complexity is much smaller than N! due to pruning from the isSafe function.

- **Worst Case**: O(N!)

Space Complexity: O(N^2 since the board uses N×N space and the recursion depth can go up to N

Problem Statement:

5) Given a valid sentence without any spaces between the words and a dictionary of valid English words, find all possible ways to break the sentence into individual dictionary words.

Example:

```
Consider the following dictionary
{ i, like, sam, sung, samsung, mobile, ice,
  and, cream, icecream, man, go, mango}

Input: "ilikesamsungmobile"
Output: i like sam sung mobile
        i like samsung mobile

Input: "ilikeicecreamandmango"
Output: i like ice cream and man go
        i like ice cream and mango
        i like icecream and man go
        i like icecream and mango
```

Algorithm/Pseudocode

Check every prefix of the sentence.

If the prefix is a valid word in the dictionary, **recursively** solve the problem for the remaining part of the string.

If you reach the end of the string and all prefixes are valid, store the solution.

Backtrack to explore all possible ways of splitting the sentence.

Program Code

```
#include <iostream>

#include <unordered_set>

#include <vector>

using namespace std;
```

```
// Function to check if a word is present in the dictionary
bool dictionaryContains(string word, unordered_set<string>& dictionary) {
    return dictionary.find(word) != dictionary.end();
}

// Helper function to perform backtracking and find all word breaks
void wordBreakUtil(string s, unordered_set<string>& dictionary, string result,
vector<string>& output) {
    // If we've reached the end of the string, add the result to the output
    if (s.size() == 0) {
        output.push_back(result);
        return;
    }

    // Try every prefix of the string
    for (int i = 1; i <= s.size(); i++) {
        string prefix = s.substr(0, i);

        // If the prefix is a valid word, proceed to check the remaining substring
        if (dictionaryContains(prefix, dictionary)) {
            string newResult = result + (result.empty() ? "" : " ") + prefix;
            wordBreakUtil(s.substr(i), dictionary, newResult, output);
        }
    }
}

// Main function to find all word breaks
vector<string> wordBreak(string s, unordered_set<string>& dictionary) {
```

```
    vector<string> output;
    wordBreakUtil(s, dictionary, "", output);
    return output;
}

int main() {
    // Create a dictionary of valid words
    unordered_set<string> dictionary = { "i", "like", "sam", "sung", "samsung", "mobile", "ice",
"and", "cream", "icecream", "man", "go", "mango" };

    // Input sentence without spaces
    string input1 = "ilikesamsungmobile";
    string input2 = "ilikeicecreamandmango";

    // Get all possible word breaks for the first input
    vector<string> result1 = wordBreak(input1, dictionary);
    cout << "Possible segmentations for input \"" << input1 << "\":" << endl;
    for (string& sentence : result1) {
        cout << sentence << endl;
    }

    // Get all possible word breaks for the second input
    vector<string> result2 = wordBreak(input2, dictionary);
    cout << "\nPossible segmentations for input \"" << input2 << "\":" << endl;
    for (string& sentence : result2) {
        cout << sentence << endl;
    }
```

```
    return 0;
}
```

Output with verity of test cases

```
Possible segmentations for input "ilikesamsungmobile":
i like sam sung mobile
i like samsung mobile

Possible segmentations for input "ilikeicecreamandmango":
i like ice cream and man go
i like ice cream and mango
i like icecream and man go
i like icecream and mango
```

Analysis in terms of complexity wherever applicable.

Time Complexity: The worst-case time complexity is O(2^N), where N is the length of the string. This is because there are exponentially many ways to split the string.

- The actual time complexity depends on the number of possible valid segmentations and the structure of the dictionary.

Space Complexity: O(N) for recursion stack in backtracking and storage for the result list.