OPEN ENDED LAB

LAB 13

EXERCISE: (Attach all printouts here)

1.What approach did you use to solve problem and why?

The code implements the Rat in a Maze problem using recursive backtracking. It utilizes a recursive function ratinMaze to explore possible paths through the maze, marking the cells in solArr to track the solution. The isSafe function ensures the current cell is within bounds and not blocked. Backtracking is employed by resetting the cell to 0 if a chosen path does not lead to the solution. This approach elegantly explores all possible paths in a depth-first manner, efficiently finding a valid path through the maze.

2. Give the algorithm of above problem, Also analyze the time & space complexity of your algorithm.

Variables Used:

- arr: 2D array representing the maze.
- solArr: 2D array for tracking the solution path.
- n: Size of the maze.
- x, y: Current position in the maze during recursion.

Algorithm:

1. Define isValid() to verify if a cell (r, c) is within the maze boundaries and unblocked.

Implement findPath() to discover all viable paths:
a. Base case: If the current position is the bottom-right cell, add the current path to the result and return.
b. Mark the current cell as visited or blocked.
c. Iterate through all possible directions:

i. Calculate the next position based on the current direction.
ii. If the next position is valid (i.e., isValid() returns true), add the direction to the current path and recursively call findPath() for the next cell.
iii. Backtrack by removing the last direction from the current path.
2. *Before returning, mark the current cell as unvisited or unblocked.*

e. Time Complexity:

- Worst-Case Complexity: In the worst case, the rat needs to explore all possible paths in the maze to reach its destination (the bottom-right corner). Therefore, the worst-case time complexity is **O(2^(n^2)).**
- Best-Case Complexity: The best-case scenario occurs when the rat reaches the destination in just one step (n = 1). In this case, the time complexity would be **O(1)** because it's a constant time to solve the maze.
- Average-Case Complexity: In practical scenarios, it tends to be closer to the worst case for large maze sizes where there are numerous possible paths.

Space Complexity:

- 0(N^2)

f. Give implementation of Rat in a Maze problem.

CODE:

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

bool isSafe(int** arr, int x, int y, int n) {
    if (x <n && y<n && arr [x][y]==1) {
        return true;
    }
    return false;
}
bool ratInMaze(int** arr, int x, int y, int n, int** solArr) {
    if (x == n - 1 && y == n - 1) {
        solArr[x][y] = 1;
        return true;
    }
    if (isSafe(arr, x, y, n)) {
        solArr[x][y] = 1;
        if (ratInMaze(arr, x + 1, y, n, solArr) ||
        ratInMaze(arr, x, y + 1, n, solArr)) {
            return true;
        }
        solArr[x][y] = 0; // Backtracking
        return false;
    }
    return false;
}
int main() {
    int n;
    cout << "Enter size of rows: ";
    cin >> n;
    int** arr = new int*[n];
```

```
    for (int i = 0; i < n; i++) {
        arr[i] = new int[n];
    }

    cout << "Enter the maze (0 or 1):\n";
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cin >> arr[i][j];
        }
    }
    int** solArr = new int*[n];
    for (int i = 0; i < n; i++) {
        solArr[i] = new int[n];
        for (int j = 0; j < n; j++) {
            solArr[i][j] = 0;
        }
    }
    if (ratInMaze(arr, 0, 0, n, solArr)) {
        cout << "Maze is solved:\n";
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                cout << solArr[i][j] << " ";
            }
            cout << endl;
        } }
 else {
        cout << "No solution found for the maze." << endl;
    }
    return 0;
}
```

DELIVERABLES :

```
Enter size of rows: 5
Enter the maze (0 or 1):
1 0 1 0 1
1 1 1 1 1
0 1 0 1 1
1 0 0 1 1
1 1 1 0 1
Maze is solved:
1 0 0 0 0
1 1 1 1 0
0 0 0 1 0
0 0 0 1 1
0 0 0 0 1

--------------------------------
```

COMPLEX ENGINEERING PROBLEM

LAB 14

EXERCISE: : (Attach all printouts here)

1. What approach do you used to solve problem and why?

The code utilizes a backtracking approach to solve the N-Queens problem. This approach involves systematically exploring and placing queens on the chessboard, validating each queen's position to ensure no two queens attack each other.

Backtracking is chosen because it systematically searches for a solution, efficiently discarding configurations that cannot lead to a valid solution.

2. Give the algorithm of above problem, Also analyze the time & space complexity of your algorithm.

Variables Used:

- arr[ ][ ]: 2D array representing the chessboard.
- x, y: Current row and column being processed.
- n: Size of the chessboard.
- col: Loop variable to iterate through columns.
- isSafe(): Function to check the safety of placing a queen.
- nQueen(): Recursive function to solve the N-Queens problem.

Algorithm :

1. Initialize an n x n chessboard grid.
2. Begin with the first row.
3. For each row:

a. Attempt to place a queen in each column.
b. If placing a queen at (x, col) is safe, proceed to the next row and repeat step 3.
c. If all columns are tried but a solution isn't found, backtrack and explore other possibilities.
4. Continue this process until all queens are positioned or all potential options are exhausted.

5. If a valid solution is discovered, display the solved chessboard configuration; otherwise, output "No solution found for N-Queens."

Time Complexity:

- The time complexity of the N-Queens problem using backtracking is exponential: O(n!), where n is the size of the chessboard.
- In the worst case, the algorithm explores all possible combinations of queen placements.

- **Space Complexity:**

- The space complexity of the algorithm is O(n2) to store the n×n chessboard.

Give implementation of 8 Queen Problem

CODE:

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

bool isSafe(int** arr, int x, int y, int n) {
    // Check if there is a queen in the same column above the current position
    for (int row = 0; row < x; row++) {
        if (arr[row][y] == 1) {
            return false;
        }
    }
    // Check upper left diagonal
    for (int row = x, col = y; row >= 0 && col >= 0; row--, col--) {
        if (arr[row][col] == 1) {
            return false;
        }
    }
    // Check upper right diagonal
    for (int row = x, col = y; row >= 0 && col < n; row--, col++) {
        if (arr[row][col] == 1) {
            return false;
        }
    }
    return true;
}
bool nQueen(int** arr, int x, int n) {
    if (x >= n) {
        return true;
    }
    for (int col = 0; col < n; col++) {
        if (isSafe(arr, x, col, n)) {
```

```
            arr[x][col] = 1;
            if (nQueen(arr, x + 1, n)) {
                return true;
            }
            arr[x][col] = 0; // Backtrack
        }}
    return false;
}
int main() {
    int n;
    cout << "Enter rows(must me greater than 3): " << endl;
    cin >> n;
    int** arr = new int*[n];
    for (int i = 0; i < n; i++) {
        arr[i] = new int[n];
        for (int j = 0; j < n; j++) {
            arr[i][j] = 0;
        }}
    if (nQueen(arr, 0, n)) {
        cout << "N-Queens is solved:" << endl;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                cout << arr[i][j] << " ";}
            cout << endl;
        }
    } else {
        cout << "No solution found for N-Queens." << endl;
    }
    return 0;
}
```

DELIVERABLES :

```
Enter rows(must me greater than 3):
5
N-Queens is solved:
1 0 0 0 0
0 0 1 0 0
0 0 0 0 1
0 1 0 0 0
0 0 0 1 0

--------------------------------
```