KHUSHI AGARWAL
B-5

Week 7 Lab B
Enrollment No - 2210312 4

Ans1)

```
#include <iostream>
using namespace std;
struct Node {
   int data;
   Node* child;
   Node* next;

   Node(int val) : data(val), child(nullptr), next(nullptr) {}
};

class MultiLinkedList {
public:
   MultiLinkedList() : head(nullptr) {}

   // Flatten the Multi-Linked List
   Node* flatten() {
      return flattenHelper(head);
   }

   // Insert a Node into the Multi-Linked List
   void insert(int data) {
      Node* newNode = new Node(data);
      if (!head || data < head->data) {
         newNode->next = head;
         head = newNode;
      } else {
         Node* current = head;
         while (current->next && current->next->data < data) {
            current = current->next;
         }
         newNode->next = current->next;
         current->next = newNode;
      }
   }

   // Print the Multi-Linked List (in a readable format)
   void print() {
      printHelper(head);
   }
```

```
private:
    Node* head;

    // Helper function to flatten the Multi-Linked List
    Node* flattenHelper(Node* node) {
        if (!node) return nullptr;

        Node* sortedNext = flattenHelper(node->next);
        Node* flattenedChild = flattenHelper(node->child);

        if (flattenedChild) {
            node->next = flattenedChild;
            while (flattenedChild->next) {
                flattenedChild = flattenedChild->next;
            }
            flattenedChild->next = sortedNext;
        }

        node->child = nullptr;

        return node;
    }

    // Helper function to print the Multi-Linked List
    void printHelper(Node* node) {
        while (node) {
            cout << node->data << " -> ";
            if (node->child) {
                printHelper(node->child);
            }
            node = node->next;
        }
        cout << "NULL" << endl;
    }
};

int main() {
    MultiLinkedList multiList;
    multiList.insert(5);
    multiList.insert(10);
    multiList.insert(2);
    multiList.insert(7);
    multiList.insert(8);
```

```
    Node* flattenedHead = multiList.flatten();

    cout << "Flattened Multi-Linked List:" << endl;
    multiList.print();

    return 0;
}
```

Output:

```
Flattened Multi-Linked List:
2 -> 5 -> 7 -> 8 -> 10 -> NULL

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

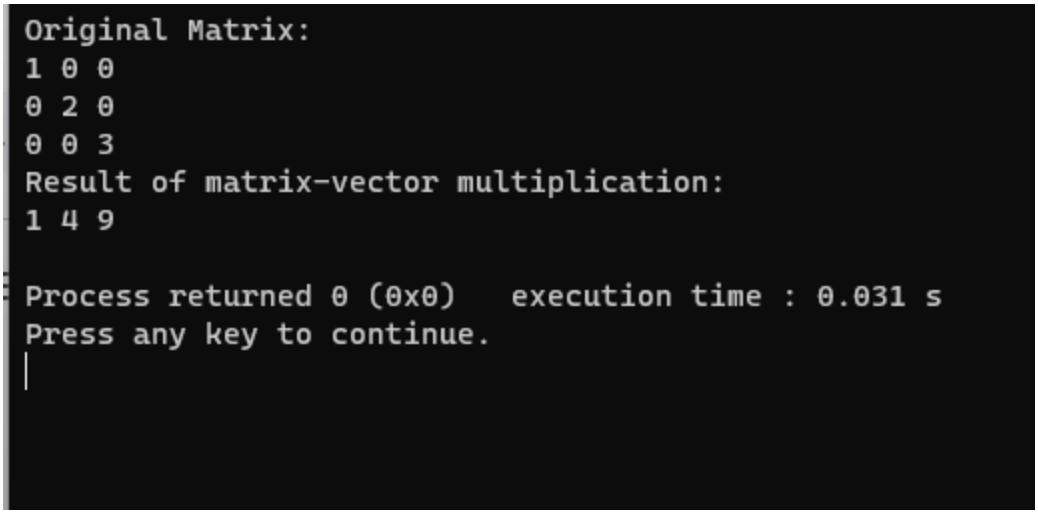
Ans2)

```
#include <iostream>
using namespace std;
class CSRMatrix {
private:
    double* values;
    int* columns;
    int* row_ptr;
    int num_rows;
    int num_cols;
    int num_nonzeros;

public:
    CSRMatrix(const double** matrix, int rows, int cols) : num_rows(rows), num_cols(cols),
num_nonzeros(0) {
        // Count non-zero elements
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                if (matrix[i][j] != 0.0) {
                    num_nonzeros++;
                }
            }
        }

        // Allocate memory for values, columns, and row_ptr
        values = new double[num_nonzeros];
```

```
    columns = new int[num_nonzeros];
    row_ptr = new int[num_rows + 1];

    int nnz = 0; // Number of non-zero elements
    row_ptr[0] = 0;

    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            if (matrix[i][j] != 0.0) {
                values[nnz] = matrix[i][j];
                columns[nnz] = j;
                nnz++;
            }
        }
        row_ptr[i + 1] = nnz;
    }
}

~CSRMatrix() {
    delete[] values;
    delete[] columns;
    delete[] row_ptr;
}

double* multiply(const double* vector) const {
    double* result = new double[num_rows];

    for (int i = 0; i < num_rows; ++i) {
        result[i] = 0.0;
        for (int j = row_ptr[i]; j < row_ptr[i + 1]; ++j) {
            result[i] += values[j] * vector[columns[j]];
        }
    }

    return result;
}

void displayMatrix() const {
    cout << "Original Matrix:" << endl;
    int nnz = 0;
    for (int i = 0; i < num_rows; ++i) {
        for (int j = 0; j < num_cols; ++j) {
            if (nnz < row_ptr[i + 1] && j == columns[nnz]) {
                cout << values[nnz] << " ";
```

```
                nnz++;
            } else {
                cout << "0 ";
            }
        }
        cout << endl;
    }
  }
};

int main() {
  const double* matrix[3] = {
    new double[3]{1.0, 0.0, 0.0},
    new double[3]{0.0, 2.0, 0.0},
    new double[3]{0.0, 0.0, 3.0}
  };

  CSRMatrix csr_matrix(matrix, 3, 3);

  csr_matrix.displayMatrix();

  const double vector[3] = {1.0, 2.0, 3.0};
  double* result = csr_matrix.multiply(vector);

  cout << "Result of matrix-vector multiplication:" << endl;
  for (int i = 0; i < 3; ++i) {
    cout << result[i] << " ";
  }
  cout << endl;

  // Clean up memory for matrix elements
  for (int i = 0; i < 3; ++i) {
    delete[] matrix[i];
  }
  delete[] result;

  return 0;
}
```

Output:



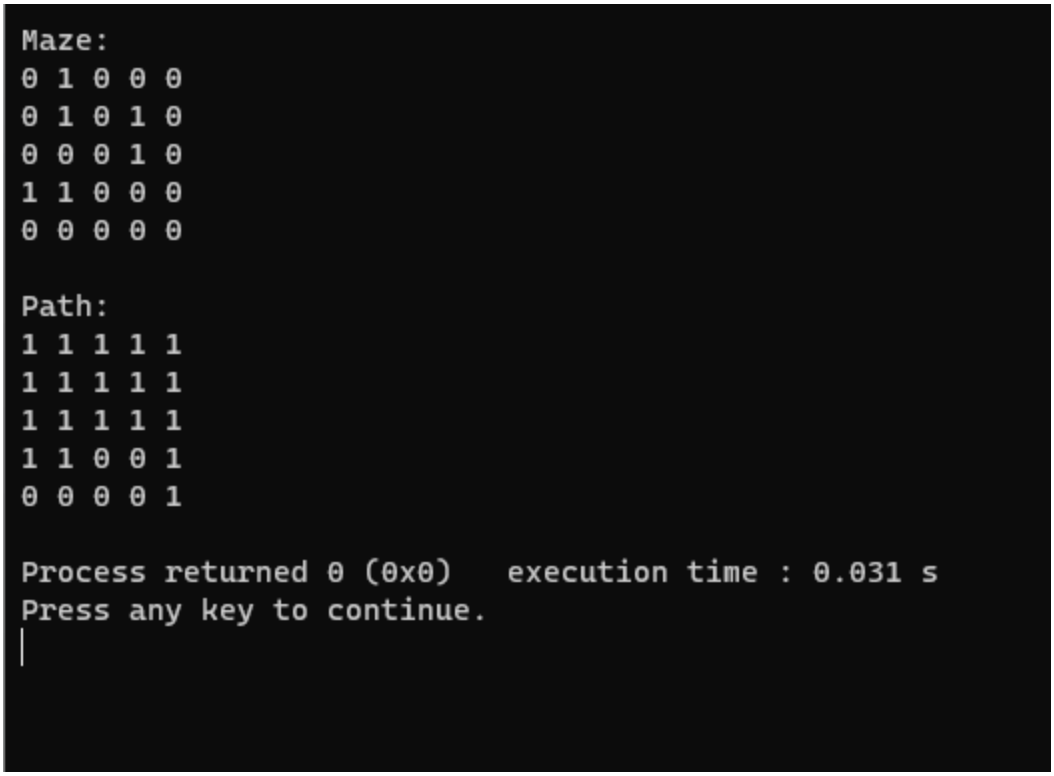
Ans3)

```
#include <iostream>
#include <vector>

using namespace std;

// Define directions for moving up, down, left, and right
const int dx[] = {-1, 1, 0, 0};
const int dy[] = {0, 0, -1, 1};

// Function to check if a given cell is valid and can be visited
bool isValidCell(int x, int y, vector<vector<int>>& maze, vector<vector<int>>& visited) {
   int rows = maze.size();
   int cols = maze[0].size();

   // Check if the cell is within the maze boundaries, is not blocked, and has not been visited
   return (x >= 0 && x < rows && y >= 0 && y < cols && maze[x][y] == 0 && visited[x][y] == 0);
}

// Recursive function to find and print a path using Depth-First Search
bool findAndPrintPathDFS(int x, int y, vector<vector<int>>& maze, vector<vector<int>>&
visited) {
   int rows = maze.size();
   int cols = maze[0].size();

   // If the rat has reached the destination cell, mark it as part of the path
   if (x == rows - 1 && y == cols - 1) {
      visited[x][y] = 1;
      return true;
```

```
    }

    // Try moving in all four directions
    for (int dir = 0; dir < 4; ++dir) {
        int newX = x + dx[dir];
        int newY = y + dy[dir];

        if (isValidCell(newX, newY, maze, visited)) {
            // Mark the current cell as visited
            visited[newX][newY] = 1;

            // Recursively explore the next cell
            if (findAndPrintPathDFS(newX, newY, maze, visited)) {
                return true; // Path found
            }
        }
    }

    return false; // No path found
}

// Function to find and print a path from start to destination in the maze
void findAndPrintPath(vector<vector<int>>& maze) {
    int rows = maze.size();
    int cols = maze[0].size();

    // Initialize a visited matrix with all zeros
    vector<vector<int>> visited(rows, vector<int>(cols, 0));

    // Start DFS from the top-left corner (0, 0)
    if (findAndPrintPathDFS(0, 0, maze, visited)) {
        // Print the maze with the path
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                if (visited[i][j] == 1) {
                    cout << "1 ";
                } else {
                    cout << maze[i][j] << " ";
                }
            }
            cout << endl;
        }
    } else {
        cout << "No path found." << endl;
```

```
    }
}

int main() {
    // Example maze (you can replace this with your own maze)
    vector<vector<int>> maze = {
        {0, 1, 0, 0, 0},
        {0, 1, 0, 1, 0},
        {0, 0, 0, 1, 0},
        {1, 1, 0, 0, 0},
        {0, 0, 0, 0, 0}
    };

    // Print the original maze
    cout << "Maze:" << endl;
    for (const vector<int>& row : maze) {
        for (int cell : row) {
            cout << cell << " ";
        }
        cout << endl;
    }

    cout << endl;

    // Find and print the path in the maze
    cout << "Path:" << endl;
    findAndPrintPath(maze);

    return 0;
}
```

Output:



Ans4)

```
#include <iostream>
#include <vector>

using namespace std;

class Solution {
public:
   vector<vector<string>> solveNQueens(int n) {
      vector<vector<string>> result;
      vector<string> board(n, string(n, '.'));

      solveNQueensHelper(result, board, 0, n);

      return result;
   }

private:
   void solveNQueensHelper(vector<vector<string>>& result, vector<string>& board, int row, int
n) {
      if (row == n) {
         result.push_back(board);
         return;
```

```
        }

        for (int col = 0; col < n; col++) {
            if (isValid(board, row, col, n)) {
                board[row][col] = 'Q';
                solveNQueensHelper(result, board, row + 1, n);
                board[row][col] = '.';
            }
        }
    }

    bool isValid(vector<string>& board, int row, int col, int n) {
        // Check the column
        for (int i = 0; i < row; i++) {
            if (board[i][col] == 'Q') {
                return false;
            }
        }

        // Check the upper-left diagonal
        for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] == 'Q') {
                return false;
            }
        }

        // Check the upper-right diagonal
        for (int i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++) {
            if (board[i][j] == 'Q') {
                return false;
            }
        }

        return true;
    }
};

int main() {
    Solution solution;
    int n;
    cout << "Enter the size of the chessboard (N): ";
    cin >> n;

    vector<vector<string>> result = solution.solveNQueens(n);
```

```
    for (const vector<string>& solutionBoard : result) {
        for (const string& row : solutionBoard) {
            cout << row << endl;
        }
        cout << endl;
    }

    return 0;
}
```

Output: