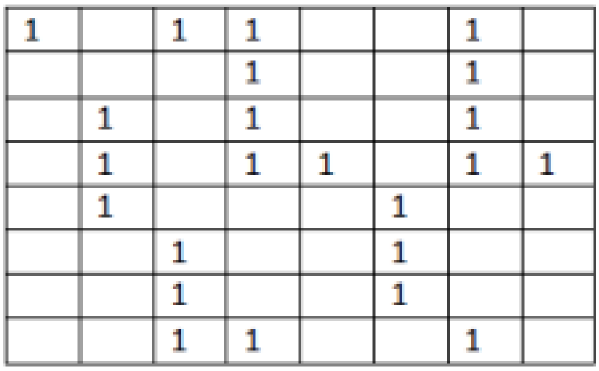
Week 6-LAB B

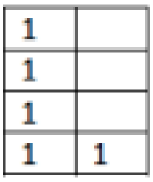
1. Search shape L of given size (user input) from following matrix. And give its size
Example Input



1		1	1			1	
			1			1	
	1		1			1	
	1		1	1		1	1
	1				1		
		1			1		
		1			1		
		1	1			1	

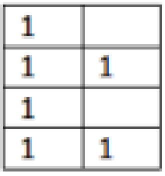
Output:

Eg size of below L shape is 5.



1	
1	
1	
1	1

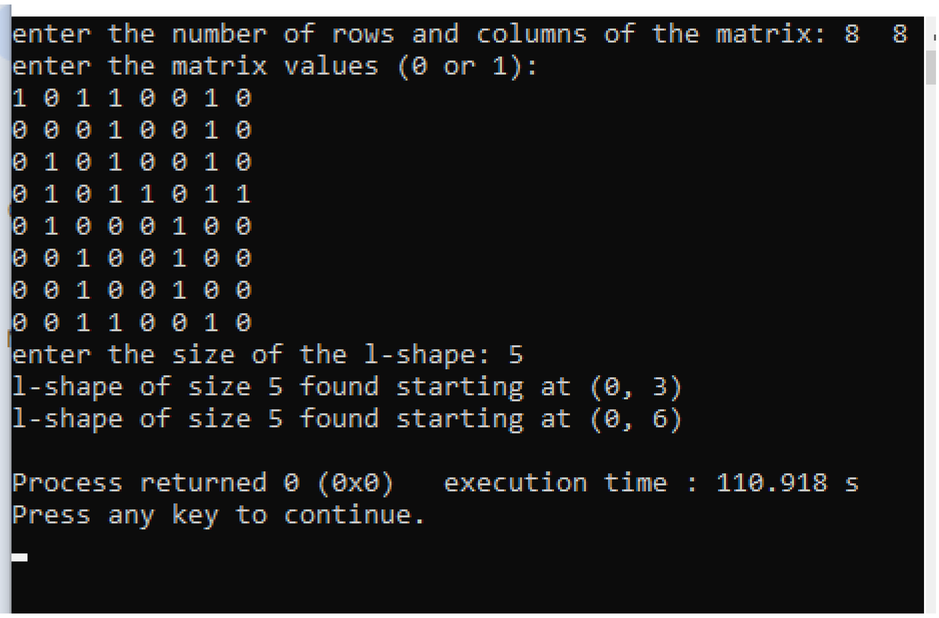
Below shape is not L shape



1	
1	1
1	
1	1

ANS:

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

const int MAX_SIZE = 100;

bool checklshape(int matrix[MAX_SIZE][MAX_SIZE], int n, int m, int row, int col, int height) {
    stack<pair<int, int>> s;

    if (row + height - 1 < n && col + 1 < m) {
        for (int k = 0; k < height; k++) {
            if (matrix[row + k][col] == 1) {
                s.push({row + k, col});
            } else {
                return false;
            }
        }
        if (matrix[row + height - 1][col + 1] == 1) {
            return true;
        }
    }

    while (!s.empty()) s.pop();
    if (row + 1 < n && col + height - 1 < m) {
        for (int k = 0; k < height; k++) {
            if (matrix[row][col + k] == 1) {
                s.push({row, col + k});
            } else {
                return false;
            }
        }
        if (matrix[row + 1][col + height - 1] == 1) {
            return true;
        }
    }

    return false;
}

int main() {
    int n, m, size;
    cout << "enter the number of rows and columns of the matrix: ";
    cin >> n >> m;

    int matrix[MAX_SIZE][MAX_SIZE];
    cout << "enter the matrix values (0 or 1):\n";
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> matrix[i][j];
        }
    }

    cout << "enter the size of the l-shape: ";
    cin >> size;

    if (size < 2) {
        cout << "invalid size for l-shape. size must be at least 2.\n";
        return 0;
    }

    int height = size - 1;
    bool found = false;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (matrix[i][j] == 1 && checklshape(matrix, n, m, i, j, height)) {
                cout << "l-shape of size " << size << " found starting at (" << i << ", " << j << ")\n";
                found = true;
            }
        }
    }

    if (!found) {
        cout << "no l-shape of size " << size << " found in the matrix.\n";
    }
    return 0;
}
```



```
enter the number of rows and columns of the matrix: 8  8
enter the matrix values (0 or 1):
1 0 1 1 0 0 1 0
0 0 0 1 0 0 1 0
0 1 0 1 0 0 1 0
0 1 0 1 1 0 1 1
0 1 0 0 0 1 0 0
0 0 1 0 0 1 0 0
0 0 1 0 0 1 0 0
0 0 1 1 0 0 1 0
enter the size of the l-shape: 5
l-shape of size 5 found starting at (0, 3)
l-shape of size 5 found starting at (0, 6)

Process returned 0 (0x0)   execution time : 110.918 s
Press any key to continue.
```

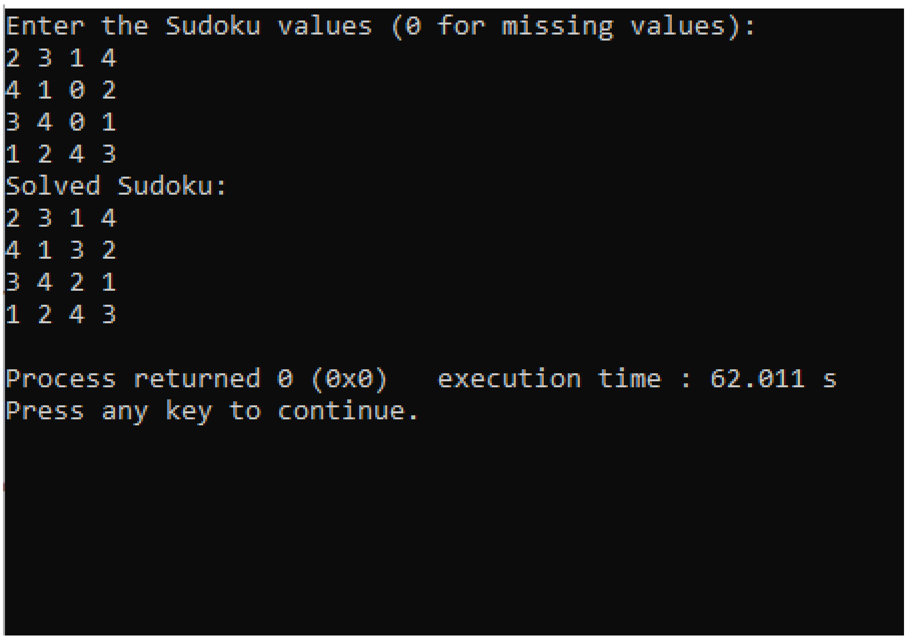
2. Consider 4x4 Sudoku problem. We have kept 14 correct values; however, 2 values are missing. But you have to find these missing values by implementing stack.

2	3	1	4
4	1	?	2
3	4	?	1
1	2	4	3

Ans:

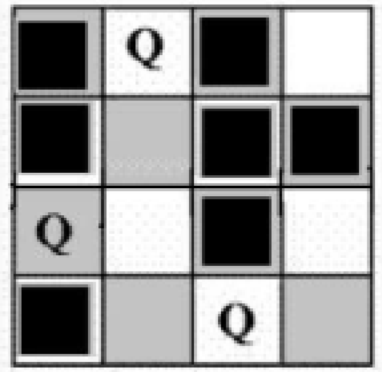
```
#include <iostream>

#include <stack>

using namespace std;

const int size = 4;

          bool issafe(int board[size][size], int row, int col, int num) {

   for (int x = 0; x < size; x++) {

      if (board[row][x] == num || board[x][col] == num) {
            return false;
        }
    }
    int startrow = row - row % 2, startcol = col - col % 2;
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            if (board[i + startrow][j + startcol] == num) {
                return false;
            }
        }
    }

    return true;
}

bool solvesudoku(int board[size][size]) {
    stack<pair<int, int> > positions;
    int missingcount = 0;

    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (board[i][j] == 0) {
                positions.push(make_pair(i, j));
                missingcount++;
            }
        }
```

```
}

if (missingcount != 2) return false;

while (!positions.empty()) {
    pair<int, int> pos = positions.top();
    positions.pop();
    int row = pos.first;
    int col = pos.second;

    bool placed = false;
    for (int num = 1; num <= 4; num++) {
        if (issafe(board, row, col, num)) {
            board[row][col] = num;

            bool allfilled = true;
            for (int i = 0; i < size; i++) {
                for (int j = 0; j < size; j++) {
                    if (board[i][j] == 0) {
                        allfilled = false;
                        break;
                    }
                }
                if (!allfilled) break;
            }
            if (allfilled) return true;
}
                placed = true;
                break;
            }
        }

        if (!placed) {
            board[row][col] = 0;
            if (positions.empty()) {
                positions.push(make_pair(row, col));
            }
        }
    }

    return false;
}

void printboard(int board[size][size]) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
}

int main() {
    int board[size][size];


    cout << "enter the sudoku values (0 for missing values):\n";
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            cin >> board[i][j];
        }
    }


    if (solvesudoku(board)) {
        cout << "solved sudoku:\n";
        printboard(board);
    } else {
        cout << "no solution exists." << endl;
    }


    return 0;
}
```



```
Enter the Sudoku values (0 for missing values):
2 3 1 4
4 1 0 2
3 4 0 1
1 2 4 3
Solved Sudoku:
2 3 1 4
4 1 3 2
3 4 2 1
1 2 4 3

Process returned 0 (0x0)   execution time : 62.011 s
Press any key to continue.
```

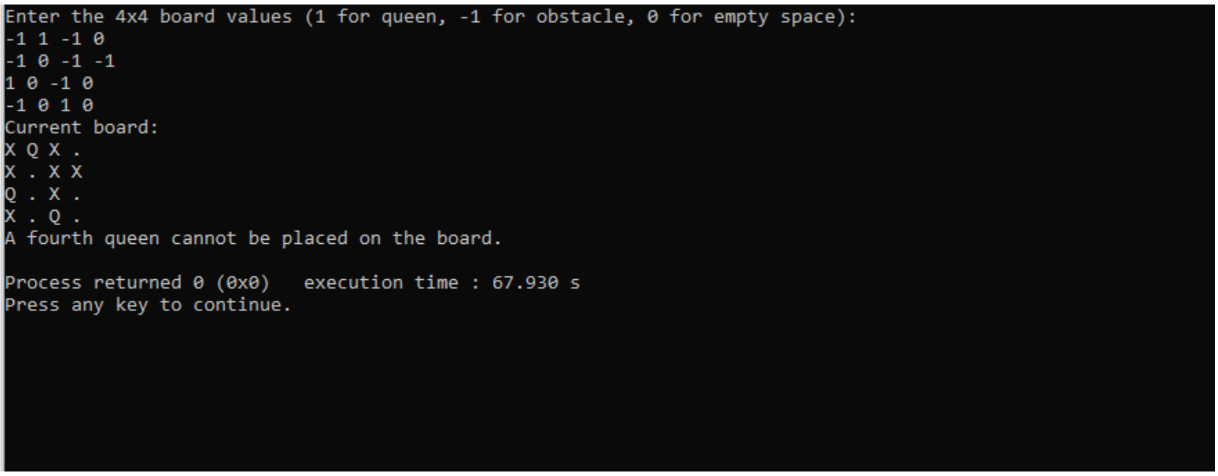
3. In a 4x4, 4 –queen problem, we have already kept 3 queens successfully. There are some obstacles represented using black colors. You have to write a code using stack to find out if you can now place a fourth queen or not? You cannot put a queen at obstacle. However, a queen can cross an obstacle to hit another queen.



Ans:

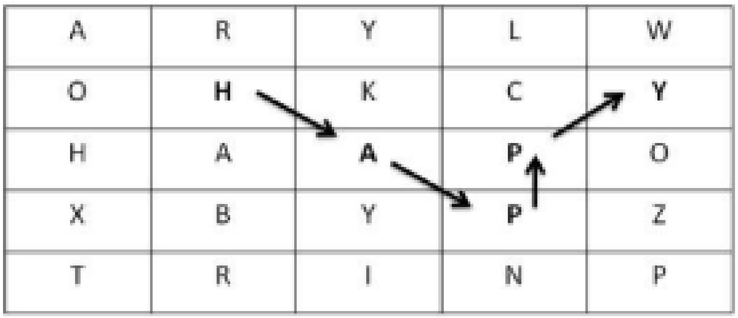
```
#include <iostream>

#include <cmath>



using namespace std;



const int size = 4;
```

```
bool issafe(int board[size][size], int row, int col) {
    for (int i = 0; i < size; i++) {
        if (board[i][col] == 1) {
            return false;
        }
    }
    for (int j = 0; j < size; j++) {
        if (board[row][j] == 1) {
            return false;
        }
    }
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (board[i][j] == 1) {
                if (abs(i - row) == abs(j - col)) {
                    int rowstep = (i < row) ? 1 : -1;
                    int colstep = (j < col) ? 1 : -1;
                    int r = i + rowstep;
                    int c = j + colstep;
                    while (r != row && c != col) {
                        if (board[r][c] == -1) {
                            return true;
                        }
                        r += rowstep;
                        c += colstep;
                    }
                    return false;
                }
            }
        }
    }
    return true;
}

bool canplacefourthqueen(int board[size][size]) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (board[i][j] == 0 && issafe(board, i, j)) {
                return true;
            }
        }
    }
    return false;
}

void printboard(int board[size][size]) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (board[i][j] == 1) {
                cout << "Q ";
            } else if (board[i][j] == -1) {
                cout << "X ";
            } else {
                cout << ". ";
            }
        }
        cout << endl;
    }
}

int main() {
    int board[size][size];

    cout << "enter the 4x4 board values (1 for queen, -1 for obstacle, 0 for empty space):\n";
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            cin >> board[i][j];
        }
    }

    cout << "current board:\n";
    printboard(board);

    if (canplacefourthqueen(board)) {
        cout << "a fourth queen can be placed on the board." << endl;
    } else {
        cout << "a fourth queen cannot be placed on the board." << endl;
    }
    return 0;

}
```



```
Enter the 4x4 board values (1 for queen, -1 for obstacle, 0 for empty space):
-1 1 -1 0
-1 0 -1 -1
1 0 -1 0
-1 0 1 0
Current board:
X Q X .
X . X X
Q . X .
X . Q .
A fourth queen cannot be placed on the board.

Process returned 0 (0x0)   execution time : 67.930 s
Press any key to continue.
```

4. Write a function to check whether a given string exists in the two-dimensional character matrix or not. Print the path if the string exists. You may use all movements such as left, right, up, and down. Consider the following two dimensional character matrix and the string to be searched is "HAPPY". Example is shown below



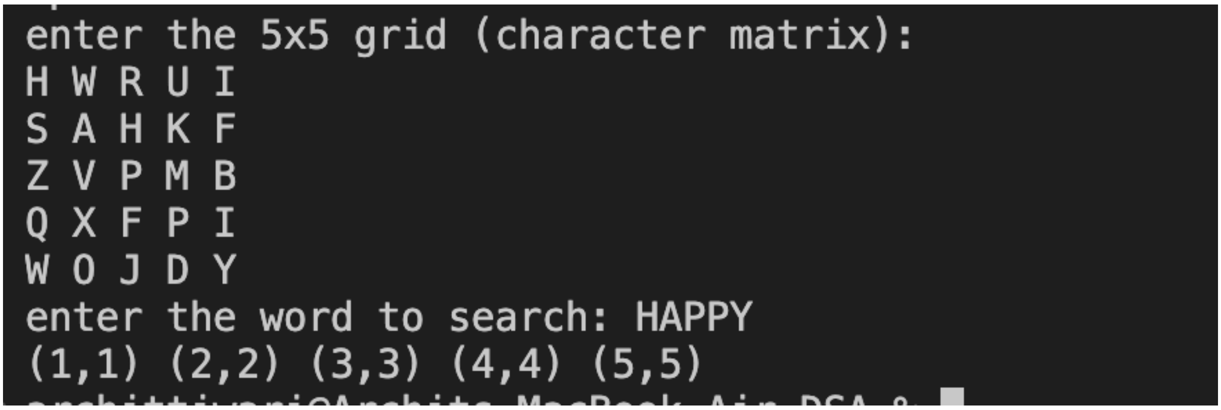
A	R	Y	L	W
O	H	K	C	Y
H	A	A	P	O
X	B	Y	P	Z
T	R	I	N	P

Assumption: a character once visited, cannot be visited again.
Output: Path: (1,1), (2,2), (3,3), (2,3), (1,4)

Ans:

```
#include <iostream>
#include <stack>

using namespace std;

const int rows = 5;
const int cols = 5;

bool is_valid(int x, int y, bool visited[rows][cols]) {
    return (x >= 0 && x < rows && y >= 0 && y < cols && !visited[x][y]);
}
```

```
bool search_word(char matrix[rows][cols], string word) {
    bool visited[rows][cols] = {false};
    int directions[8][2] = {
        {1, 0},
        {-1, 0},
        {0, 1},
        {0, -1},
        {1, 1},
        {1, -1},
        {-1, 1},
        {-1, -1}
    };
    int path[rows * cols][2];
    int path_length = 0;

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (matrix[i][j] == word[0]) {
                stack<pair<pair<int, int>, int> > stk;
                stk.push(make_pair(make_pair(i, j), 0));
                visited[i][j] = true;
                path[path_length][0] = i + 1;
                path[path_length][1] = j + 1;
                path_length++;

                while (!stk.empty()) {
                    pair<pair<int, int>, int> top_element = stk.top();
                    int current_row = top_element.first.first;
                    int current_col = top_element.first.second;
                    int index = top_element.second;
                    stk.pop();

                    if (index == word.length() - 1) {
                        for (int k = 0; k < path_length; k++) {
                            cout << "(" << path[k][0] << "," << path[k][1] << ") ";
                        }
                        cout << endl;
                        return true;
                    }

                    for (int d = 0; d < 8; d++) {
                        int new_x = current_row + directions[d][0];
                        int new_y = current_col + directions[d][1];

                        if (is_valid(new_x, new_y, visited) && matrix[new_x][new_y] == word[index + 1]) {
                            visited[new_x][new_y] = true;
                            stk.push(make_pair(make_pair(new_x, new_y), index + 1));
                            path[path_length][0] = new_x + 1;
                            path[path_length][1] = new_y + 1;
                            path_length++;
                            break;
                        }
                    }

                    if (!stk.empty() && stk.top().second == index) {
                        visited[stk.top().first.first][stk.top().first.second] = false;
                        path_length--;
                    }
                }
                visited[i][j] = false;
            }
        }
    }
    return false;
}

int main() {
    char matrix[rows][cols];
    string word;

    cout << "enter the 5x5 grid (character matrix):" << endl;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cin >> matrix[i][j];
        }
    }

    cout << "enter the word to search: ";
    cin >> word;

    if (!search_word(matrix, word)) {
        cout << "word not found." << endl;
    }

    return 0;
}
```



```
enter the 5x5 grid (character matrix):
H W R U I
S A H K F
Z V P M B
Q X F P I
W O J D Y
enter the word to search: HAPPY
(1,1) (2,2) (3,3) (4,4) (5,5)
```

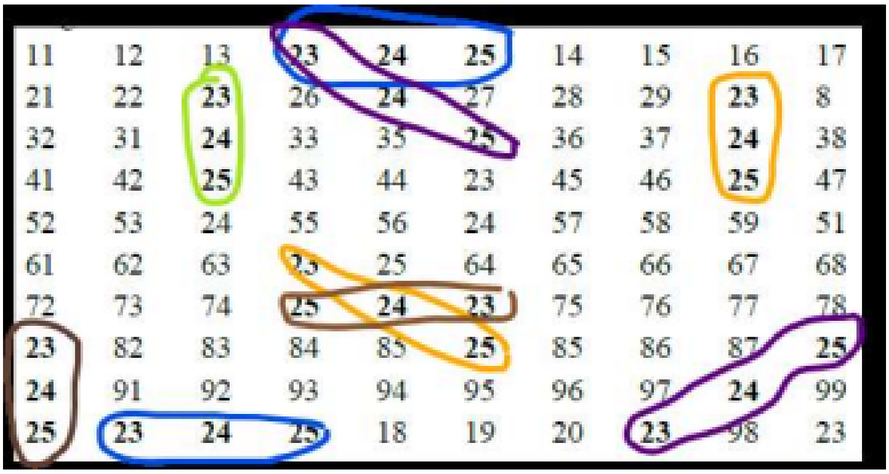
5. Consider following two matrices

=		2 13 23 24 25 14 15 16 17 21 22 23 26 24 2 32 31 24 33 35 25 36 37 24 38 41 42 25 43 6 25 47 52 53 24 55 56 24 57 58 59 51 61 6 4 65 66 67 68 72 73 74 25 24 23 75 76 77 7 4 85 25 85 86 87 25 24 91 92 93 94 95 96 9 3 24 25 18 19 20 23 98 23	

M2 = [23, 24, 25]

Write a function to check whether elements of M2 are present in M1. If elements exist, print the count that how many times M2 elements present in M1. You may use all movements such as left, right, up, and down.

Output: 9

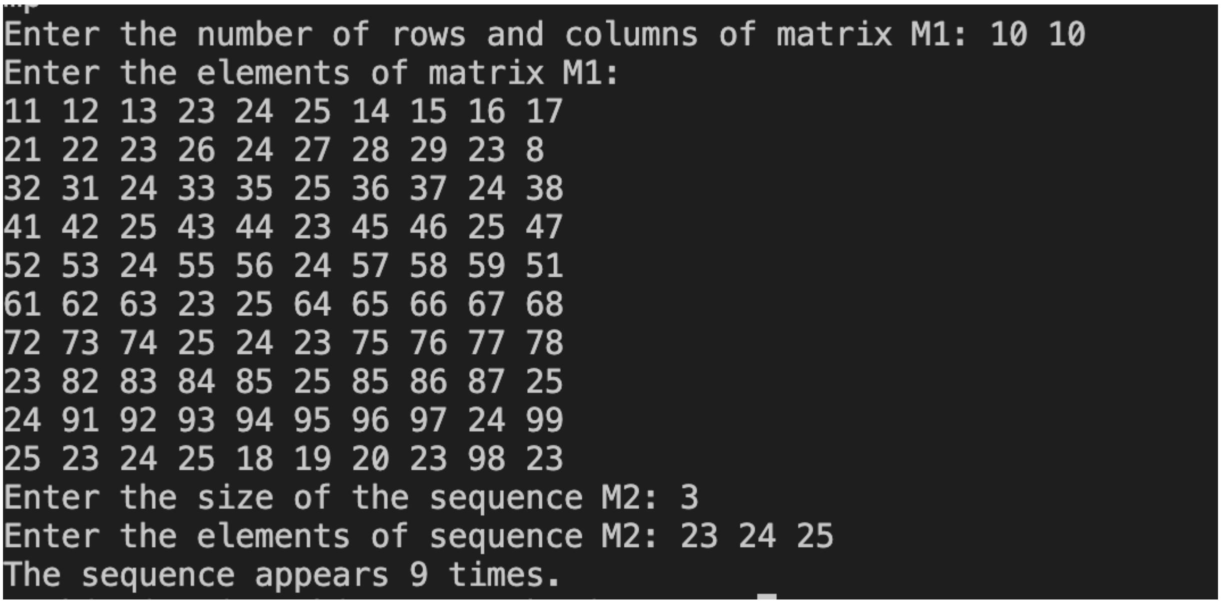


11	12	13	23	24	25	14	15	16	17
21	22	23	26	24	27	28	29	23	8
32	31	24	33	35	25	36	37	24	38
41	42	25	43	44	23	45	46	25	47
52	53	24	55	56	24	57	58	59	51
61	62	63	23	25	64	65	66	67	68
72	73	74	25	24	23	75	76	77	78
23	82	83	84	85	25	85	86	87	25
24	91	92	93	94	95	96	97	24	99
25	23	24	25	18	19	20	23	98	23

Ans:

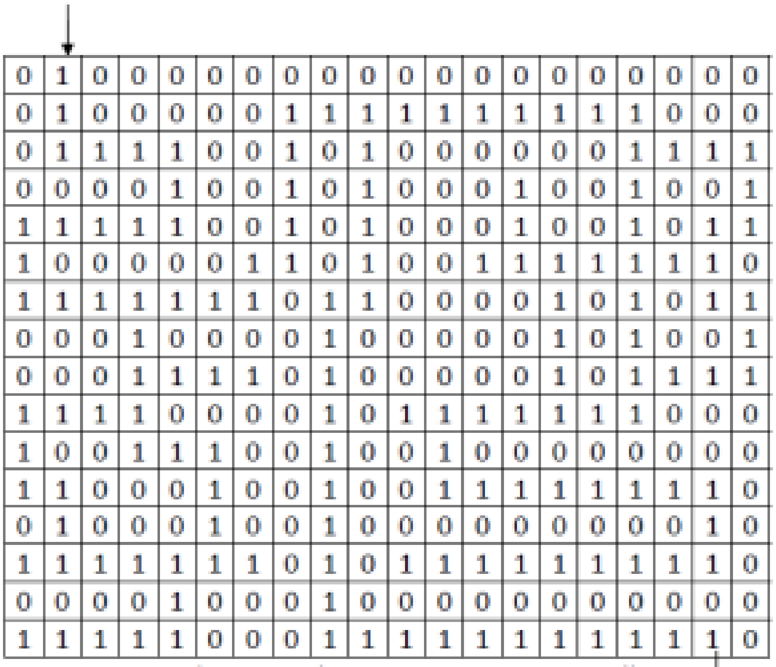
```
#include <iostream>

#include <vector>


using namespace std;


struct direction {

    int x;

    int y;


    direction(int a, int b) : x(a), y(b) {}

};
```

```
bool checksequence(vector<vector<int> >& m1, int i, int j, vector<int>& m2, int di, int dj) {
    for (int k = 0; k < m2.size(); k++) {
        int new_i = i + k * di;
        int new_j = j + k * dj;

        if (new_i < 0 || new_i >= m1.size() || new_j < 0 || new_j >= m1[0].size()) {
            return false;
        }
        if (m1[new_i][new_j] != m2[k]) {
            return false;
        }
    }
    return true;
}

int countoccurrences(vector<vector<int> >& m1, vector<int>& m2) {
    int count = 0;
    int rows = m1.size();
    int cols = m1[0].size();

    vector<direction> directions;
    directions.push_back(direction(0, 1));
    directions.push_back(direction(0, -1));
    directions.push_back(direction(1, 0));
    directions.push_back(direction(-1, 0));
    directions.push_back(direction(1, 1));
    directions.push_back(direction(-1, -1));
    directions.push_back(direction(1, -1));
    directions.push_back(direction(-1, 1));

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            for (int d = 0; d < directions.size(); d++) {
                int di = directions[d].x;
                int dj = directions[d].y;
                if (checksequence(m1, i, j, m2, di, dj)) {
                    count++;
                }
            }
        }
    }

    return count;
}

int main() {
    int rows, cols, m2size;

    cout << "enter the number of rows and columns of matrix m1: ";
    cin >> rows >> cols;

    vector<vector<int> > m1(rows, vector<int>(cols));

    cout << "enter the elements of matrix m1: " << endl;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cin >> m1[i][j];
        }
    }

    cout << "enter the size of the sequence m2: ";
    cin >> m2size;
    vector<int> m2(m2size);


    cout << "enter the elements of sequence m2: ";
    for (int i = 0; i < m2size; i++) {
        cin >> m2[i];
    }


    int result = countoccurrences(m1, m2);
    cout << "the sequence appears " << result << " times." << endl;


    return 0;
}
```



```
Enter the number of rows and columns of matrix M1: 10 10
Enter the elements of matrix M1:
11 12 13 23 24 25 14 15 16 17
21 22 23 26 24 27 28 29 23 8
32 31 24 33 35 25 36 37 24 38
41 42 25 43 44 23 45 46 25 47
52 53 24 55 56 24 57 58 59 51
61 62 63 23 25 64 65 66 67 68
72 73 74 25 24 23 75 76 77 78
23 82 83 84 85 25 85 86 87 25
24 91 92 93 94 95 96 97 24 99
25 23 24 25 18 19 20 23 98 23
Enter the size of the sequence M2: 3
Enter the elements of sequence M2: 23 24 25
The sequence appears 9 times.
```

6. Direct the Player out of the Maze

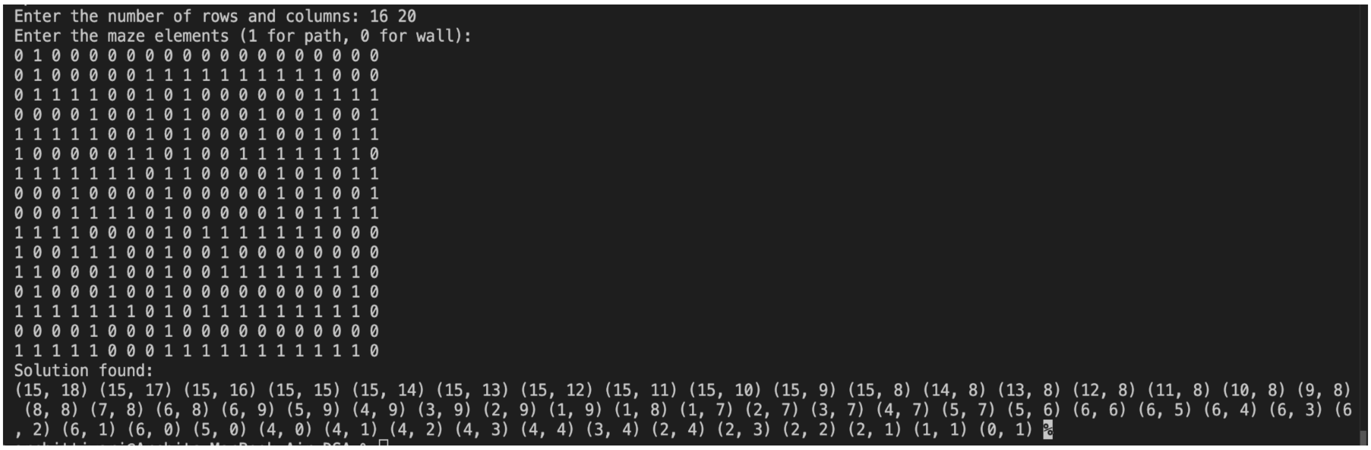


0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0
0	1	1	1	1	0	0	1	0	1	0	0	0	0	0	0	1	1	1	1
0	0	0	0	1	0	0	1	0	1	0	0	0	1	0	0	1	0	0	1
1	1	1	1	1	0	0	1	0	1	0	0	0	1	0	0	1	0	1	1
1	0	0	0	0	0	1	1	0	1	0	0	1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0	1	1	0	0	0	0	1	0	1	0	1	1
0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0	1
0	0	0	1	1	1	1	0	1	0	0	0	0	0	1	0	1	1	1	1
1	1	1	1	0	0	0	0	1	0	1	1	1	1	1	1	1	0	0	0
1	0	0	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0
1	1	0	0	0	1	0	0	1	0	0	1	1	1	1	1	1	1	1	0
0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0
1	1	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	0
0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0

1 represents a leeway, whereas 0 represents a wall.

Ans:

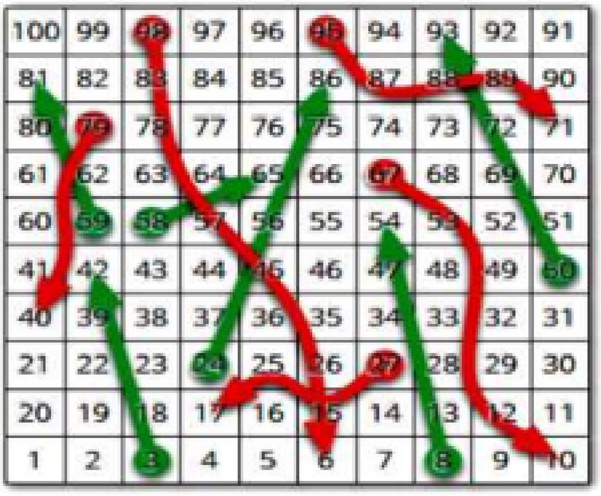
```
#include <iostream>
#include <stack>

using namespace std;

struct Position {
    int x, y;
};

// Function to check if a position is valid
bool isValid(int x, int y, int rows, int cols, int maze[][100], bool visited[][100]) {
    return x >= 0 && x < rows && y >= 0 && y < cols && maze[x][y] == 1 && !visited[x][y];
}

// Function to solve the maze using DFS
bool solveMaze(Position current, Position& exit, stack<Position>& path, int maze[][100], bool
visited[][100], int rows, int cols) {
    // Check if the current position is the exit
    if (current.x == exit.x && current.y == exit.y) {
        path.push(current);
        return true;
    }
    // Mark the current position as visited
    visited[current.x][current.y] = true;
    path.push(current);

    // Explore all possible directions (up, down, left, right)
    Position directions[] = {{current.x - 1, current.y}, {current.x + 1, current.y}, {current.x, current.y - 1}, {current.x, current.y + 1}};
    for (int i = 0; i < 4; i++) {
        if (isValid(directions[i].x, directions[i].y, rows, cols, maze, visited)) {
            if (solveMaze(directions[i], exit, path, maze, visited, rows, cols)) {
                return true;
            }
        }
    }

    // If no path found, backtrack
    visited[current.x][current.y] = false;
    path.pop();
    return false;
}

int main() {
    int rows, cols;
    cout << "Enter the number of rows and columns: ";
    cin >> rows >> cols;

    int maze[100][100];
    bool visited[100][100] = {false};

    cout << "Enter the maze elements (1 for path, 0 for wall):" << endl;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cin >> maze[i][j];
        }
    }

    // Starting position and exit position
    Position start = {0, 1};
    Position exit = {rows - 1, cols - 2};

    stack<Position> path;
    if (solveMaze(start, exit, path, maze, visited, rows, cols)) {
        cout << "Solution found:" << endl;
        while (!path.empty()) {
            Position pos = path.top();
            path.pop();
            cout << "(" << pos.x << ", " << pos.y << ") ";
        }
    } else {
        cout << "No solution found." << endl;
    }

    return 0;
}
```



```
Enter the number of rows and columns: 16 20
Enter the maze elements (1 for path, 0 for wall):
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 1 1 1 1 1 1 1 1 1 1 0 0 0
0 1 1 1 1 0 0 1 0 1 0 0 0 0 0 0 1 1 1 1
0 0 0 0 1 0 0 1 0 1 0 0 0 1 0 0 1 0 0 1
1 1 1 1 1 0 0 1 0 1 0 0 0 1 0 0 1 0 1 1
1 0 0 0 0 0 1 1 0 1 0 0 1 1 1 1 1 1 1 0
1 1 1 1 1 1 1 0 1 1 0 0 0 0 1 0 1 0 1 1
0 0 0 1 0 0 0 0 1 0 0 0 0 0 1 0 1 0 0 1
0 0 0 1 1 1 1 0 1 0 0 0 0 0 1 0 1 1 1 1
1 1 1 1 0 0 0 0 1 0 1 1 1 1 1 1 1 0 0 0
1 0 0 1 1 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0
1 1 0 0 0 1 0 0 1 0 0 1 1 1 1 1 1 1 1 0
0 1 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 1 0
1 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 0
0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0
Solution found:
(15, 18) (15, 17) (15, 16) (15, 15) (15, 14) (15, 13) (15, 12) (15, 11) (15, 10) (15, 9) (15, 8) (14, 8) (13, 8) (12, 8) (11, 8) (10, 8) (9, 8)
 (8, 8) (7, 8) (6, 8) (6, 9) (5, 9) (4, 9) (3, 9) (2, 9) (1, 9) (1, 8) (1, 7) (2, 7) (3, 7) (4, 7) (5, 7) (5, 6) (6, 6) (6, 5) (6, 4) (6, 3) (6
, 2) (6, 1) (6, 0) (5, 0) (4, 0) (4, 1) (4, 2) (4, 3) (4, 4) (3, 4) (2, 4) (2, 3) (2, 2) (2, 1) (1, 1) (0, 1) %
```

7. Snakes and Ladders is a classic board game, originating in India no later than the 16th century. The board consists of an *n* × *n* grid of squares, numbered consecutively from 1 to *n*^{2}, starting in the bottom left corner and proceeding row by row from bottom to top, with rows alternating to the left and right. Certain pairs of squares in this grid, always in different rows, are connected by either “snakes” (leading down) or “ladders” (leading up). Each square can be an endpoint of at most one snake or ladder, as presented in given figure.

You start with a token in cell 1, in the bottom left corner. In each move, you advance your token up to *k* positions, for some fixed constant *k*. If the token ends the move at the *top* end of a snake, it slides down to the bottom of that snake. Similarly, if the token ends the move at the *bottom* end of a ladder, it climbs up to the top of that ladder. It is required to compute the smallest number of moves required for the token to reach the last square of the grid.

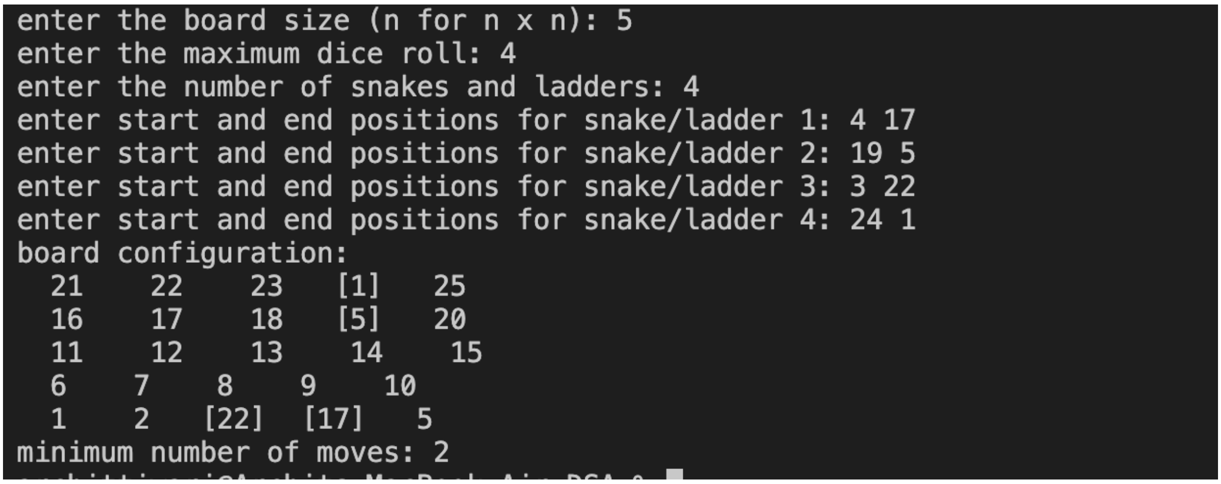


A typical Snakes and Ladders board.
Upward straight arrows are ladders; downward wavy arrows are snakes.

Ans:

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

using namespace std;

struct snakeladder {
    int start, end;
    snakeladder(int start, int end) : start(start), end(end) {}
};

const int no_snake_ladder = -1;

int solve_snakes_and_ladders(const vector<snakeladder>& snakesandladders, int n, int k) {
    vector<int> board(n * n, no_snake_ladder);

    for (const snakeladder& snake_ladder : snakesandladders) {
        board[snake_ladder.start - 1] = snake_ladder.end - 1;
    }

    queue<int> q;
    q.push(0);
    vector<bool> visited(n * n, false);
    visited[0] = true;

    int moves = 0;
    while (!q.empty()) {
        int size = q.size();
        while (size--) {
            int current = q.front();
            q.pop();

            if (current == n * n - 1) {
                return moves;
            }

            for (int i = 1; i <= k; i++) {
                int next = current + i;
                if (next >= n * n) {
                    break;
                }

                if (!visited[next]) {
                    int jumpto = board[next];
                    if (jumpto != no_snake_ladder) {
                        next = jumpto;
                    }

                    visited[next] = true;
                    q.push(next);
                }
            }
        }

        moves++;
    }

    return -1;
}
```

```
void print_board(int n, const vector<snakeladder>& snakesandladders) {
    vector<int> board(n * n, 0);

    for (int i = 0; i < n * n; i++) {
        board[i] = i + 1;
    }

    for (const snakeladder& snake_ladder : snakesandladders) {
        board[snake_ladder.start - 1] = -snake_ladder.end;
    }

    for (int i = n - 1; i >= 0; i--) {
        for (int j = 0; j < n; j++) {
            int index = i * n + j;
            if (board[index] < 0) {
                cout << " [" << -board[index] << "] ";
            } else {
                cout << "  " << board[index] << "  ";
            }
        }
        cout << endl;
    }
}

int main() {
    int n;
    cout << "enter the board size (n for n x n): ";
    cin >> n;

    int k;
    cout << "enter the maximum dice roll: ";
    cin >> k;

    int num_snakes_ladders;
    cout << "enter the number of snakes and ladders: ";
    cin >> num_snakes_ladders;

    vector<snakeladder> snakesandladders;

    for (int i = 0; i < num_snakes_ladders; i++) {
        int start, end;
        cout << "enter start and end positions for snake/ladder " << i + 1 << ": ";
        cin >> start >> end;

        if (start < 1 || start > n * n || end < 1 || end > n * n) {
            cout << "invalid positions. please enter values between 1 and " << n * n << "." << endl;
            i--;
            continue;
        }
        snakesandladders.push_back(snakeladder(start, end));
    }

    cout << "board configuration:" << endl;
    print_board(n, snakesandladders);

    int moves = solve_snakes_and_ladders(snakesandladders, n, k);
    if (moves != -1) {
        cout << "minimum number of moves: " << moves << endl;
    } else {
        cout << "no solution found." << endl;
    }

    return 0;
}
```



```
enter the board size (n for n x n): 5
enter the maximum dice roll: 4
enter the number of snakes and ladders: 4
enter start and end positions for snake/ladder 1: 4 17
enter start and end positions for snake/ladder 2: 19 5
enter start and end positions for snake/ladder 3: 3 22
enter start and end positions for snake/ladder 4: 24 1
board configuration:
   21    22    23   [1]   25
   16    17    18   [5]   20
   11    12    13    14    15
   6    7    8    9    10
   1    2   [22]  [17]   5
minimum number of moves: 2
```