CS101 – Computer Programming
Practice Problems on Arrays

Problem1: 2D Maze

You are given a maze represented using a 2 dimensional array of size n x n. A value of 1 for the maze[i][j]th element implies path exists and 0 implies no path. The goal destination is to reach the n-1, n-1 cell in the maze.

For example, the following is a valid maze matrix.

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

You are also given a path represented using a 1-D array that specifies the sequence of steps that needs to be taken at each stage and the following is the format used.

Up = 1
Right = 2
Down = 3
Left = 4

Once the end is reached all subsequent elements of the path matrix are 0.

So for the above maze a valid path matrix would be:

path = {2,3,3,2,1,2,3,3,0,0,0,0,0,0,0,0};

The program is divided into two parts.

Part1 : Validate the path array

The initial check is trivial. You want to ensure that all elements have values 1-4 and if an element is 0 all subsequent elements are 0.

The further checks are:

i) Check for horizontal and vertical path overlaps
ii) Check if the path ends at (n-1,n-1)
iii) Check if the path stays within the dimensions of the matrix i.e. x and y coordinates must have values within 0 to n-1 at all points

Part2: Check if the given path array satisfies the given maze matrix

Complete the following program by filling in the code snippets. Take the value of n to be 10.

Code Snippet

```
#include<iostream>
using namespace std;
int main() {
        int maze[10][10];
        int path[100];

        //code to get the maze matrix
        code_snippet1

        //code to get the path array
        code_snippet2

        //code to validate the path array
        /* ensure valid and continuous moves
         * ensure no overlaps
         * ensure no overflows
         * ensure termination at 9,9
         */
        code_snippet3

        //code to validate path against the maze
        /* Think!
         * Clue : each move must be valid against
                the maze
         */
        code_snippet4

        return 0;
}
```

Problem 2: Game of Life

Game of Life is a **n x n** two-dimensional array, each of whose cell is in one of two possible states, *alive (1)* or *dead(0)*. Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

- Any live cell with fewer than two live neighbours dies, as if caused by under-population.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overcrowding.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

Example:

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

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

Question 1 (Manual)

To understand the above generation draw the next states that can be produced from the present state? (Check your answer on the next page)

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

Question 2:

Write a C++ code to produce the next generation given the current using the above rules for an 10x10 matrix. You can use the code framework given in the next page.

Code Sinppet (Game of Life)

```
#include<iostream>
using namespace std;
int main() {
        int matrix1[10][10]; //input matrix
        int matrix2[10][10]; //next generation output matrix

        //code to get the matrix
        code_snippet1

        //code to initialize the output matrix
        code_snippet2

        //code to generate the next generation
        /* iterate through each cell
         *for each cell check if it alive based
                on the above rules
         *also check for boundary conditions
         */
        code_snippet3

        //code to output the next generation
        code_snippet4

        return 0;
}
```

Optional : Modify the above code to generate the next n generations of life! Hint : loops!

Problem3: Compute Lengths of Blocks!

Ram has 10 rectangular blocks of varying lengths (> 1 unit) arranged in a 10 x 10 matrix either horizontally or vertically. All blocks are of width 1 unit. Each block has a unique label from 0 – 9, and every 1 unit x 1 unit square on a particular block is marked with the same label. Thus we have 10 different blocks lying over a 10 x 10 matrix. One such example is shown below.

-1	-1	**0**	**0**	**0**	-1	**2**	-1	-1	-1
1	**1**	**1**	-1	-1	-1	**2**	-1	-1	-1
-1	**9**	-1	-1	-1	-1	**2**	-1	6	6
-1	**9**	**4**	**3**	**3**	-1	**2**	-1	5	5
-1	**9**	**4**	-1	-1	**7**	**7**	**7**	**7**	-1
-1	**9**	**4**	-1	-1	**8**	-1	-1	-1	-1
-1	**9**	**4**	-1	-1	**8**	-1	-1	-1	-1
-1	**9**	**4**	-1	-1	**8**	-1	-1	-1	-1
-1	**9**	**4**	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

As shown above, the cells that are empty are denoted using -1. The cells that are occuppied by a block are marked with the corresponding label of the block.

Ram wants to find out the following by writing code in C++:

1. The sizes of all horizontal blocks.
2. The number of vertical blocks.

On the next page is given a code framework. Fill in code_snippets 1, 2 and 3

Code Snippet (Compute Lengths of Blocks!)

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

int main() {
        int matrix[10][10];
        //2d array to store matrix
        //code to take the 2d array as input
        code_snippet1

        int total_size = 0;
        //code to find the size of all horizontal
                blocks
        /* outer for loop that runs for all rows
         * inner for loop that counts each valid
                label
         * label is a valid horizontal label if next
                label
         * is not same as current label
         */
        code_snippet2

        int no_of_vblocks = 0;
        //code to count number of vertical blocks
        /* Think! Looping over columns is the
                clue! */
        code_snippet3

        cout << "total size of all horizontal
                blocks" << total_size;
        cout << "number of vertical blocks" <<
                no_of_vblocks;
        return 0;
}
```