**W4 -** PRACTICE

*2D Arrays – Problem Solving*

## *At the end of this practice, you should be able to…*

* Practice **array manipulations** with functions
* Consolidate **Top-Down Design** and function-based refactoring applied to 2D arrays
* Apply **scope**, **memory**, and **function call** concepts

## *How do we structure exercises?*

We organize this practice into 4 parts:

| GAME | Play a serious game to **grasp the concepts** |
| --- | --- |
| ANALYSE | **Understand** existing codes, find the **bugs** or **complete** missing gaps |
| MANIPULATE | Ensure you can **apply the theory** with some basic challenges |
| CREATE | **Express your creativity** with more complex challenges |

## *Are you lost?*

You can read the following documentation to be ready for this practice

<https://pseudocode.deepjain.com/guides/functions/>

<https://www.w3schools.com/c/c_functions.php>

**GAME**

**RESCUE CODE: THE 2D ARRAY QUEST**

****

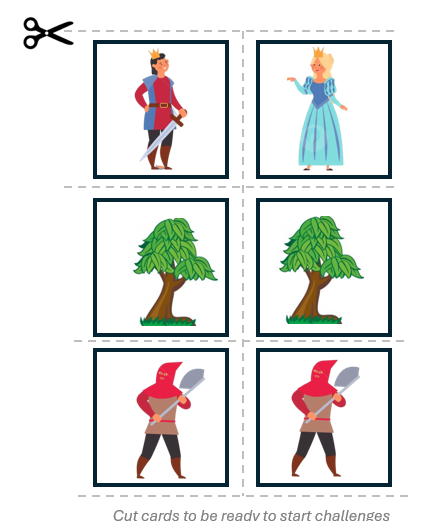
*Teams of 3–4 players ! 4 Challenges*

**Before starting:**

* Cut the cards appropriately to place them on the board

**For each challenge**

* Place the initial cards on the 5×5 board following the setup.
* Execute the code manually and update the board after each action
* Discuss as a group before revealing the correct answer on the whiteboard !



**ANALYSIS**

**EX 1** **(Browse a 2D array)**

Look at the bellow code.

int main() {

int matrix[3][2] = {

{15, 5},

{7, 5},

{6, 0},

};

int result[6] = {0};

int resultIndex = 0;

for (int column = 0; column < 2; column++) {

for (int row = 0; row < 3; row++) {

result[resultIndex] = matrix[row][column];

resultIndex++;

}

}

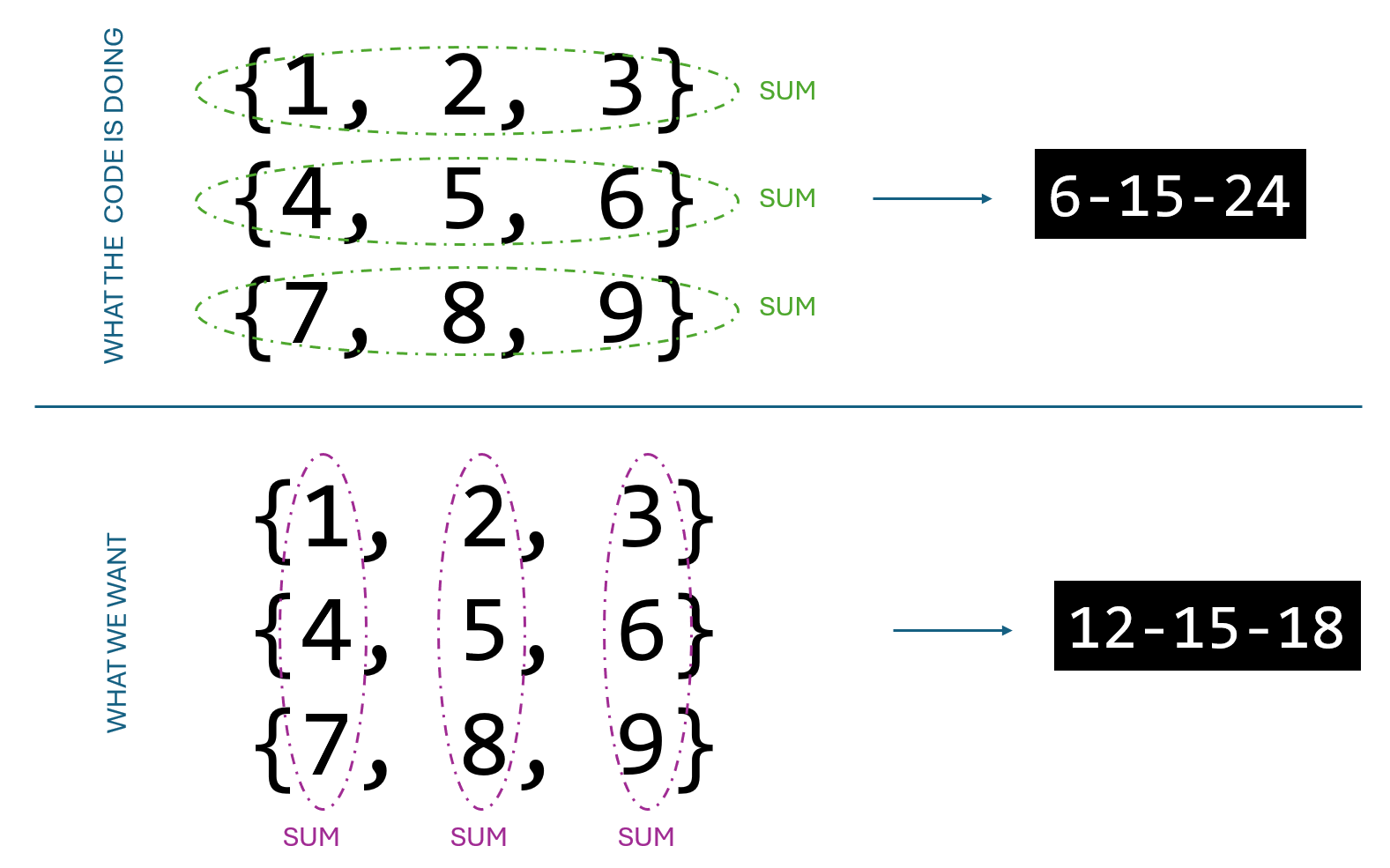
return 0;

}

**Q1 -** Complete the bellow table, by specifying the variable state - for **every step** of the execution

| STEP | Row | Column | resultIndex | result |
| --- | --- | --- | --- | --- |
| BEFORE LOOP | | | | |
| 1 | ? | ? | ? | {0, 0, 0, 0, 0, 0} |
| DURING LOOP | | | | |
| 2 | 0 | 0 | 0 | {15, 0, 0, 0, 0, 0} |
| 3 | 0 | 0 | 1 | {15, 0, 0, 0, 0, 0} |
| 4 | 1` | 0 | 1 | {15, 0, 0, 0, 0, 0} |
| 5 | 1 | 0 | 1 | {15, 7, 0, 0, 0, 0} |
| 6 | 1 | 0 | 2 | {15, 7, 0, 0, 0, 0} |
| 7 | 2 | 0 | 2 | {15, 7, 0, 0, 0, 0} |
| 8 | 2 | 0 | 2 | {15, 7, 6, 0, 0, 0} |
| 9 | 0 | 1 | 2 | {15, 7, 6, 0, 0, 0} |
| 10 | 0 | 1 | 3 | {15, 7, 6, 0, 0, 0} |
| 11 | 0 | 1 | 3 | {15, 7, 6, 5, 0, 0} |
| 12 | 1 | 1 | 4 | {15, 7, 6, 5, 0, 0} |
| 13 | 1 | 1 | 4 | {15, 7, 6, 5, 5, 0} |
| 14 | 2 | 1 | 4 | {15, 7, 6, 5, 5, 0} |
| 15 | 2 | 1 | 4 | {15, 7, 6, 5, 5, 0} |

**EX 2** **(Adapt a code to a new problem)**



In this exercise, you need to adapt the below code to display the **sum of each column** (instead of the **sum of each row**).

#include <stdio.h>

int main() {

  int matrix[3][3] = {

      {1, 2, 3},

      {4, 5, 6},

      {7, 8, 9},

  };

  for (int row = 0; row < 3; row++) {

    int sumRow =0;

    for (int column = 0; column < 3; column++) {

      sumRow+=matrix[row][column];

    }

    printf("%d - ", sumRow);

  }

  return 0;

}

**Q1 -** If you want to sum a column: which index **should stay constant**, and which **should change** in the loop?

**The column index should stay constant until we finish summing every row in the column.**

**Q2 -** How do you access **all** **elements of a column**?

**By increment the row by 1 until it’s done**

**Q3 –** Provide the **adapted code**

#include <stdio.h>

int main() {

  int matrix[3][3] = {

      {1, 2, 3},

      {4, 5, 6},

      {7, 8, 9},

  };

  for (int column = 0; column < 3; column++) {

    int sumColumn =0;

    for (int row = 0; row < 3; row++) {

      sumColumn+=matrix[row][column];

    }

    printf("%d - ", sumColumn);

  }

  return 0;

}

**EX 3 (Check if two arrays are equal – PSEUDO CODE ONLY)**

In this exercise, you need to check whether 2 arrays 2D are equal or not.

*Note : we assume we work on 3X3 dimensions arrays.*

| INPUTS | First array 2D  Second array 2D |
| --- | --- |
| OUTPUT | Equal / No Equal |

Examples

| INPUT | OUTPUT |
| --- | --- |
| { {1, 2, 3},        {4, 5, 6},        {7, 8, 9} }       { {1, 2, 3},        {4, 5, 6},        {7, 8, 9} } | Equal |
| { {1, 2, 3},        {4, 5, 6},        {7, 8, 2} }       { {1, 2, 3},        {4, 5, 6},        {7, 8, 9} } | Not equal |

**Q1 –** Write the **PSEUDO CODE** to solve this problem

You can use the following blocks:

INPUT <variable>

SET <variable> TO <value>

FOR <number> FROM <start> TO <end>

WHILE <condition>

IF <condition>

PRINT

PSEUDO CODE:

input 2 2D arrays

set notEqual to 0;

for <column> from <1> to <3>

for <row> from <1> to <3>

if<array1[row][column] != array2[row][column]>

print <not equal>;

set notEqual to 1;

if <notEqual == 0>

print <equal>;

int maxAnd = 0, maxOr = 0, maxXor = 0;

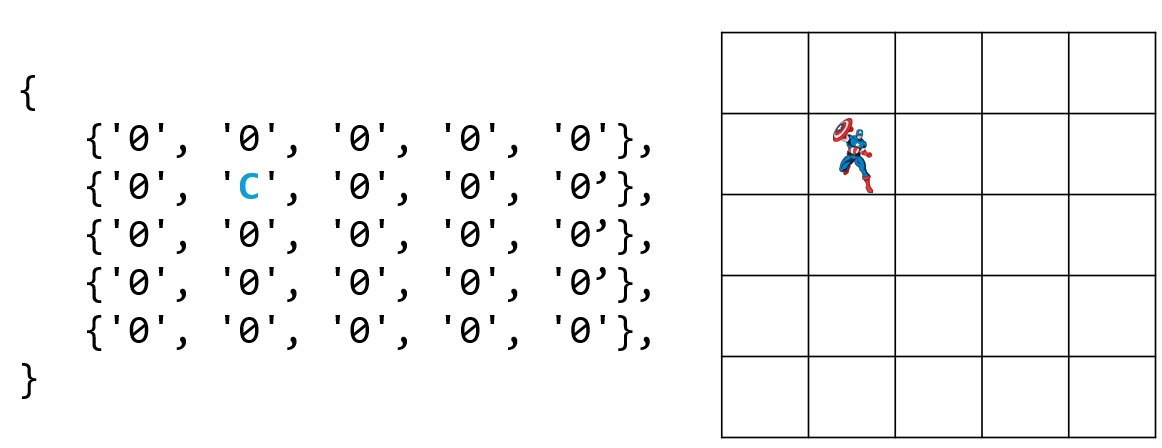
**MANIPULATE**

**EX 1 (MOVE CAPTAIN AMERICA)**

****

*You are building a game on console where Captain America is represented within a grid of 5X5 cells.*

* Each cell has a character which can be ‘’0” (no captain) or “C” (captain position)
* Your goal is to implement the functions to move the captain **UP, DOWN, RIGHT, LEFT**



WARNING : The captain cannot go outside the grid dimensions !

**Q1 – Move the captain !**

Implement the function **moveCaptain** as follows:

| FUNCTION NAME | moveCaptain | |
| --- | --- | --- |
| FUNCTION DESCRIPTION | Update the grid with the requested move (up, down ,left, right) | |
| PARAMETERS | char [5][5] | The game grid |
| char | Move types : UP, DOWN, LEFT, RIGHT |
| RETURN | Bool | True if the move was performed  *(if captain is on border sides, nothing is done, return false)* |

*EXAMPLES*

| PARAMETERS | UPDATED ? |
| --- | --- |
| {  {'0', '0', '0', '0', '0'},  {'0', '**C**', '0', '0', '0’},  {'0', '0', '0', '0', '0’},  {'0', '0', '0', '0', '0’},  {'0', '0', '0', '0', '0'},  }  DOWN | Yes  {  {'0', '0', '0', '0', '0'},  {'0', '0', '0', '0', '0’},  {'0', '**C**', '0', '0', '0’},  {'0', '0', '0', '0', '0’},  {'0', '0', '0', '0', '0'},  } |
| {  {'0', '0', '0', '0', '0'},  {'0', '0', '0', '0', '0’},  {'0', '0', '0', '0', '0’},  {'0', '0', '0', '0', '0’},  {'0', '**C**', '0', '0', '0'},  }  DOWN | *No - Nothing updated, as the captain cannot go down* |

#include <stdio.h>

#include <stdbool.h>

bool moveCaptain(char gameGrid[5][5], char move){

int rowC = 0, columnC = 0;

char canMove = false;

bool notfound = true ;

for(int row = 0; row < 5 && notfound; row++){

for(int column = 0; column < 5 && notfound; column++){

if(gameGrid[row][column] == 'C') {

rowC = row;

columnC = column;

}

}

}

if (move == 'u' && rowC - 1 >= 0) {

gameGrid[rowC - 1][columnC] = 'C';

gameGrid[rowC][columnC] = '0';

canMove = true;

} else if (move == 'd' && rowC + 1 <= 5) {

gameGrid[rowC + 1][columnC] = 'C';

gameGrid[rowC][columnC] = '0';

canMove = true;

} else if (move == 'r' && columnC + 1 >= 0) {

gameGrid[rowC][columnC + 1] = 'C';

gameGrid[rowC][columnC] = '0';

canMove = true;

} else if (move == 'l' && columnC - 1 <= 5) {

gameGrid[rowC][columnC + 1] = 'C';

gameGrid[rowC][columnC] = '0';

canMove = true;

}

return canMove;

}

void updateGrid(char gameGrid[5][5]){

printf("yes\n");

for(int row = 0; row < 5; row++){

for (int column = 0; column < 5; column++){

printf("%c ", gameGrid[row][column]);

}

printf("\n");

}

}

int main() {

char move;

bool canMove;

char gameGrid[5][5] = {

{'C', '0', '0', '0', '0'},

{'0', '0', '0', '0', '0'},

{'0', '0', '0', '0', '0'},

{'0', '0', '0', '0', '0'},

{'0', '0', '0', '0', '0'},

};

printf("(u)p\t(d)own\t(r)ight\t(l)eft\nEnter your move : ");

scanf(" %c", &move);

canMove = moveCaptain(gameGrid, move);

if (canMove) updateGrid(gameGrid);

else printf("No\n");

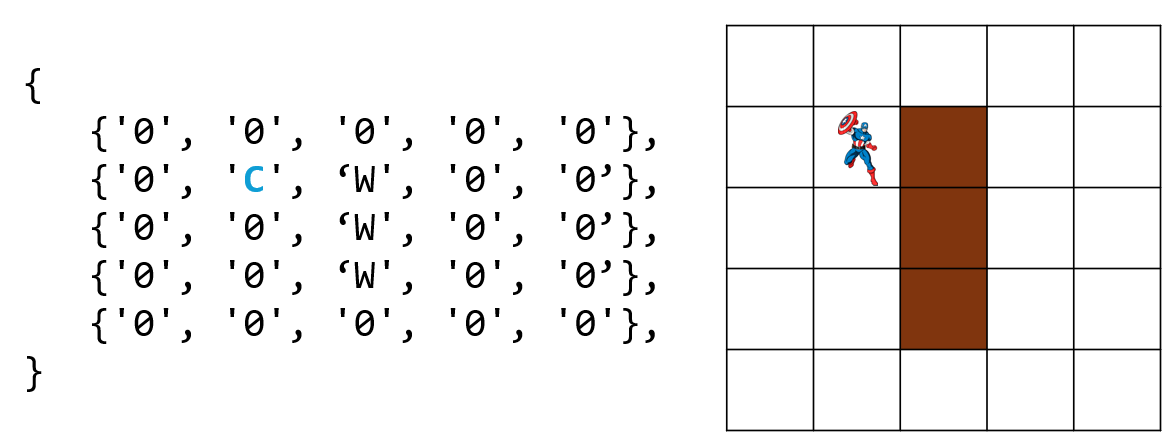
return 0;

}

**Q2- Manage the walls !!**

* Now each cell can have another a character “W” which represents a wall.
* The captain cannot move on walls !

Your job is to update the previous function to handle the walls in the game



**#include <stdio.h>**

**#include <stdbool.h>**

**bool moveCaptain(char gameGrid[5][5], char move){**

**int rowC = 0, columnC = 0;**

**char canMove = false;**

**bool notfound = true ;**

**for(int row = 0; row < 5 && notfound; row++){**

**for(int column = 0; column < 5 && notfound; column++){**

**if(gameGrid[row][column] == 'C') {**

**rowC = row;**

**columnC = column;**

**}**

**}**

**}**

**if (move == 'u' & v4565& rowC - 1 >= 0 && gameGrid[rowC - 1][columnC] != 'W') {**

**gameGrid[rowC - 1][columnC] = 'C';**

**gameGrid[rowC][columnC] = '0';**

**canMove = true;**

**} else if (move == 'd' && rowC + 1 <= 5 && gameGrid[rowC + 1][columnC] != 'W') {**

**gameGrid[rowC + 1][columnC] = 'C';**

**gameGrid[rowC][columnC] = '0';**

**canMove = true;**

**} else if (move == 'r' && columnC + 1 <= 5 && gameGrid[rowC][columnC + 1] != 'W') {**

**gameGrid[rowC][columnC + 1] = 'C';**

**gameGrid[rowC][columnC] = '0';**

**canMove = true;**

**} else if (move == 'l' && columnC - 1 >= 0 && gameGrid[rowC][columnC - 1] != 'W') {**

**gameGrid[rowC][columnC + 1] = 'C';**

**gameGrid[rowC][columnC] = '0';**

**canMove = true;**

**}**

**return canMove;**

**}**

**void updateGrid(char gameGrid[5][5]){**

**printf("yes\n");**

**for(int row = 0; row < 5; row++){**

**for (int column = 0; column < 5; column++){**

**printf("%c ", gameGrid[row][column]);**

**}**

**printf("\n");**

**}**

**}**

**int main() {**

**char move;**

**bool canMove;**

**char gameGrid[5][5] = {**

**{'C', 'W', '0', '0', '0'},**

**{'0', '0', '0', '0', '0'},**

**{'0', '0', '0', '0', '0'},**

**{'0', '0', '0', '0', '0'},**

**{'0', '0', '0', '0', '0'},**

**};**

**printf("(u)p\t(d)own\t(r)ight\t(l)eft\nEnter your move : ");**

**scanf(" %c", &move);**

**canMove = moveCaptain(gameGrid, move);**

**if (canMove) updateGrid(gameGrid);**

**else printf("No\n");**

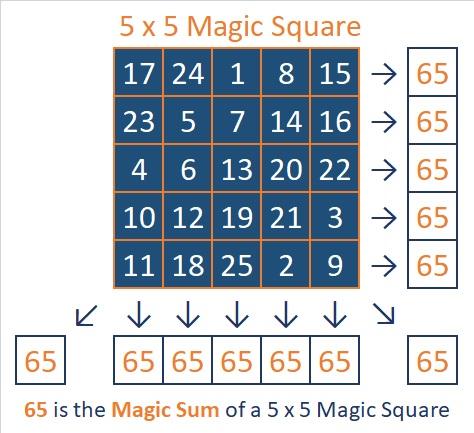
**return 0;**

**}**

**EX 2 (IS THIS A MAGIC SQUARE ?)**

A magic square of size 5×5 is a grid where:

* Each row sums to 65.
* Each column sums to 65.
* Both main diagonals (top-left to bottom-right and top-right to bottom-left) sum to 65.
* All numbers from 1 to 25 appear exactly once



For this exercise, you are provided with a start code:

* Note that this code introduces CONSTANTS values (#define <CONSTANT> <VALUE> )
* We will come back to this syntax in the coming weeks

#include <stdio.h>

#include <stdbool.h>

#define N 5

#define MAGIC\_SUM 65

bool **isMagicSquare**(int size, int grid[N][N]) {

return true; // YOUR JOB IS HERE !

}

int main() {

int square[N][N] = {

{17, 24, 1, 8, 15},

{23, 5, 7, 14, 16},

{4, 6, 13, 20, 22},

{10, 12, 19, 21, 3},

{11, 18, 25, 2, 9}

};

if (isMagicSquare(square)) {

printf("Valid magic square!\n");

} else {

printf("Not a valid magic square.\n");

}

return 0;

}

**Q1- Reflective questions**

Q1.1 - What does it mean for a row/column to be “valid” in a magic square?

for a row/column to be valid in a magic square the sum of row/column has to be 65.

Q1.2 - How do you check **diagonals**?

i start by summing the first row and the last column and continuously -1 from the var column and +1 to the var row

Q1.3 - How do you ensure all numbers from 1 to 25 appear **exactly once**?

*Tips: What about another array of 25 boolean?*

i use loop in loop to compare each elements make sure that its not the same.

**Q2- Top-Down Design Strategy**

To divide the job into small tasks, we have identified 4 sub functions.

Complete the table with the 2 missing functions

| FUNCTION | DESCRIPTION | INTPUT | OUTPUT |
| --- | --- | --- | --- |
| check Unique numbers | Check all numbers are unique in the grid | Grid[N][N] | bool |
| checkMainDiagonals | Check the 2 diagonals have their sum = 65 | Grid[N][N] | bool |
| checkSumOfRows | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Grid[N][N] | bool |
| checkSumOfColumns | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Grid[N][N] | bool |

**Q3- implement the code**

Once you are clear with your top-down design functions, implement the code and test it using different test cases.

#include <stdio.h>

#include <stdbool.h>

#define N 5

#define MAGIC\_SUM 65

bool checkUniqeNumber(int grid[N][N]){

bool same = false;

for (int n = 0; n < N ; n++){

for (int m = 0; m < N; m++){

for(int row = 0; row < N; row++){

for(int column = 0; column < N; column++){

if (grid[row][column] == grid[n][m]) same = true;

}

}

}

}

return same;

}

bool sumRows(int grid[N][N]){

bool sumRow = true;

for (int row = 0; row < N && sumRow; row++){

int sum = 0;

for (int column = 0 ; column < N && sumRow; column++){

sum += grid[row][column];

}

if (sum != MAGIC\_SUM ) sumRow = false;

}

return sumRow;

}

bool sumColumn(int grid[N][N]){

bool sumColumn = true;

for (int column = 0; column < N && sumColumn; column++){

int sum = 0;

for (int row = 0 ; row < N && sumColumn; row++){

sum += grid[row][column];

}

if (sum != MAGIC\_SUM ) sumColumn = false;

}

return sumColumn;

}

bool sumDiagonal(int grid[N][N]){

bool sumDiagonal = true;

int sumDiagonal1 = 0, sumDiagonal2 = 0;

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

sumDiagonal1 += grid[i][i];

sumDiagonal2 += grid[i][4 - i];

}

if (sumDiagonal1 != MAGIC\_SUM || sumDiagonal2 != MAGIC\_SUM ) sumDiagonal = false;

return sumDiagonal;

}

bool isMagicSquare( int grid[N][N]) {

// YOUR JOB IS HERE !

bool magicSum = false;

if (checkUniqeNumber(grid) && sumRows(grid) && sumColumn(grid) && sumDiagonal(grid)) magicSum = true;

return magicSum;

}

int main() {

int square[N][N] = {

{17, 24, 1, 8, 15},

{23, 5, 7, 14, 16},

{4, 6, 13, 20, 22},

{10, 12, 19, 21, 3},

{11, 18, 25, 2, 9}

};

if (isMagicSquare( square)) {

printf("Valid magic square!\n");

} else {

printf("Not a valid magic square.\n");

}

return 0;

}