TREASURE HUNT

PROBLEM DEFINITION:

The aim of the project is to find the hidden treasure within a treasure map with the help of a key, where both the map and key are provided as matrices in .txt files. The key need to be moved over the map and each time, the local product between both the matrices gives us a clue regarding the next move, which is one of up, down, left or right. This procedure is repeated until the treasure is found.

METHODS AND SOLUTION:

The source code is divided into three functions:

1. main()
2. get_sub_map()
3. get_product()

The **main()** function takes care of:

1. Processing of input arguments.
    a. There will be five input arguments to the executable, namely:
        i. Map dimensions. Eg: 12x18
        ii. Key size. Eg: 3
        iii. Map matrix. Eg: mapmatrix.txt
        iv. Key matrix. Eg: keymatrix.txt
        v. Output directions to the treasure. Eg: output.txt

```
$ ./code 12x18 3 mapmatrix.txt keymatrix.txt output.txt
```

    The above line is the command line execution of the executable 'code'. For processing the map dimensions "12x18", we follow this procedure:

    b. We load the first argument into a char* and then iterating through that char*, we find the index where 'x' is located like this:

```
for (i = 0; i < len; ++i) {
    if (map_size[i] == 'x') {
        x_indx = i;
        break;
    }
}
```

    c. Then we use strncpy() to copy over the 12 into map_dimen[0] and for the other dimension i.e., 18, we iterate from x_indx + 1 upto the end of the map dimension string "12x18" to extract the dimension 18 and copy it over to map_dimen[1]. So, now the map_dimen = {12, 18}.

2. Initialization of map and key matrices.
    a. We load the files using file pointers in C.
    b. Allocate memory for the map and key matrices using malloc().

```
map = (int**)malloc(map_dimen[0] * sizeof(int*));
for (i = 0; i < map_dimen[0]; ++i)
    map[i] = (int*)malloc(map_dimen[1] * sizeof(int));
```

This will allocate the memory for the 2D array, map. Similarly, we allocate memory for the key matrix.

3. Populating the map and key matrices from the input mapmatrix.txt and keymatrix.txt files.
    a. We loop over the file using feof(file_pointer), which basically checks for the EOF (end-of-file), and the we scan in, using fscanf(), each integer into the map matrix.

```
while (!feof(fp_map)) {
    for (i = 0; i < map_dimen[0]; ++i) {
        for (j = 0; j < map_dimen[1]; ++j) {
            fscanf(fp_map, "%d ", &var);
            map[i][j] = var;
        }
    }
}
```

4. Parts 5-8 will be a while loop, which exits when the treasure is found.
5. Calling get_sub_map() to get (key_size x key_size) cropped section from the map matrix, which will be explained below.

```
sub_map = get_sub_map(map, key_size, x, y);
```

6. Calling product() to get the product between the above sub_map return from get_sub_map() function and the key matrix to get element-wise product matrix, which will be explained below.

```
product = get_product(sub_map, key, key_size);
```

7. Summing up the elements of the product matrix obtained above and then take the modulus with 5.

```
int sum = 0;
for (i = 0; i < key_size; ++i) {
    for (j = 0; j < key_size; ++j) {
        sum += product[i][j];
    }
}
int mod = sum % 5;
```

8. Depending on the value of the modulus, the move is decided – using if-else conditions, which will be explained below.
    a. Moves:
        i. mod = 0 implies ‘treasure found’.
        ii. mod = 1 implies ‘go up’ or if failed to do so, ‘go down’
            `x = x - key_size; (for going up)`
        iii. mod = 2 implies ‘go down’ or if failed to do so, ‘go up’
            `x = x + key_size; (for going down)`
        iv. mod = 3 implies ‘go right’ or if failed to do so, ‘go left’
            `y = y + key_size; (for going right)`
        v. mod = 4 implies ‘go left’ or if failed to do so, ‘go right’

```
y = y - key_size; (for going left)
```

b. The failure conditions happen when the location of the sub_map goes beyond the border of the map matrix. The **valid conditions** are:

```
(x >= 0 && x <= (map_dimen[0] - key_size) &&
 y >= 0 && y <= (map_dimen[1] - key_size))
```

The **get_sub_map()** function takes care of:

1. Extracting (key_size x key_size) section from the original map matrix starting from a specified location (x, y).
2. The main logic of this function goes like this:
   a. Allocate memory for the sub_map using malloc
   b. Populate the sub_map using elements from map matrix like shown below:

```
for (i = x; i < x + key_size; ++i) {
    for (j = y; j < y + key_size; ++j) {
        sub_map[i - x][j - y] = map[i][j];
    }
}
```

The **get_product()** function takes care of:

1. Finding the element-wise product between the sub_map returned by the get_sub_map() function and the key_matrix. This is done as shown below:

```
for (i = 0; i < key_size; ++i) {
    for (j = 0; j < key_size; ++j) {
        product[i][j] = sub_map[i][j] * key[i][j];
    }
}
```

FUNCTIONS IMPLEMENTED:

1. Dynamic allocation using malloc() – the map and key matrices and all intermediate matrices and arrays use dynamic allocations.
2. Two dimensional arrays using double pointers (int**) – for map, key, sub_map and product matrices.
3. Necessary commenting in the source code.

FUNCTIONS NOT IMPLEMENTED:

1. Recursion – Implemented the core logic in an iterative fashion in while loop, which exits when the treasure is found.