Investigation 3 : Sherlock's a"Maze"ing Games

Submission instructions

At the end of the practical, your Git repository must follow this architecture:

```
csharp-tp3-firstname.lastname/
|-- README
|-- .gitignore
|-- Arrays101/
    |-- Arrays101.sln
    |-- Basics/
        |-- Arrays.cs
        |-- Basics.csproj
        |-- Program.cs
        |-- References.cs
    |-- Maze/
        |-- Maze.cs
        |-- Maze.csproj
        |-- Program.cs
```

Do not forget to check the following requirements before submitting your work:

- You shall obviously replace `firstname.lastname` with your login.
- The `README` file is mandatory.
- There must be no `bin` or `obj` folder in the repository.
- You must respect the prototypes of the given and asked functions.
- Remove all personal tests from your code.
- **The code MUST compile !**

README

In this file, you can write any and all comments you might have about the practical, your work, or more generally about your strengths and weaknesses. You must list and explain all the bonuses you have implemented. An empty `README` file will be considered as an invalid archive (malus).

1 Lesson

1.1 Memory, variables

When programming, and in particular in C# we declare a certain number of variables, functions. Where are they stored ? What happens when I declare something like `int a = 5;`. This value is stored in what we call *memory*. We are not going to dive deep into what's going on "under the hood" because it is a vast and complicated subject that would require an entire dedicated course. For the sake of simplicity, we are going to assume that what we refer to as memory is in fact an big table that holds values. For instance, let's represent the declaration and assignation `int a = 5;` in memory.

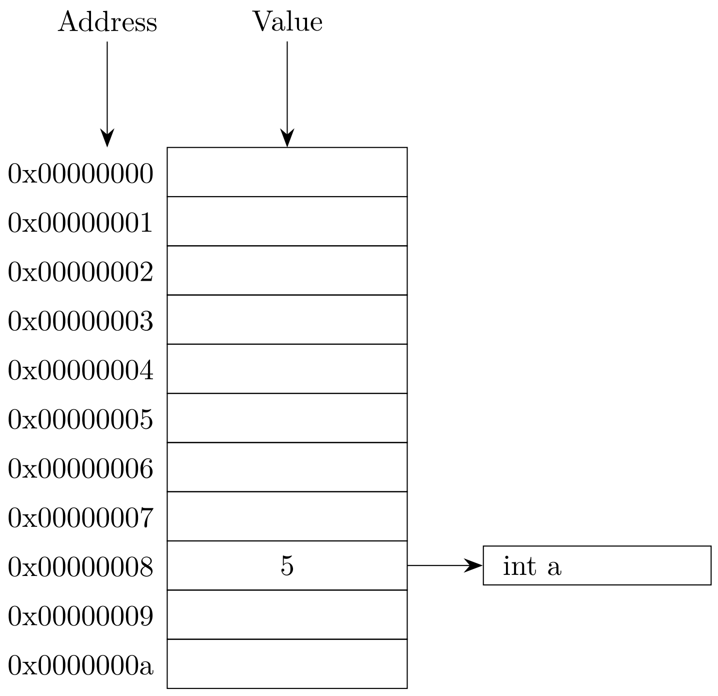


Figure 1: Possible state of the memory after assigning the value 5 to `a`

We notice several things

- each value has an address.
- the order in which values are arranged is not guaranteed.

Indeed, it is perfectly possible that one variable declared after another is placed before (in terms of address) in memory. Furthermore, it is also possible to modify the content of a memory cell without modifying the address.
Suppose I assign 10 to my variable `a`, it'll be stored at the same memory address but the content will have been modified.
Thus, we make a distinction between two things :

- **Value** : the content of the cell
- **Reference** : a link to access the cell

Take your time to fully understand this difference because you'll need it throughout your learning at EPITA.

1.2 Pass by value, pass by reference

When we talk about type in C# we mean the language primitives (`int`, `float`, etc) but also types that are defined by the programmer, like classes and records. [1]
Two categories emerge when talking about types:

- Value types.
- Reference types.

1.2.1 Value types

Values types are types that are systematically copied when moved (affectation, passed as parameters of a function, returned from a function).
Let's take this example:

```
using System;

public class Program
{
    public static void ModifyMyValue(int a)
    {
        a = 5;
    }
    public static void Main()
    {
        int b = 10;
        ModifyMyValue(b);
        Console.WriteLine(b);
    }
}
```

What is the output of this program?
This program outputs `10`. Indeed, the `int` type is a *value type* and is then copied when passed to the function `ModifyMyValue` as an argument. If we go back to the memory layout we previously had, these are the memory modifications happening:

0x000	int b = 10;
0x001	
0x002	
0x003	
0x004	
0x005	

0x000	int b = 10;
0x001	int a = 10;
0x002	
0x003	
0x004	
0x005	

0x000	int b = 10;
0x001	int a = 5;
0x002	
0x003	
0x004	
0x005	

Figure 2: Memory layout after execution of the code

We notice that it's the local copy of `a` by the function that has been modified and not our variable `b` declared in the `Main` function. This behavior is known as `pass by value`.

[1]You will learn about classes next week and records soon after that.

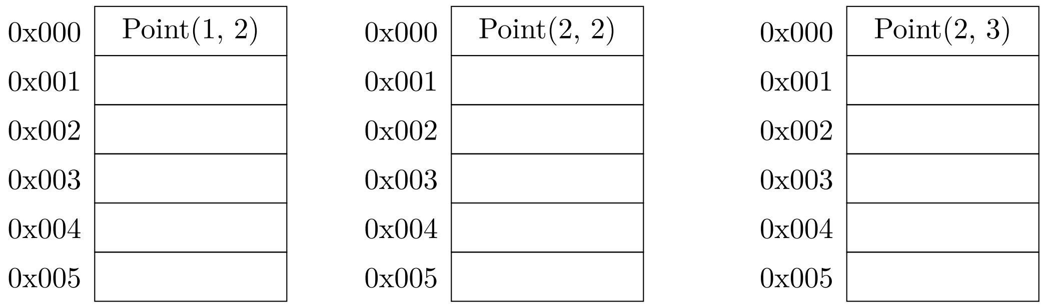
1.2.2 Reference types

On the other hand, reference types are passed as a reference to the actual memory object. Instead of copying the content of the memory cell, a reference to the cell is passed to the function in order to modify it in place.
Let's take the following example:

```csharp
using System;

public class Point
{
    public Point(int x, int y)
    {
        this.X = x;
        this.Y = y;
    }

    public int X {get;set;}
    public int Y {get;set;}
}

public class Program
{
    public static void AddOneToCoords(Point point)
    {
        point.X += 1;
        point.Y += 1;
    }

    public static void Main(string[] argv)
    {
        Point point = new Point(1, 2);
        AddOneToCoords(point);
        Console.WriteLine(point.X);
        Console.WriteLine(point.Y);
    }
}
```

Don't panic, no need to understand all the code that is written. What you have to remember is that we define a new type `Point` which contains two variables `X` and `Y`. This type is a class and is, therefore, a reference type. In the function `AddOneToCoords` we add 1 to our two coordinates. Since a class is a reference type, we pass to our function `AddOneToCoords` not a copy of the value of the class but simply an indication of where it is located in memory. Modifying the inside of the class is like modifying the contents of the cell in memory.

Let's see what's going on :



(a) The Point object is declared with X = 1 and Y = 2

(b) We execute the first instruction of the function, add 1 to X

(c) We execute the first instruction of the function, add 1 to Y

Figure 3: Memory layout after code execution

We notice that no copy is made. It's the content of the class that is modified and not copied and modified entirely.
This behavior is known as *pass by reference*.

1.2.3 The `ref` keyword

The `ref` keyword allows to force the *pass by reference* of a variable that would have been passed as a copy by default (a variable of a type that is value type).
For instance, if we modify the first example like so:

```csharp
using System;

public class Program
{
    public static void ModifyMyValue(ref int a)
    {
        a = 5;
    }
    public static void Main()
    {
        int b = 10;
        ModifyMyValue(ref b);
        Console.WriteLine(b);
    }
}
```

The output of our program won't be `10` anymore, but `5` instead.
We also notice that the keyword is required in the declaration but also in the function call. Indeed, it is useful to know for a programmer who would read the code without knowing the implementation behind that the function `ModifyMyValue` can modify the value passed as a parameter.

1.3 Arrays

1.3.1 Some concepts

An array is a set of values arranged in a specific order. All arrays have the following characteristics:

- The size of an array is fixed and cannot be modified after the creation of the array.
- All values in an array have the same type.
- We can access and change the value of a case at any moment.
- An array can contain other nested arrays.
- Values are placed in cells from the index 0 to n-1 where n is the size of the array.

1.3.2 Declaration of an array

To use an array, we need to start by declaring it. For this we have several possibilities:

```csharp
/* This variable will contain an array of int*/
int[] array;

// We create an array that will contain 5 values that we don't know yet.
array = new int[5];

// We create an array that will contain 5 values that we already know.
int[] bis = {1, 2, 3, 4, 5};
int[][] tabtab = new int[5][]; // Array of arrays
int[][] tabtab2 = { bis, bis }; // Array of already declared arrays.
int[,] dim2 = new int[5, 1]; // Two-dimensional array
int[,] dim2bis = { { 1, 2, 3 }, { 4, 5, 6 } };
// Another two-dimensional array
```

We can declare an array either by giving to the computer only the size that we want for the array without giving any value to put in the array, as in the first example, or with its values as in the second example. In both cases, size is fixed and cannot be changed afterwards. In the second method, the size is the number of values that we put in the array. If we use the first method, all cells of the array are set to 0. The last four lines are declarations of multidimensional arrays, and both types work in the same way, only their use changes.

1.3.3 Accessing a value

To access a value, we proceed like this:

```
bis[0]; // return 1
tabtab2[1][2]; // return 3
dim2bis[1,2]; // return 6
```

We put the index of the cell that we want to access between brackets.

Warning

Trying to access a space outside of the array will make your program crash !

This method allows the user to change the content of a cell:

```
bis[4] = 42; // Array with one dimension
tabtab[2][0] = 23; // Array of arrays
dim2bis[0,1] = 7 // Two-dimensional array
```

In an array of arrays, you have references to arrays. This allows you to store arrays with different sizes. This is not valid with two-dimensional arrays.

```
int[][] tabtab = new int[5][];
tabtab[0] = new int[5];
tabtab[1] = new int[42];
tabtab[2] = new int[1];
tabtab[3] = new int[100];
tabtab[4] = new int[13];
```

1.3.4 Getting the length of a board

To get the length of a board, we proceed like this:

```
array.Length; // return 5
```

For multidimensional boards, we proceed like this:

```
dim2.GetLength(0); // return 5
dim2.GetLength(1); // return 1
```

Where the number in parentheses represents the index of the dimension you want the length of.
For more examples about multidimensional arrays: `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/multidimensional-arrays`

2 Exercices

2.1 References

Let's start by looking at some concrete examples of how references are used in C#. These exercises will allow you to familiarize yourself with the concept and to understand its usefulness.

2.1.1 And Vice and Versa

Write the `Swap` function that swaps two integers.

```
public static void Swap(ref int a, ref int b)
```

2.1.2 Euclidian Division

Write the function `EuclideanDivision` that returns the quotient of the Euclidean division of two integers `a` and `b` and stores the value of the remainder into `a`.

```
public static int EuclideanDivision(ref int a, int b)
```

2.1.3 A complex example

Write the function `ComplexSquare` which calculates the square of a complex number identified by its real part and its imaginary part. The result is stored directly in the references passed as parameters of the function.

```
public static void ComplexSquare(ref int re, ref int im)
```

Remember ?

A complex number looks like this :

$$a + bi$$

where a is called **real part** and b **imaginary part**.
i satisfies the equation $i^2 = -1$.

2.2 Arrays

Let's now look at the manipulation of arrays. It is important to learn how to use this data structure correctly, as it is often very useful.

2.2.1 Swap in an array

Write the function `Swap`, similar to the function in the first part that swaps two elements of an array.

```
public static void Swap(int[] arr, int i, int j)
```

2.2.2 Print me!

Write the function `Print` that prints an array, followed by a new line.

```
public static void Print(int[] arr)
```

An array will be printed between square brackets and the elements split up by pipes `|` looking like this:

$$[\ e_1 \mid e_2 \mid e_3 \mid \ldots \mid e_n\ ]$$

Note the spaces between numbers and brackets as well as spaces between numbers and pipes.

Be careful!

It is mandaroty to scrupulously respect the format specified, that includes spaces (be sure not to add any more spaces that necessary).

2.2.3 Vice max

Write the `ViceMax` functions that return second biggest element of an array of integers.

```
public static int ViceMax(int[] arr)
```

Hint The size of the array will always be at least equal to 2.

Exemples

```
int[] a1 = { -10, 24, 69, 112 };
int[] a2 = { 42, 42, 42, 42 };
int[] a3 = { 57, 92, 48, 29, 13 };

ViceMax(a1); // 69
ViceMax(a2); // 42
ViceMax(a3); // 57
```

2.2.4 Upside down?

Write the `Reverse` function that inverses an array in place.

```
public static void Reverse(int[] arr)
```

2.2.5 Concatenate

Write the `Concat` function that returns the concatenation of two arrays.

```csharp
public static int[] Concat(int[] lhs, int[] rhs)
```

2.2.6 Am I sorted?

Write the `IsSorted` that returns `true` if the array is sorted in increasing order.

```csharp
public static bool IsSorted(int[] arr)
```

2.2.7 Sort me!

Write the `InsertionSort` function that does an in place "in place" insertion sort on the array.

```csharp
public static void InsertionSort(int[] arr)
```

Insertion sort is a fairly simple sorting algorithm. It's principle is the folowing for an array T of size N:

```
for each element $e_i$ in $T$, $i$ ranging from 1 to $n$
    insert $e_i$ at the right position in the array $T[1 \ldots i]$
```

Going further

Insertion sort is relatively slow for large arrays. Despite this, it has many interesting properties, such as being very efficient on small or almost already sorted entries. In practice, it is often used in combination with other algorithms.

2.2.8 Bonus sort - Be creative!

Implement an alternative sorting algorithm to insertion sorting. Document in your `README` its name as well as its advantages and weaknesses compared to the insertion sort.

```csharp
public static void OtherSort(int[] arr)
```

Tip

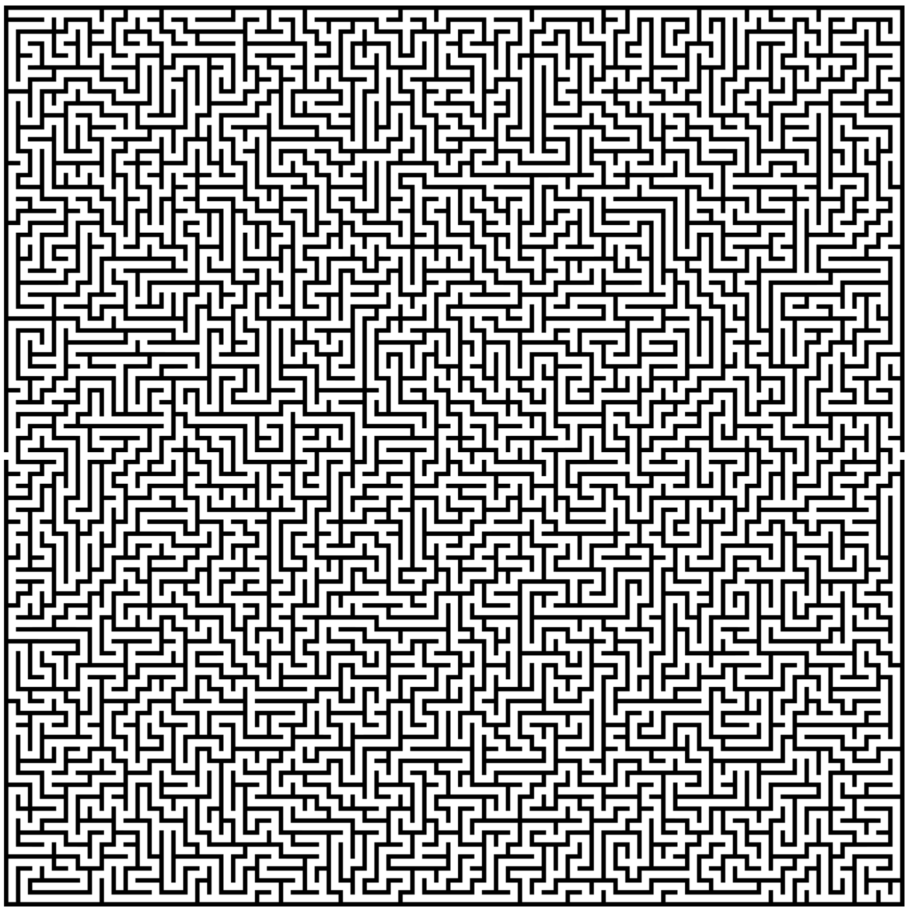
You can for instance implement : quick sort or heap sort that are much more efficient than insertion sort on bigger arrays.
It may be interesting to explore other sorting algorithms such as radix sort. That sorts without doing any comparaison.

2.3 Sherlock's a"Maze"ing mazes

As you know, Sherlock has always had a passion for puzzles and problems of all kinds. Among his favorite puzzles when he was younger was studying and solving mazes. The goal of this exercise will be to see if you can match the talent of our favorite detective.
We will represent a maze as an array of characters in two dimensions. Each character represents a different element of the maze.

Character	**Meaning**
`' '`	Empty cell
`'#'`	Wall
`'@'`	Starting point
`'x'`	Finish point
`'o'`	Cell that is in a path from start to finish

At the end of this exercise, it should be possible to display a maze, determine if a maze contains a valid path and construct such a path. Elementary, isn't it?



2.3.1 Starting from the beginning

Write the `FindStart` function which determines the position of the character `'@'` in the maze passed as input. The function returns `true` if the character exists, `false` otherwise. If the character exists, its position is stored in the references `x` and `y` passed as parameters of the function.

```
public static bool FindStart(char[,] maze, ref int x, ref int y)
```

Hint The character `'@'` is present at least once in the matrix.

Exemple

```
char[,] maze =
{
    {'#', 'x', '#', '#', '#', '#'},
    {'#', ' ', '#', ' ', ' ', '#'},
    {'#', ' ', ' ', ' ', ' ', '#'},
    {'#', '#', ' ', '#', ' ', '#'},
    {'#', ' ', ' ', '#', ' ', '#'},
    {'#', ' ', '#', '#', '#', '#'},
    {'#', ' ', ' ', ' ', ' ', '#'},
    {'#', '#', '#', '#', '@', '#'},
};

int x = 0;
int y = 0;
bool found = FindStart(maze, ref x, ref y);
// found = true
// x = 4
// y = 7
```

2.3.2 Printing, a maze?

Write the `Print` function that prints a maze.

```
public static void Print(char[,] maze)
```

In order for the maze to be pretty, you must print each character twice.

Exemple

```
char[,] maze =
{
    {'#', '@', '#', '#', '#', '#'},
    {'#', ' ', '#', ' ', ' ', '#'},
    {'#', ' ', ' ', ' ', ' ', '#'},
    {'#', '#', ' ', '#', ' ', '#'},
    {'#', ' ', ' ', '#', ' ', '#'},
    {'#', ' ', '#', '#', '#', '#'},
    {'#', ' ', ' ', ' ', ' ', '#'},
    {'#', '#', '#', '#', 'x', '#'},
};

Print(maze);
// ##@@########
// ##  ##    ##
// ##        ##
// ####  ##  ##
// ##    ##  ##
// ##  ########
// ##        ##
// ########xx##
```

2.3.3 Let's check our path

Write the `IsPathValid` function that checks if the labyrinth contains a valid path.

```
public static bool IsPathValid(char[,] maze)
```

A path is indicated by the character `'o'`. A valid path exists if there is a sequence of contiguous `'o'` connecting the beginning and the end of the maze.

Exemple

```
char[,] maze =
{
    {'#', 'x', '#', '#', '#', '#'},
    {'#', 'o', '#', ' ', ' ', '#'},
    {'#', 'o', 'o', ' ', ' ', '#'},
    {'#', '#', 'o', '#', ' ', '#'},
    {'#', 'o', 'o', '#', ' ', '#'},
    {'#', 'o', '#', '#', '#', '#'},
    {'#', 'o', 'o', 'o', 'o', '#'},
    {'#', '#', '#', '#', '@', '#'},
};

IsPathValid(maze); // true
```

Aide

Don't hesitate to use the `FindStart` function that you previously wrote.

2.3.4 Let's solve the maze

Write the `FindPath` funciton that builds a valid path in a maze. If there is no such path, the maze is left unchanged.

```
public static void FindPath(char[,] maze)
```

Exemple

```
char[,] maze =
{
    {'#', '@', '#', '#', '#', '#'},
    {'#', ' ', '#', ' ', ' ', '#'},
    {'#', ' ', ' ', ' ', ' ', '#'},
    {'#', '#', ' ', '#', ' ', '#'},
    {'#', ' ', ' ', '#', ' ', '#'},
    {'#', ' ', '#', '#', '#', '#'},
    {'#', ' ', ' ', ' ', ' ', '#'},
    {'#', '#', '#', '#', 'x', '#'},
};

FindPath(maze);
Print(maze);
// ##@@########
// ##oo##    ##
// ##oooo    ##
// ####oo##  ##
// ##oooo##  ##
// ##oo########
// ##oooooooo##
// ########xx##
```

Some help

To write this function, you can use an auxiliary recursive function that tries to explore every possible path.
Be carefull not to check the same path several times, at risk of getting a `Stack overlow` error.

Be careful!

A function that simply fills in all the free cells with the character `'o'` will not be accepted. Also, remember that if there is no path, the maze should not be modified.

2.3.5 Bonus - Your own maze!

As you know, Sherlock loves solving mazes. However, what he loves even more is to imagine new ones from scratch. Will you be able to do the same?
Write the `Generate` function which, given a height and width, generats a new valid maze.

```
public static char[,] Generate(int width, int height)
```

Where to start?

There exist a lot of algorithms to generate mazes, most used are :

Kruskal's algorithm or Prim's algorithm.

**There is nothing more deceptive,
than an obvious fact.**