TP 10 : Puzzle Solver

1 Submission instructions

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

```
csharp-tp10-sherlock.holmes/
|-- README
|-- .gitignore
|-- Puzzle/
    |--Puzzle.sln
    |--puzzle_game
        |--Board
            |--Tiles
                |--Tile.cs
            |--Board.cs
            |--BoardCheck.cs
            |--BoardMove.cs
            |--BoardPrint.cs
            |--BoardSolver.cs
            |--Direction.cs
        |--GenericTree
            |--GenericTree.cs
            |--MinHeap.cs
        |-- Viewer
            |-- everything
```

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

— You shall replace sherlock.holmes with your login.
— The `README` is mandatory.
— here must be no `bin` or `obj` in 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).

2 Introduction

2.1 Goal

The goal of this practical is to introduce you to the use of new data structures. This will allow for a better and more appropriate representation of data. We will also briefly touch on algorithmic complexity as well as the practicality of certain models in the realization and optimization of algorithms. This will be done through the resolution of a Taquin also known as a 15-puzzle.

Watch out !

This practical is long ! It is strongly recommended to start it as soon as possible !

2.2 A brief history

The Taquin - also called 15-puzzle - looks like this :

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

The Taquin is composed of 15 small tiles numbered from 1 to 15 which slide into a frame designed for 16. The aim of the game is to put the 15 tiles back in order from any initial initial configuration.

Created around 1870 in the United States and claimed later in 1891 by Sam Loyd. Meanwhile, the game has known a real craze, both in the United States as well as in Europe. Indeed, Loyd claimed that he had ”driven the whole world crazy” with a modified Taquin.

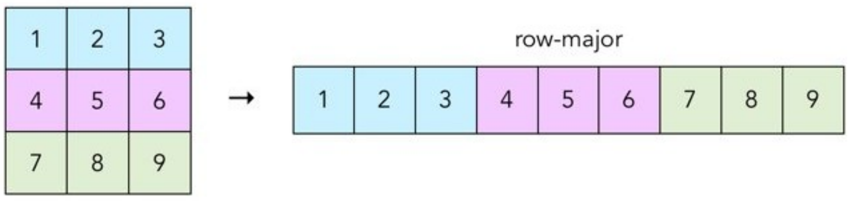
In the proposed configuration, tiles 14 and 15 were swapped, with the empty space at the bottom right. Loyd allegedly promised 1,000 USD to anyone who could put the tiles back in order, but the reward was never claimed. The craze is such that the famous scientific journal *”The American Journal Of Mathematics”* published a research paper called *”Notes on the 15-puzzle”*. The latter demonstrated that with the help of simple mathematical concepts, solvable combinations could be found.

3 Lesson

3.1 Row- and Column-major order

Major row order is a way of representing the elements of a multidimensional array in sequential memory. In row major order, the elements of a multidimensional array are arranged row by row, which means that we must fill in all the indices in the first row and then move to the next row.

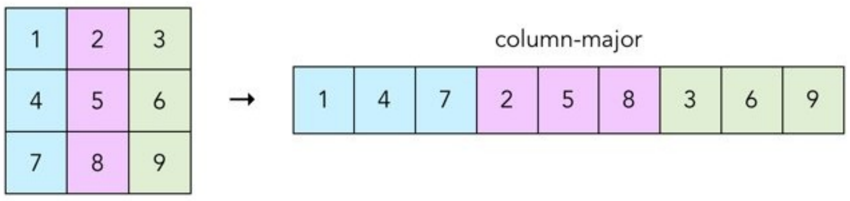
Row-Major Example :



To access the element at **table[0][1]** we apply the following Row-major formula :

row_major[i * width + j] => row_major[0 * width + 1]

Column-Major Example :



To access the element at **table[0][1]** we apply the following Column-major formula :

column_major[j * height + i] => column_major[1 * height + 0]

Watch out! Bugs aplenty!

When a row- column- major has a flawed implementation, it can be a source of very hard to find bugs. Take your time when implementing them!

3.2 Partial Classes

3.2.1 Definition

In C#, you can divide the implementation of a class, a structure, a method or an interface into several .cs files using the `partial` keyword. The compiler will combine all the implementations of several files .cs when the program is compiled. This allows you to organize your code in several files and to avoid the infamous 300 lines file.

3.2.2 Example

File Foo.cs

```csharp
public partial class Foo {
    private int birthYear;
    private string name;

    public Foo(string name, int birthYear){
        this.birthYear = birthYear;
        this.name = name;
    }

    private int getAge(){
        DateTime now = DateTime.Now;
        return (now.Year - this.birthYear > 0) ?
            now.Year - this.birthYear :
            0;
    }
}
```

File Print.cs

```csharp
public partial class Foo {

    public printName(){
        Console.Write($"My name is {this.name} !");
    }

    public printAge(){
        Console.Write($"I am {getAge()} year's old");
    }
}
```

Going further

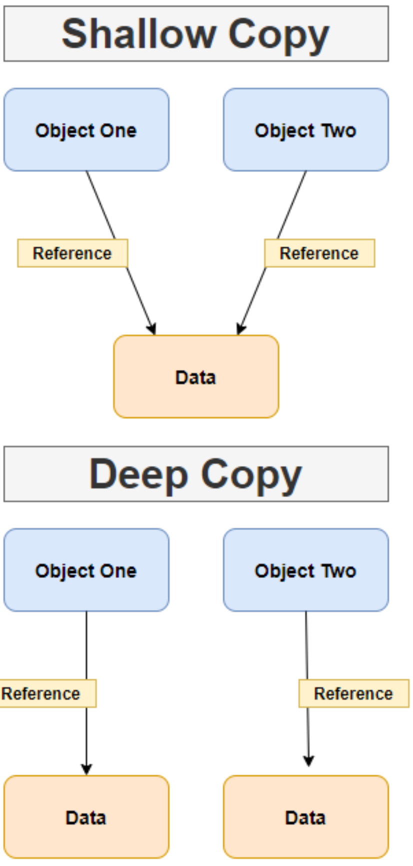
For more information about partial classes, feel free to read more about it *here* !

3.3 DeepCopy and Clones

When we copy objects, they share the same memory address. Normally we use the "=" assignment operator, which copies the reference, not the object (except when there is a value field). For example : Suppose G1 refers to the memory address 5000, then G2 will also refer to 5000. Thus, if we change the value of the data stored at address 5000, G1 and G2 will display the same changed data.

The **ShallowCopy** consists in copying the value fields of an object into the target object. The object's reference types are copied as references into the target object, but not the referenced object itself. It copies the types bit by bit. The result is that both instances are cloned and the original will reference the same object. We can obtain this behavior by using `MemberwiseClone()`.

The **DeepCopy** is used to make a complete deep copy of the internal reference types. To do this, we need to set up the object returned by `MemberwiseClone()`. In other words, a deep copy occurs when an object is copied with the objects it references instead of the references.



Code example for ShallowCopy :

```csharp
    class Program
    {
        static void Main(string[] args)
        {
            Employee emp1 = new Employee();
            emp1.Name = "Adam";
            emp1.Department = "IT";
            emp1.EmpAddress = new Address() { address = "Marseille"};

            Employee emp2 = emp1.GetClone();
            emp2.Name = "Maxence";
            emp2.EmpAddress.address = "Lille"; // Changing the address

            Console.WriteLine("Employee 1: ");
            Console.WriteLine("Name: " + emp1.Name + ", Address: " +
                emp1.EmpAddress.address + ", Dept: " + emp1.Department);

            Console.WriteLine("Employee 2: ");
            Console.WriteLine("Name: " + emp2.Name + ", Address: " +
                emp2.EmpAddress.address + ", Dept: " + emp2.Department);

            Console.Read();
        }
    }
    public class Employee{
        public string Name;
        public string Department;
        public Address EmpAddress;

        public Employee GetClone(){
            return (Employee)this.MemberwiseClone();
        }
    }

    public class Address{
        public string address { get; set; }
    }
```

```
# Output on STDOUT
Employee 1:
Name: Adam, Address: Lille, Dept: IT
Employee 2:
Name: Maxence, Address: Lille, Dept: IT
```

Code example for DeepCopy :

```
    class Program
    {
        static void Main(string[] args)
        {
            Employee emp1 = new Employee();
            emp1.Name = "Anurag";
            emp1.Department = "IT";
            emp1.EmpAddress = new Address() { address = "BBSR"};

            Employee emp2 = emp1.GetClone();
            emp2.Name = "Pranaya";
            emp2.EmpAddress.address = "Mumbai";

            Console.WriteLine("Emplpyee 1: ");
            Console.WriteLine("Name: " + emp1.Name + ", Address: " +
            emp1.EmpAddress.address + ", Dept: " + emp1.Department);

            Console.WriteLine("Emplpyee 2: ");
            Console.WriteLine("Name: " + emp2.Name + ", Address: " +
            emp2.EmpAddress.address + ", Dept: " + emp2.Department);

            Console.Read();
        }
    }
    public class Employee
    {
        public string Name;
        public string Department;
        public Address EmpAddress;

        public Employee GetClone(){
            Employee employee = (Employee)this.MemberwiseClone();
            employee.EmpAddress = EmpAddress.GetClone();
            return employee;
        }
    }
    public class Address
    {
        public string address;

        public Address GetClone(){
            return (Address)this.MemberwiseClone();
        }
    }
```

```
# Output on STDOUT
Employee 1:
Name: Adam, Address: Marseille, Dept: IT
Employee 2:
Name: Maxence, Address: Lille, Dept: IT
```

3.4 Overloading

3.4.1 Definition

Many programming languages allow you to have default or optional parameters. This allows the programmer to make one or more parameters optional by giving them a default value. This technique is particularly useful when adding features to existing code.

For example, let's say you want add functionality to an existing function and it requires the addition of one or more parameters. In doing so, you would break the existing code that calls that function, since it would not pass the required number of parameters. To get around this problem, you can can set the newly added parameters as optional and give them a default value value that matches the way the the way the code would behave before the parameters were added.

3.4.2 Example

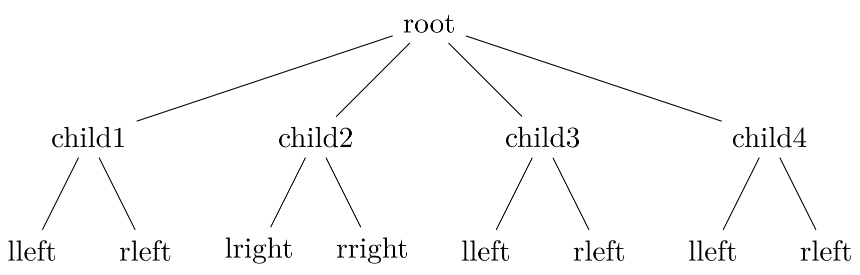
```
    class SillyMath
    {
        public static int Mult(int number1, int number2){
            return number1 * number2;
        }

        public static int Mult(int number1, int number2, int number3){
            return number1 * number2 * number3;
        }

        public static int Mult(int number1, int number2,
                int number3, int number4){
            return number1 * number2 * number3  * number4;
        }
    }
```

3.5 Reminders on general trees

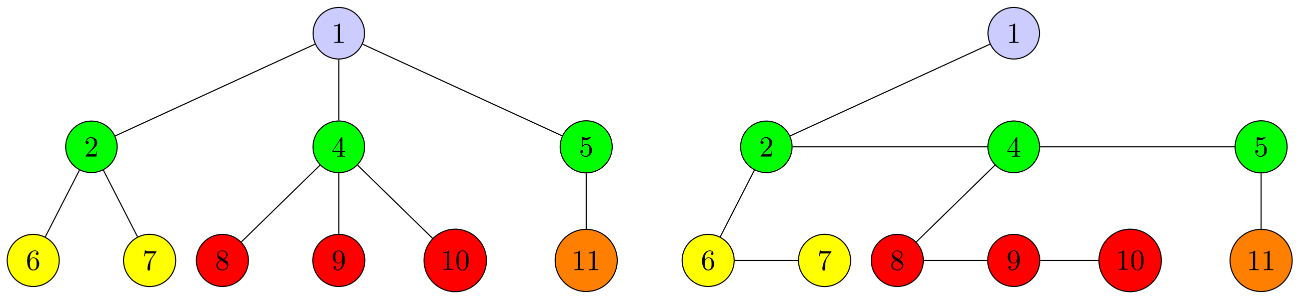
A tree is a non-empty collection of nodes and edges with the following properties :



— A node is a simple object.
— An edge is a link between two nodes.
— A branch of the tree is a sequence of distinct nodes in which two successive nodes are connected by an edge.
— There is a special node called the root.
— The defining property of a tree is that there is exactly one branch between the root and each of the other nodes of the tree.
— It is customary to represent trees with the root located above all other nodes.
— Each node, except the root, has a node "above it", called the father.
— The nodes directly below a node are called children. We can also talk about siblings, grandfathers, etc...
— Nodes that have no progeny are called leaves, terminal nodes or external nodes. Nodes that do are called internal nodes.

3.5.1 Implementation

There are several ways to implement a general tree. In this tutorial you will be confronted with the ***"Left-Child Right-Sibling"*** implementation.



(a) Classic (b) LC-RS

FIGURE 1 – General tree representation

To build a Left-Child Right-Sibling tree you must :

1. At each node, link the children of the same parent from left to right.
2. Remove links from the parent to all children except the first child.

Since we have a link between the children, we do not need additional links from the parents to all the children. This representation allows us to go through all the elements starting with the first child of the parent.

That said, it is possible to **label each node** in order to later determine a path from the root to a given node.

Advice

If you plan to make the bonuses, we recommended you to have a look at the file **GenericTree.cs** !

3.5.2 Distances

Let E be a set. We call distance over E an application $d : E \times E \to \mathbb{R}_+$ that verifies the following 3 propreties :

1. $d(x, y) = d(y, x), \forall (x, y) \in E^2$
2. $d(x, y) = 0, x = y, (x, y) \in E^2$
3. $d(x, y) \leq d(x, z) + d(z, y), \forall (x, y, z) \in E^3$

When equipped with a distance E becomes a **metric space**.
In computer science, the notion of distance is very important as they play an important role in machine learning. They form the basis of many popular and effective machine learning algorithms, such as k-nearest neighbors for supervised learning and *k-means* clustering for unsupervised learning.

Here is a non-exhaustive list of distances that you should encounter during your time at EPITA.

Name	Parameter	Function
euclidean distance	2-distance	$\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$
Manhattan distance	1-distance	$\sum_{i=1}^{n} \lvert x_i - y_i \rvert$
Levenshtein distance	1-distance	$\text{lev(A,B)}=\begin{cases} \max(\lvert a\rvert,\lvert b\rvert) & \text{si } \min(\lvert a\rvert,\lvert b\rvert) = 0 \\ lev(a-1, b-1) & \text{si } a[0] = b[0] \\ 1 + \min \begin{cases} lev(a-1, b) \\ lev(a, b-1) \\ lev(a-1, b-1) \end{cases} & \text{sinon} \end{cases} \quad (1)$

During this practical we will focus on the **Manhattan distance**, which is the distance between two points measured along right angle axes. In a plane with p_1 at coordinates (x_1, y_1) and p_2 at (x_2, y_2), the distance between p_1 and p_2 would be $|x_1 - x_2| + |y_1 - y_2|$.

3.5.3 Heuristic functions

A heuristic function is a function that computes an approximate cost for a given problem (or ranks the alternatives).

For example, the problem might be to find the shortest driving distance to a point. A heuristic cost would be the straight line distance to that point. It is simple and fast to compute, an important property of most heuristics functions. The actual distance would probably be higher because we have to stick to roads and it is much harder to compute.

Heuristic functions are often used in combination with search algorithms. You may also see the term admissible, which means that the heuristic never overestimates the actual cost. Admissibility is an important quality and is required for some search algorithms like the A* algorithm.

Going further

If you want to read more about heuristics you will find this *link* to be quite interesting.

3.6 MinHeap

In computer science, a heap data structure is a special tree that satisfies the heap property, which means that the parent is less than or equal to the child node for a minimum heap, also called a min heap, and the parent is greater than or equal to the child node for a maximum heap, also called a max heap. The binary heap was created by J.W.J. Williams in 1964 for **heapsort**.

A binary heap is a binary tree with two other constraints :

1. Shape property : A binary heap is a complete binary tree, which means that all levels of the tree are completely filled, except possibly the last level. The nodes are filled from left to right.
2. Heap property : The value stored in each node is either (greater than or equal to) OR (less than or equal to) its children depending on whether it is a maximum or minimum heap.

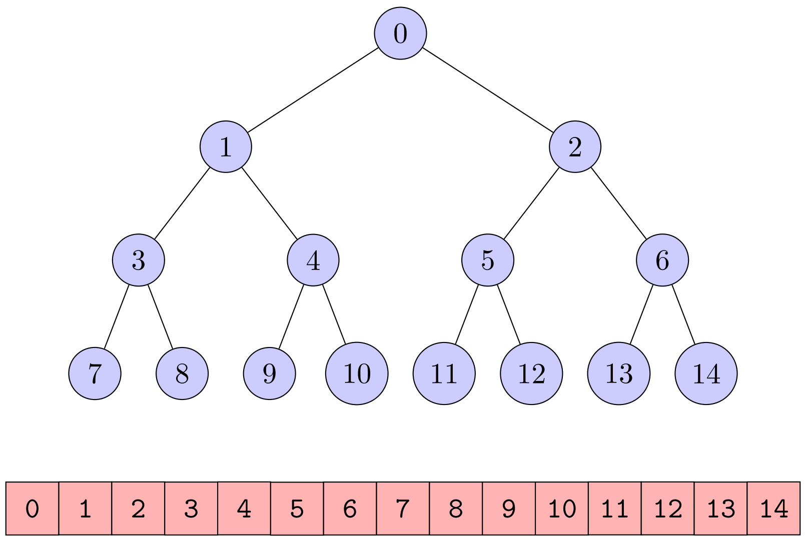
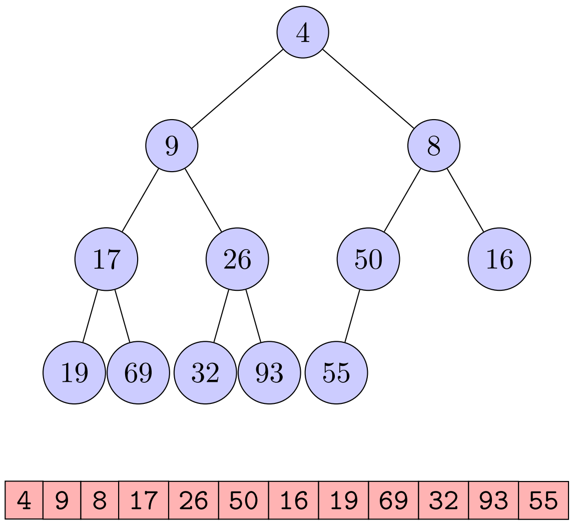


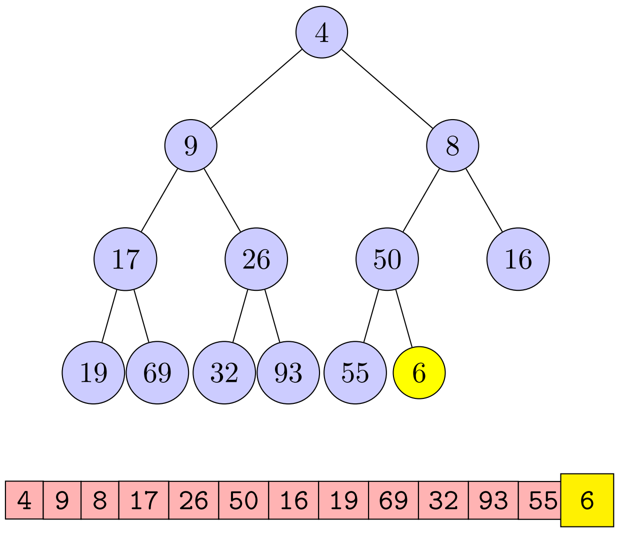
FIGURE 2 – The Eytzinger method represents a complete binary tree as an array

3.6.1 Add

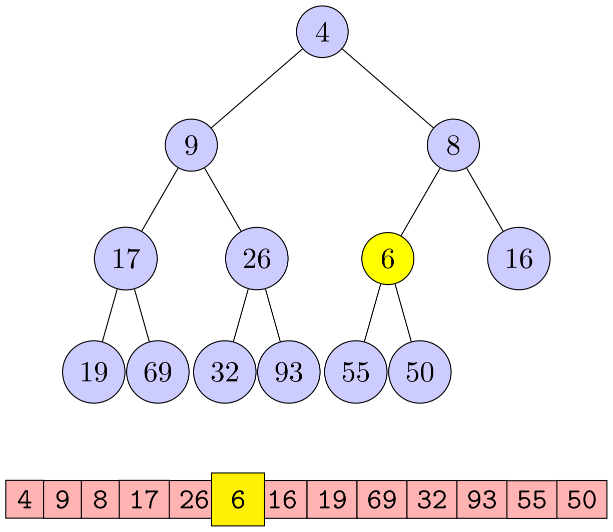
The add(x) function is quite simple. As with all all array-based structures, we first check if a is full (by checking if length(a) = n) and, if so, we increase the size of a. Then we place x at the location $a[n]$ and increment n. At this point, all that remains is to make sure that we maintain the property of the heap. To do so, we repeatedly swap x with its parent until x is no longer smaller than its parent.



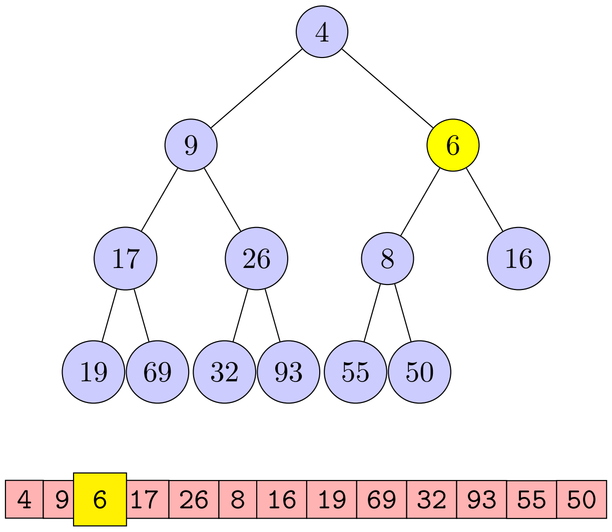
(a) Starting tree



(b) We add the node 6 at the end of the list



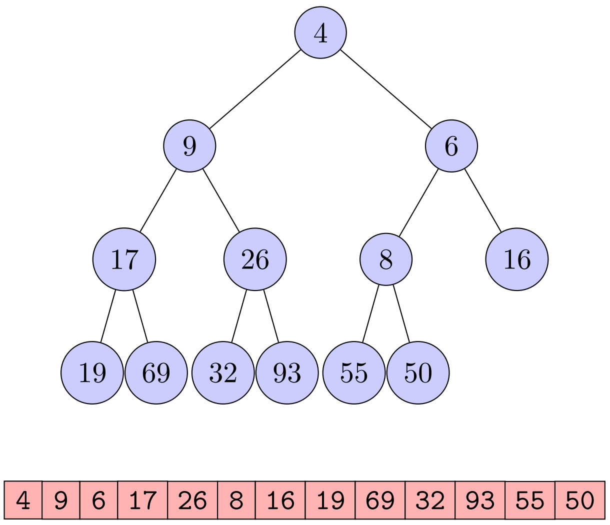
(a) We swap the nodes 6 and 50



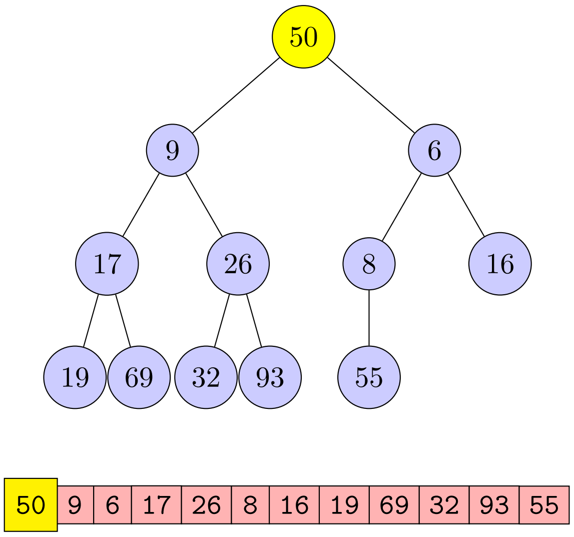
(b) We swap the nodes 8 and 6

3.7 Remove

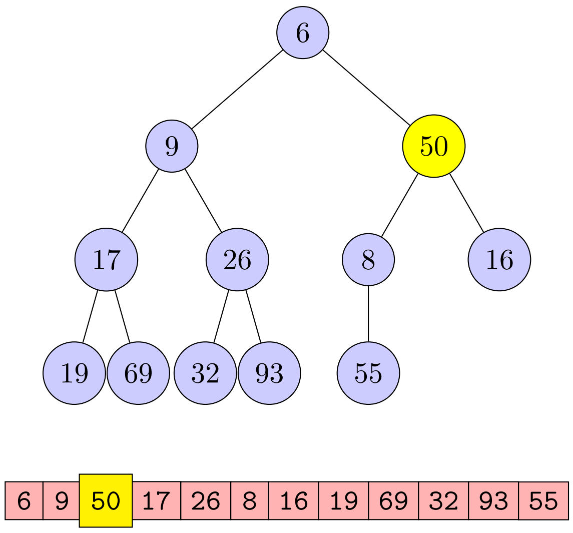
The remove() function, which removes the smallest value from the heap, is a bit trickier. We know where the smallest value is value (at the root), but we need to replace it after removing it and make sure we maintain the property of a heap. The easiest way to do this is to replace the root with the value $a[n-1]$, delete this value, and decrement n. Unfortunately, the new root element element is probably not the smallest element, so it must be moved down. To do this, we repeatedly compare this element to its two children. If it is the smallest of the three, we are done. If not, we swap this element with the smallest of its two children and continue.



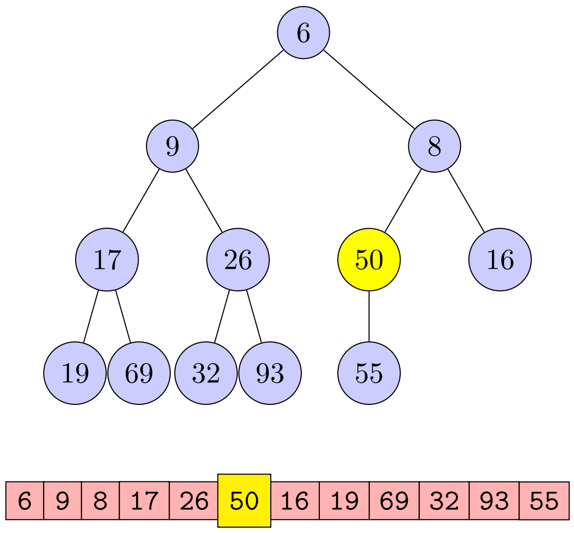
(a) The starting tree



(b) We swap the nodes 4 and 50 then we delete node 4



(a) We swap the nodes 6 and 50



(b) We swap the nodes 8 and 50

4 Exercises

In this exercise you will implement a 15-Puzzle and an algorithm to solve it.
You will have to implement an array to represent the game board. To do this, you will use the Row-Major representation of a matrix.

Warning

The use of a matrix or a two-dimensional array is prohibited and will be penalized.

You will start by implementing the functions that will be used to create the 15-Puzzle array as well as the cells that compose it. Once we have an array to manipulate, you will implement the different functions that will be used to mix and solve the 15-Puzzle.

Implementation

The return values and exceptions that the functions return will be specified in each exercise. If nothing is said then you will not have to deal with this case.

Recommendation

It is strongly recommended to have played at least once the 15-puzzle in order not to be lost during the writing of the functions. Here is a little *link* !

We have provided you with a puzzle viewer. This viewer will be used to visualize the puzzle and its resolution once you have implemented the mandatory exercises.

Warning !

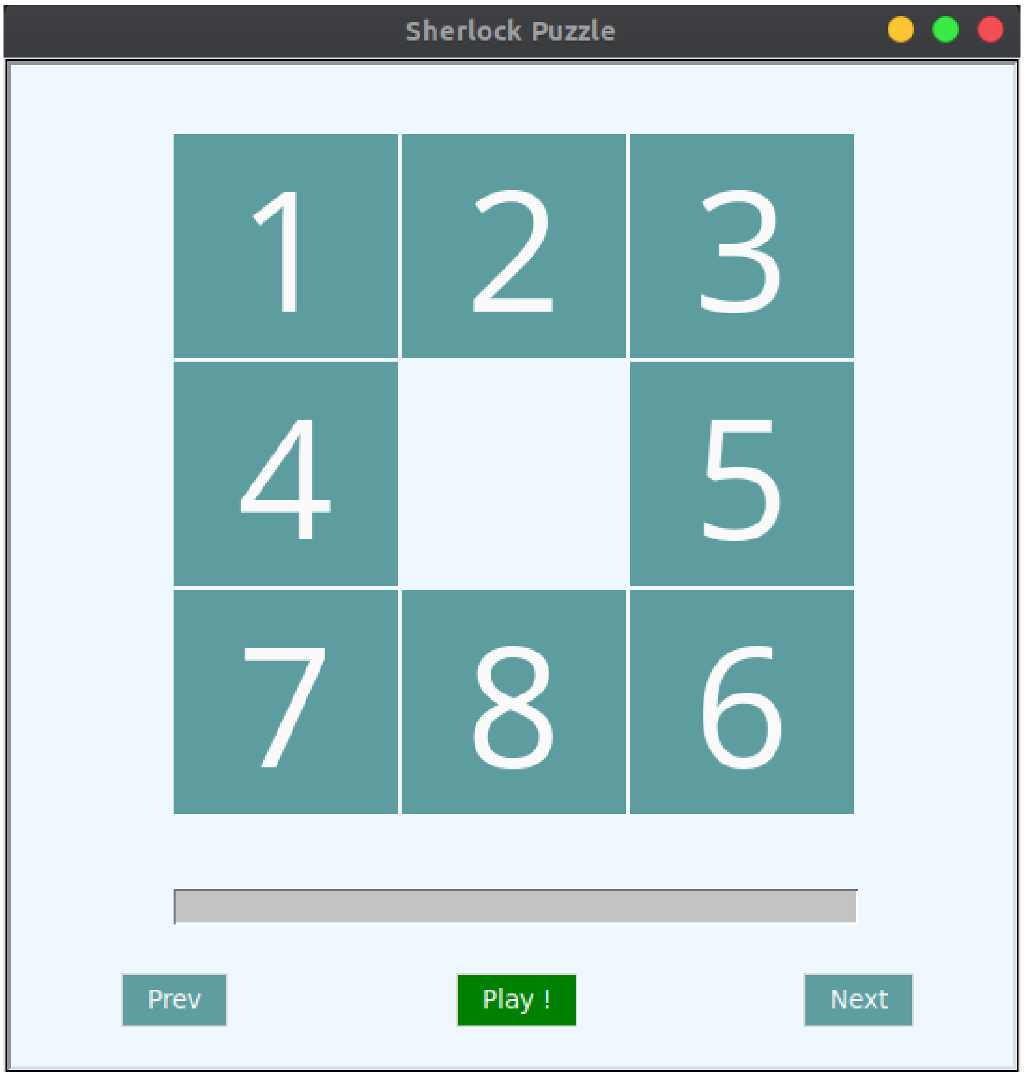
The viewer has been developed to run on the school's machines. We cannot guarantee that it will work on other machines.

To launch the `Viewer` :

```csharp
Board board = new Board(16);

//List<Direction> steps = board.SolveBoard();
List<Direction> steps = board.SolveBoardBonus();

Viewer.Parse(board, steps); // Important! Otherwise it won't work!
Viewer.LaunchViewer();
```



- **Play** : Automatically starts the resolution sequence written in the file `Viewer/steps.out`
- **Next** : Next step.
- **Prev** : Previous step.

Installation

If you want to run the `Viewer` on your machines, it is strongly recommended to install the dependencies in the `Viewer/requirements.txt` file.

```
pip install -r requirements.txt
```

4.1 Board

4.1.1 Constructors

The Tile constructor should follow the following prototype :

```
public Tile(int value, bool empty)
```

In the Tile constructor you initialize the following members :

— type - represents the state of the box, whether it contains a number or not.
— value - the number in the box

Info

The value of a cell of type `Tile.EMPTY` is a 0.

The Board constructor shall follow the following prototype :

```
public Board(int size)
```

In the Board constructor you initialize the following members :

— size - the total number of cells in our array, must be a perfect square[1].
— width - the number of cells in a row
— board - The Tile list
— solved - a Boolean that reflects the state of the puzzle initialized to `false`

In the following cases you must throw a new exception of type `ArgumentException` :

— `size <= 0` : the exception will be thrown with the message "Size needs to be positive !".
— `size` is not a perfect square : the exception will be thrown with the message "Size needs to be a perfect square !

4.1.2 Deep copy

The DeepCopy functions will follow the following prototypes :

```
public Board DeepCopy()

public Tile DeepCopy()
```

Making copies of our Boards is unfortunately not a simple matter. If we wrote '`Board newBoard = oldBoard`' we would not really have a new Board because the values of newBoard and oldBoard would still be linked. To overcome this problem we will make a deep copy[2].

In this exercise you will have to create a new Board, copy all the values manually and then return it. You will do the same for the DeepCopy function of Tile.

1. For more information about perfect squares :*https ://en.wikipedia.org/wiki/Square_number*
2. For more information about the different types of copies : *https ://en.wikipedia.org/wiki/Object_copying*

4.1.3 AreConsecutive

The AreConsecutive function will follow the following prototype :

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

Our game board will contain boxes numbered from 1 to 15 and an empty box. These cells are not necessarily in the correct order but all the numbers between 1 and 15 are present. For this function you will check that the elements in `arr` are the integers between 0 and the size of the list.

You will return `True` if these conditions are met and `False` otherwise

Example :

```csharp
int[] array = new int[]{0,1, 2, 3, 4, 5};
Board.AreConsecutive(array); // True
array = new int[]{5, 4, 2, 1, 3, 7, 6, 0};
Board.AreConsecutive(array); // True
array = new int[]{0};
Board.AreConsecutive(array); // True
array = new int[]{1, 6, 8, 10, 12};
Board.AreConsecutive(array); // False
array = new int[]{1, 2, 3, 3, 4};
Board.AreConsecutive(array); // False
array = new int[]{1, 3, 5, 7, 9};
Board.AreConsecutive(array); // False
```

4.1.4 Fill

The Fill function will follow the following prototype :

```csharp
public void Fill()
```

In this function you must fill the member `board` with boxes numbered from 1 to 15 and an empty box. These boxes must be ordered in ascending order and the list must end with an empty box.

```csharp
public void Fill(int[] array)
```

In this function you have to fill the board from the array `array` given in parameter. Don't forget to check that the array is valid and that there is a single empty cell. Be careful! if the array given in parameter is not consecutive then raise an exception of type `ArgumentException`.

Reminder

Remember that the value of a `Tile.EMPTY` type box is a 0.

4.2 BoardCheck

4.2.1 IsCorrect

The IsCorrect function should follow the following prototype :

```
public bool IsCorrect()
```

The IsCorrect function is relatively simple, you need to check that the puzzle has been solved. We consider the puzzle to be solved when all the Tiles contained in the Board are ordered and only the last Tile is of type `TileType.EMPTY|` .

You will return `True` if the Board is resolved and `False` otherwise.

4.2.2 FindEmptyPos

The FindEmptyPos function will follow the following prototype :

```
public int FindEmptyPos()
```

In this function, you must return the index of the empty cell. Return -1 if not found

Example `board` :

```
Board board = new Board(16);
int array = new int[]
{
    3, 5, 6, 4,
    1, 2, 7, 8,
    9, 10, 0, 12,
    13, 14, 11, 15
};
board.Fill(array);

Console.WriteLine($"The Empty Tile is at position {board.FindEmptyPos()}");
```

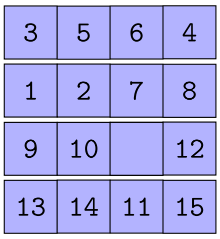
```
The Empty tile is at position 10
```

4.2.3 IsSolvable

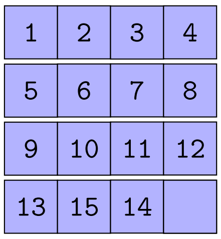
The IsSolvable function will follow the following prototype :

```
public bool IsSolvable()
```

It is possible to know in advance if an array is solvable by counting the number of inversions present in it. You will implement this test in this function.



(a) Solvable Grid



(b) Non-Solvable Grid !

FIGURE 7 – Some grid example

Here, N represents the size of the grid.

1. If N is odd and the number of inversions is even, the puzzle is soluble.
2. If N is even, the puzzle is solvable if :
   — The empty square is on an even row counting from the bottom (second to last, fourth to last, etc.) and the number of inversions is odd.
   — The empty square is on an odd line counting from the bottom (last, third, fifth, etc.) and the number of inversions is even.
3. In any other case, the puzzle is not solvable.

How to count the inversions, you ask me ?
A pair of numbers, in a table, is an inversion when they appear in the wrong order. Suppose we have the following table **{1, 2, 4, 3}**, then the pair **(4,3)** is an inversion because $4 > 3$ and 4 appears before 3. The array **{3, 1, 2, 4}** has two inversions **(3,1)** and **(3,2)**, while **{1, 2, 3, 4}** has no inversions.

In other words, two elements `arr[i]` and `arr[j]` form an inversion if the following two conditions are verified : `arr[i] > arr[j]` and `i < j`.

4.3 BoardMove

4.3.1 GetPossibleDirections

The GetPossibleDirections function will follow the following prototype :

```
public Direction[] GetPossibleDirections()
```

```
int[] array = new int[] {
    1,2,3,
    4,5,6,
    7,8,0
};
Board board = new Board(9);
board.Fill(array);
Console.WriteLine("Possible Directions");
foreach (var direction in board.GetPossibleDirections()){
    Console.Write($"{direction} ");
}
```

```
UP LEFT
```

4.3.2 SwapTile

The SwapTile function will follow the following prototype :

```
public void SwapTile(int i1, int i2)
```

`SwapTile` exchanges the position of two cells !
If the cells are of the same type, you must raise an exception of type `ArgumentException()`

4.3.3 MoveDirection

The MoveDirection function will follow the following prototype :

```
public bool MoveDirection(Direction direct)
```

`MoveDirection` allows you to move the `Empty Tile` by swapping it with the cell that is `UP, DOWN, LEFT, RIGHT`.

```
Board board = new Board(9);
int[] array = new int[]{
    1,2,3,
    4,5,6,
    7,8,0
};

board.MoveDirection(LEFT);
board.Print();
board.MoveDirection(UP);
board.Print();
```

```
╔═══════════════╗
║│ 1 ││ 2 ││ 3 │║
║├───┤├───┤├───┤║
║│ 4 ││ 5 ││ 6 │║
║├───┤├───┤├───┤║
║│ 7 ││ X ││ 8 │║
╚═══════════════╝
╔═══════════════╗
║│ 1 ││ 2 ││ 3 │║
║├───┤├───┤├───┤║
║│ 4 ││ X ││ 6 │║
║├───┤├───┤├───┤║
║│ 7 ││ 5 ││ 8 │║
╚═══════════════╝
```

4.3.4 Shuffle

The Shuffle function will follow the following prototype :

```csharp
public void Shuffle(int nbr)
```

Shuffle the board while keeping it solvable. To mix the board you will have to make valid random moves. Be careful : `nbr` must be positive! If `nbr` is not valid then raise an exception of type `ArgumentException`

4.4 BoardPrint

4.4.1 PadCenter

The PadCenter function will follow the following prototype :

```
public string PadCenter(string s, int width, char c)
```

`PadCenter` returns the string `s`, padded right, left or both ways with the character `c` until it reaches the size of `width`.

```
string result = PadCenter("Hello", 9, '\$');
Console.WriteLine(result);
result = PadCenter("Hello", 8, '\$');
Console.WriteLine(result);
result = PadCenter("Hello", 3, '\$');
Console.WriteLine(result);
result = PadCenter("Hello", -3, '\$');
Console.WriteLine(result);
```

```
$$Hello$$
$$Hello$
Hello
Hello
```

4.4.2 PrintLine

The PrintLine function will follow the following prototype :

```
public void PrintLine(int i, int width, int longest_number)
```

The PrintLine function takes 3 arguments :
— longest_number : The length of the largest number. i.e 100 -> 3, 1000 -> 4, 2 -> 1.
— width : width of the grid
— i : the index of the line of the matrix
This function displays the intermediate rows that make up the table.

```
// Case where i = 0, first line of the grid
Console.WriteLine("New_line:");
PrintLine(0, 4, 2); //  We print for a grid from 1 to 15

// Case where i != 0 and != width, intermediate line
Console.WriteLine("New_line:");
PrintLine(2, 4, 2); //  We print for a grid from 1 to 15

// Case where i = width, last line of the grid
Console.WriteLine("New_line:");
PrintLine(4, 4, 2); //  We print for a grid from 1 to 15
```

```
New_line:
╔════╦════╦════╦════╗

New_Line:
╠════╬════╬════╬════╣

New_Line:
╚════╩════╩════╩════╝
```

4.4.3 Print

The Print function will follow the following prototype :

```csharp
public void Print()
```

This functions will print the entire board.

```csharp
int[] sizes = new int[] {16, 9, 1};

foreach (int size in sizes)
{
    Board board = new Board(size);
    board.Fill();
    Console.WriteLine(\$"Board of size {size}:");
    board.Print();
}
```

Tip !

Using the functions implemented above can be very useful !

Warning !

The display must be identical to the example below !

```csharp
int[] sizes = new int[] {16, 9, 1};

foreach (int size in sizes){
    Console.WriteLine($"Board of size {size}");
    Board board = new Board(size);
    board.Print();
}
```

```
Board of size 16:
╔═══╦═══╦═══╦═══╗
║  1║  2║  3║  4║
╠═══╬═══╬═══╬═══╣
║  5║  6║  7║  8║
╠═══╬═══╬═══╬═══╣
║  9║ 10║ 11║ 12║
╠═══╬═══╬═══╬═══╣
║ 13║ 14║ 15║  X║
╚═══╩═══╩═══╩═══╝
Board of size 9:
╔═══╦═══╦═══╗
║ 1 ║ 2 ║ 3 ║
╠═══╬═══╬═══╣
║ 4 ║ 5 ║ 6 ║
╠═══╬═══╬═══╣
║ 7 ║ 8 ║ X ║
╚═══╩═══╩═══╝
Board of size 1:
╔═══╗
║ X ║
╚═══╝
```

4.5 SolveBoard

4.5.1 ManhattanDistance

The ManhattanDistance function will follow the following prototype :

```
public int ManhattanDistance(int i1, int i2)
```

You will have to implement a function to determine the shortest Manhattan distance between 2 cells of the Board. The formula has been given to you in the *lesson* !

The function has two arguments :
— i1 : The index of the first cell in Row Major
— i2 : The index of the second cell in Row Major
If the index of the cells is not correct, you must raise the following exception : `ArgumentException`.

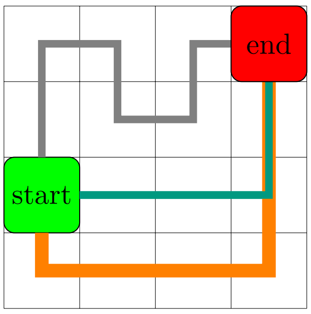


FIGURE 8 – Manhattan Distance on the grid

4.5.2 CalculateHeuristic

The CalculateHeuristic function will follow the following prototype :

```
public int CalculateHeuristic()
```

To allow you to make a choice when solving the puzzle, you need to implement a function that ranks the alternatives at each branching step according to the location of each tile to decide which branch to follow. Here, the function calculates the sum of the Manhattan distances of each tile with its final position.
For example :

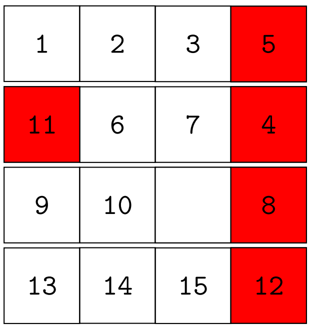
$$\alpha(x, y) = \begin{cases} d_{Manhattan}(x, y) & \text{if } goal[y] > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\beta_{Heuristic}(board, goal) = \sum_{i=0}^{N} \alpha(board[i], goal[i])$$

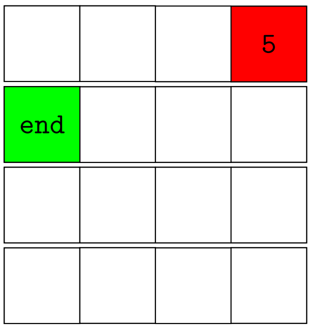
1	2	3	5
11	6	7	4
9	10		8
13	14	15	12

First of all let's select the cells that are not not in the right place (Ignoring the empty empty cell of course) !

1	2	3	🟥 5
🟥 11	6	7	🟥 4
9	10		🟥 8
13	14	15	🟥 12



We then calculate for each cell badly placed its distance of Manhattan!



So we apply the same process for each tile and we add the whole!

```
Board board = new Board(9);
int[] array = new int[]
{
    4,3,1,
    7,2,5,
    0,8,6
};

board.Fill(array);
Console.WriteLine($"Heuristic Value of the board \
{board.CalculateHeuristic()}.");

array = new int[]
{
    1,2,3,
    4,5,6,
    0,8,7
};
board.Fill(array);

Console.WriteLine($"Heuristic Value of the board \
{board.CalculateHeuristic()}.");
```

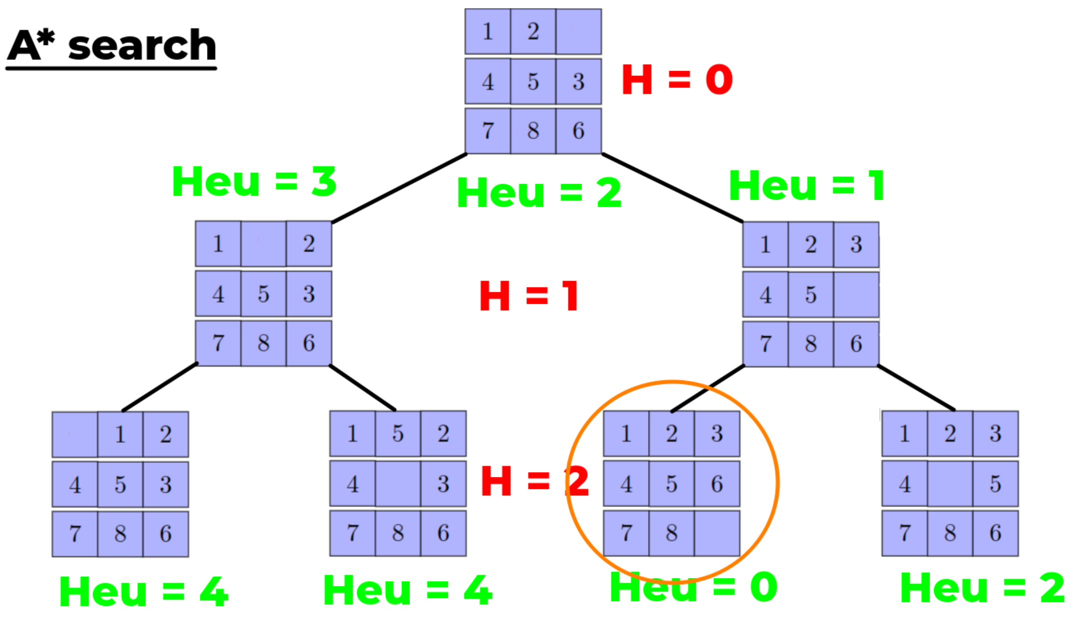
```
Heuristic Value of the board 8.
Heuristic Value of the board 2.
```

4.5.3 SolveBoard

The SolveBoard function will follow the following prototype :

```
public List<Direction> SolveBoard()
```

You will have to implement the function that allows to solve the board! It returns a list of `Direction` which allows the resolution of the board. For that we propose a naive algorithm which works only for the puzzles which do not contain cycles!



For each possible move, determine and select the child that will allow you to arrive in the fewest possible steps on the final board!.

Watch out for cycles!

In some cases, puzzles may contain cycles. Your algorithm is currently not supposed to handle them. For example, this *board* contains a cycle!

```csharp
Board board = new Board(9);

int[] array = new int[]{
    1,2,3,
    4,0,5,
    7,8,6
};
board.Fill(array);
List<Direction> steps = board.SolveBoardBonus();

Console.WriteLine("Resolution finished");
foreach (var step in steps) {
    Console.Write($"{step} ");
}
```

```
Resolution finished
RIGHT DOWN
```

5 Bonus

In this section we will optimize the solving algorithm so that it can solve puzzles including cycles.

5.1 MinHeap

Before starting this part it is advisable to read the part of the course on MinHeap. You will have to code in the file `GenericTree/MinHeap.cs`. In the file there are several non-required functions. However, they are not mandatory but their use is strongly recommended.

5.1.1 MinHeapify

The time complexity of this operation is $O(\log n)$. If the value of the descending key of a node is greater than that of the parent of the node, we continue. Otherwise, we traverse upwards to correct the violation of the heap property. The function `MinHeapify` will follow the following prototype :

```
public void minHeapify(int pos);
```

For more information we redirect you to this *link* .

5.1.2 Enqueue

The function `Enqueue` adds the element in the MinHeap keeping it coherent.
The function `Enqueue` will follow the following prototype :

```
public void Enqueue(HeapElement<T> element)
```

For more information please refer to the course *here* .

5.1.3 Dequeue

The function `Dequeue` returns the minimum while keeping the MinHeap consistent.
The function `Dequeue` will follow the following prototype :

```
public HeapElement<T> Dequeue();
```

For more information please refer to the course *here* .

5.2 Solver

5.2.1 SolveBoardBonus

For the last function of this tutorial, you will have to implement the algorithm of **A*** on general trees.
The function `SolveBoardBonus` will follow the following prototype :

```
public List<Direction> SolveBoardBonus()
```

Tip

When you are looking for an algorithm, get into the habit of looking on **http ://scholar.google.fr**. You will understand very quickly why :).

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