Exercises: Escape from Labyrinth

This document defines the **in-class exercises** assignments for the "Data Structures" course @ Software University.
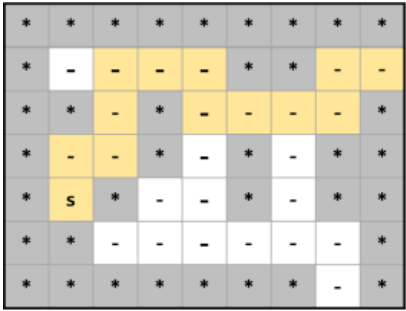
Problem 1. Find the Nearest Exit from a Labyrinth

The second part of this lab aims to implement the **Breadth-First-Search (BFS) algorithm** to find the nearest possible exit from a labyrinth. We are given a labyrinth. We start from a cell denoted by '**s**'. We can move **left**, **right**, **up** and **down**, through empty cells '**-**'. We cannot pass through walls '*****'. An exit is found when a cell on a labyrinth side is reached.
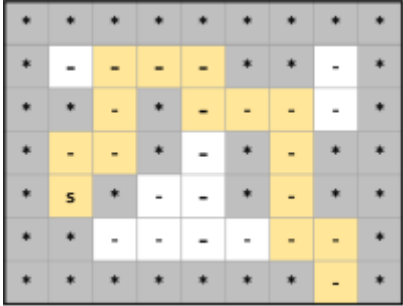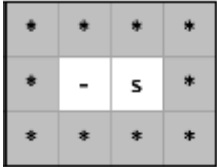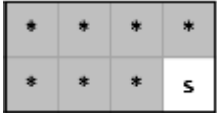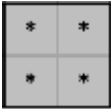
For **example**, consider the labyrinth below. It has size **9 x 7**. We start from cell **{1, 4}**, denoted by '**s**'. The nearest exit is at the right side, the cell **{8, 1}**. The path to the nearest exit consists of **12** moves: **URUURRDRRRUR** (where '**U**' means up, '**R**' means right, '**D**' means down and '**L**' means left). There are two exits and several other paths to these exits, but the path **URUURRDRRRUR** is the shortest.

The input comes from the console. The first line holds the labyrinth **width**. The second line holds the labyrinth **height**. The next height lines hold the labyrinth cells – characters '*****' (wall), '**-**' (empty cell) or '**s**' (start cell).

Examples:

Input	Labyrinth	Output
9 7 * * * * * * * * * * - - - - * * - - * * - * - - - - *		Shortest exit: URUURRDRRRUR

`* - - * - * - * *`
`* s * - - * - * *`
`* * - - - - - *`
`* * * * * * - *`		

9
7
`* * * * * * * * *`
`* - - - - * * - *`
`* * - * - - - - *`
`* - - * - * - * *`
`* s * - - * - * *`
`* * - - - - - *`
`* * * * * * - *`
		Shortest exit: URUURRDRRDDDRD

4
3
`* * * *`
`* - s *`
`* * * *`
		No exit!

4
2
`* * * *`
`* * * s`
		Start is at the exit.

2 2 `* *` `* *`		No exit!

Problem 1. Escape from Labyrinth – Project Skeleton

You are given a **Visual Studio project skeleton** (unfinished project) holding the unfinished class **EscapeFromLabyrinth** and **unit tests** for its functionality. The project holds the following assets:
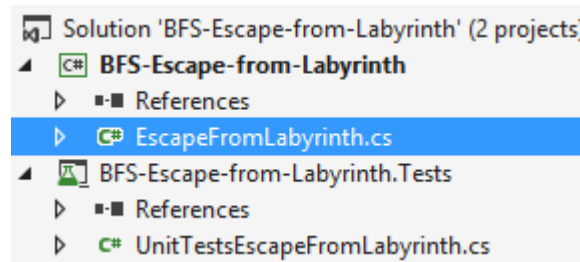
The project skeleton opens correctly in **Visual Studio 2013** but can be open in other Visual Studio versions as well and also can run in **SharpDevelop** and **Xamarin Studio**.

The unfinished **EscapeFromLabyrinth** class stays in the file **EscapeFromLabyrinth.cs**:

<table>
<tr><td align="center">EscapeFromLabyrinth.cs</td></tr>
</table>

```csharp
public class EscapeFromLabyrinth
{
    public static void Main()
    {
        // TODO: implement me
    }
}
```

The project comes with **unit tests** covering the functionality of the **EscapeFromLabyrinth** class:

```
var outputWriter = new StringWriter();
using (outputWriter)
{
    Console.SetIn(inputReader);
    Console.SetOut(outputWriter);
    EscapeFromLabyrinth.Main();
}
var output = outputWriter.ToString();

// Assert
var expectedOutput = "Shortest exit: URUURRDRRRUR";
output = output.Replace("\r\n", "\n").Trim();
Assert.AreEqual(expectedOutput, output);
}

[TestMethod]
public void TestLabyrinth4x3()
```
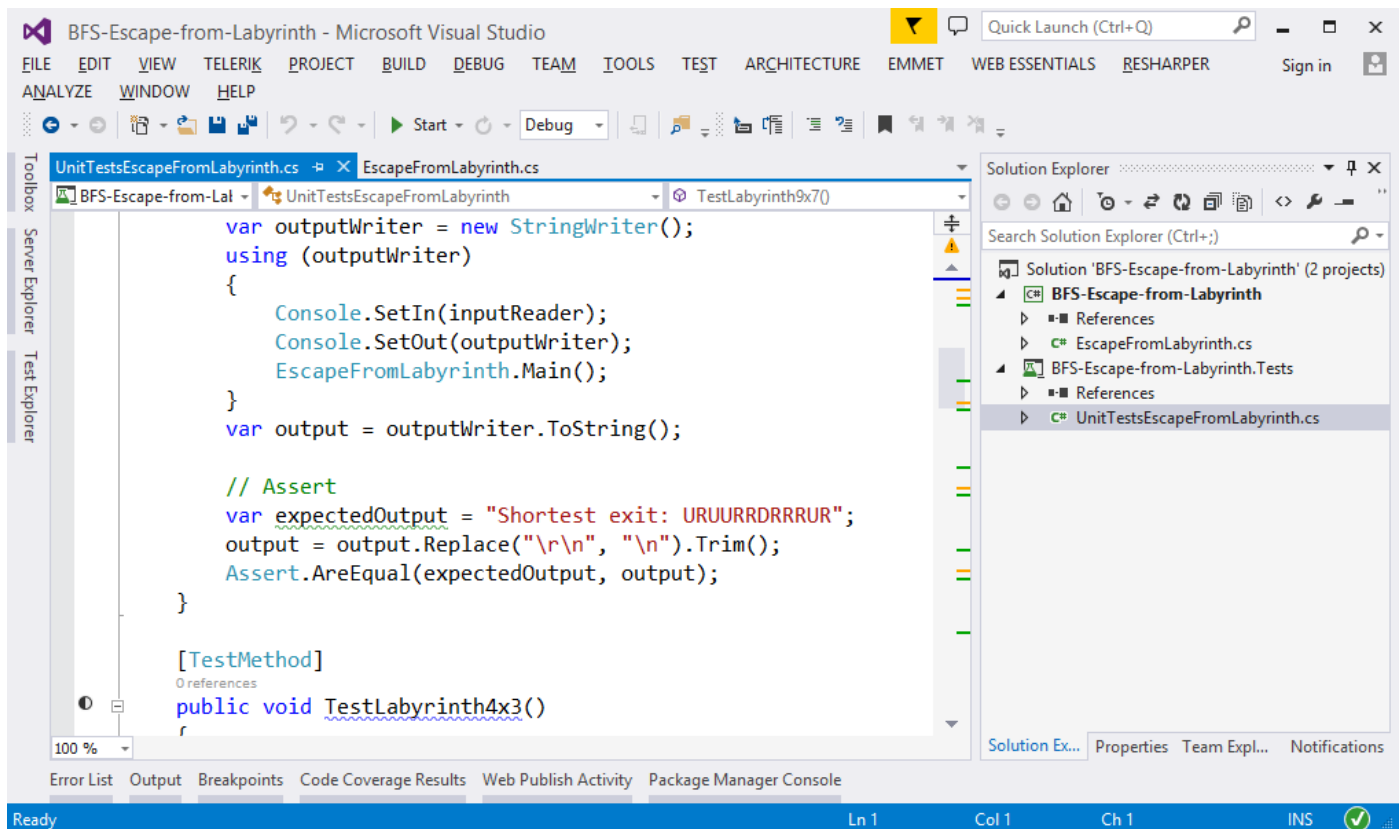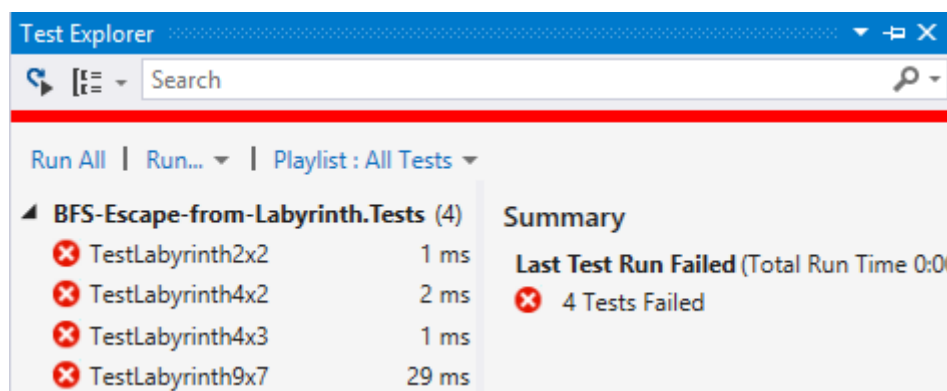
Problem 2. Run the Unit Tests to Ensure They Initially Fail

Run the unit tests from the **BFS-Escape-from-Labyrinth.Tests** project. Open the "**Test Explorer**" window (Menu → Test → Windows → Test Explorer) and run all tests. The expected behavior is that all tests should fail:

This is quite normal. We have unit tests, but the code covered by these tests is missing. Let's write it.

Problem 3. Define a Sample Labyrinth

The first step is to define a sample labyrinth. It will be used to test the code during the development:

```
static int width = 9;
static int height = 7;
static char[,] labyrinth =
{
    { '*',  '*',  '*',  '*',  '*',  '*',  '*',  '*',  '*' },
    { '*',  '_',  '_',  '_',  '_',  '*',  '*',  '_',  '_' },
    { '*',  '*',  '_',  '*',  '_',  '_',  '_',  '_',  '*' },
    { '*',  '_',  '_',  '*',  '_',  '*',  '_',  '*',  '*' },
    { '*',  's',  '*',  '_',  '_',  '*',  '_',  '*',  '*' },
    { '*',  '*',  '_',  '_',  '_',  '_',  '_',  '_',  '*' },
    { '*',  '*',  '*',  '*',  '*',  '*',  '*',  '_',  '*' },
};
```

This sample data will be used to test the code we write instead of entering the labyrinth each time we run the program.

Problem 4. Define the Class Point

We will define the class **Point** to hold a cell in the labyrinth (**x** and **y** coordinates). It will also hold the **direction** of move (Left / Right / Up / Down) used to come to this cell, as well as the previous cell. In fact, the class **Point** is a **linked list** that holds a cell in the labyrinth along with a link to the previous cell:

```
class Point
{
    0 references
    public int X { get; set; }
    0 references
    public int Y { get; set; }
    0 references
    public string Direction { get; set; }
    0 references
    public Point PreviousPoint { get; set; }
}
```

Problem 5. Implement the BFS Algorithm

The next step is to implement the **BFS** (Breadth-First-Search) algorithm to traverse the labyrinth starting from a specified cell:

```
static string FindShortestPathToExit()
{
    var queue = new Queue<Point>();
    var startPosition = FindStartPosition();
    if (startPosition == null)
    {
        // No start position -> no exit
        return null;
    }
    queue.Enqueue(startPosition);
    while (queue.Count > 0)
    {
        var currentCell = queue.Dequeue();
        //Console.WriteLine("Visited cell: ({0}, {1})", currentCell.X, currentCell.Y);
        if (IsExit(currentCell))
        {
            return TracePathBack(currentCell);
        }
        TryDirection(queue, currentCell, "U", 0, -1);
        TryDirection(queue, currentCell, "R", +1, 0);
        TryDirection(queue, currentCell, "D", 0, +1);
        TryDirection(queue, currentCell, "L", -1, 0);
    }

    return null;
}
```

This is **classical implementation of BFS**. It first puts in the queue the start cell. Then, while the queue is not empty, the BFS algorithm takes the next cell from the queue and puts its all unvisited neighbors (left, right, up and left). If, at some moment, an exit is reached (a cell at some of the labyrinth sides), the algorithm returns the path found.

The above code has several missing pieces: finding the start position, checking if a cell is an exit, adding a neighbor cell to the queue, and printing the path found (a sequence of cells).

Problem 6. Find the Start Cell

Finding the start position (cell) is trivial. Just scan the labyrinth and find the **'s'** cell in it:

```
const char VisitedCell = 's';
```

```
static Point FindStartPosition()
{
    for (int x = 0; x < width; x++)
    {
        for (int y = 0; y < height; y++)
        {
            if (labyrinth[y, x] == VisitedCell)
            {
                return new Point() { X = x, Y = y };
            }
        }
    }

    return null;
}
```

Problem 7. Check If a Cell is at the Exit

Checking whether a cell is at the exit from the labyrinth is simple. We just check whether the cell is at the left, right, top or bottom sides:

```
static bool IsExit(Point currentCell)
{
    bool isOnBorderX = currentCell.X == 0 || currentCell.X == width - 1;
    bool isOnBorderY = currentCell.Y == 0 || currentCell.Y == height - 1;
    return isOnBorderX || isOnBorderY;
}
```

Problem 8. Try the Neighbor Cell in Given Direction

Now, write the code to try to visit the neighbor cell in given direction. The method takes an existing cell (e.g. {3, 5}), a direction (e.g. right {+1, 0}). It checks whether the cell in the specified direction exists and is empty '-'. Then, the cell is changed to "not empty" and is appended in the queue. To preserve the path to this cell, it remembers the previous cell (point) and move direction. See the code below:

```
static void TryDirection(Queue<Point> queue, Point currentCell,
    string direction, int deltaX, int deltaY)
{
    int newX = currentCell.X + deltaX;
    int newY = currentCell.Y + deltaY;
    if (newX >= 0 && newX < width && newY >= 0 && newY < height && labyrinth[newY, newX] == '-')
    {
        labyrinth[newY, newX] = VisitedCell;
        var nextCell = new Point()
        {
            X = newX,
            Y = newY,
            Direction = direction,
            PreviousPoint = currentCell
        };
        queue.Enqueue(nextCell);
    }
}
```

Problem 9. Recover the Path from the Exit to the Start

In case an exit is found, we need to trace back the path from the exit to the start. To recover the path, we start from the exit, then go to the previous cell (in the linked list we build in the BFS algorithm), then to the previous, etc. until we reach the start cell. Finally, we need to reverse the back, because it is reconstructed from the end to the start:

```
static string TracePathBack(Point currentCell)
{
    var path = new StringBuilder();
    while (currentCell.PreviousPoint != null)
    {
        path.Append(currentCell.Direction);
        currentCell = currentCell.PreviousPoint;
    }
    var pathReversed = new StringBuilder(path.Length);
    for (int i = path.Length - 1; i >= 0; i--)
    {
        pathReversed.Append(path[i]);
    }
    return pathReversed.ToString();
}
```

Problem 10. Test the BFS Algorithm

Now, test whether the BFS algorithm implementation for finding the exit from a labyrinth:

```csharp
public static void Main()
{
    string shortestPathToExit = FindShortestPathToExit();
    if (shortestPathToExit == null)
    {
        Console.WriteLine("No exit!");
    }
    else if (shortestPathToExit == "")
    {
        Console.WriteLine("Start is at the exit.");
    }
    else
    {
        Console.WriteLine("Shortest exit: " + shortestPathToExit);
    }
}
```
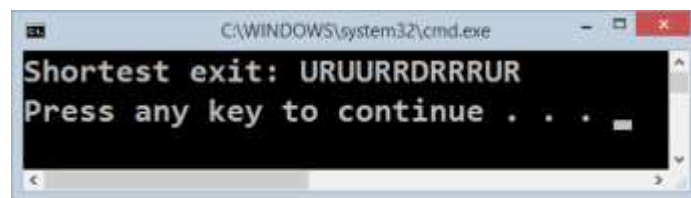
The method **FindShortestPathToExit()** returns a value that has three cases:

- **null** → exit not found
- **""** → the path is empty → the start is at the exit
- non-empty string → the path is returned as sequence of moves

So, let's test the code. Run it ([Ctrl] + [F5]):


```
C:\WINDOWS\system32\cmd.exe
Shortest exit: URUURRDRRRUR
Press any key to continue . . .
```

Problem 11. Read the Input Data from the Console

Usually, when we solve problems, we work on hard-coded sample data (in our case the **labyrinth** is hard-coded) and we write the code step by step, test it continuously and finally, when the code is ready and it works well, we change the hard-coded input data with a logic that reads it. Let's implement the data entry logic (read the labyrinth from the console):

```csharp
static void ReadLabyrinth()
{
    width = int.Parse(Console.ReadLine());
    height = int.Parse(Console.ReadLine());
    labyrinth = new char[height, width];
    for (int row = 0; row < height; row++)
    {
        // TODO: Read and parse the next labyrinth line
    }
}
```

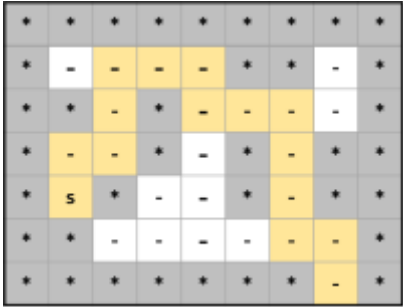The code above is unfinished. You need to write it.

Modify the main method to read the labyrinth from the console instead of using the hard-coded labyrinth:
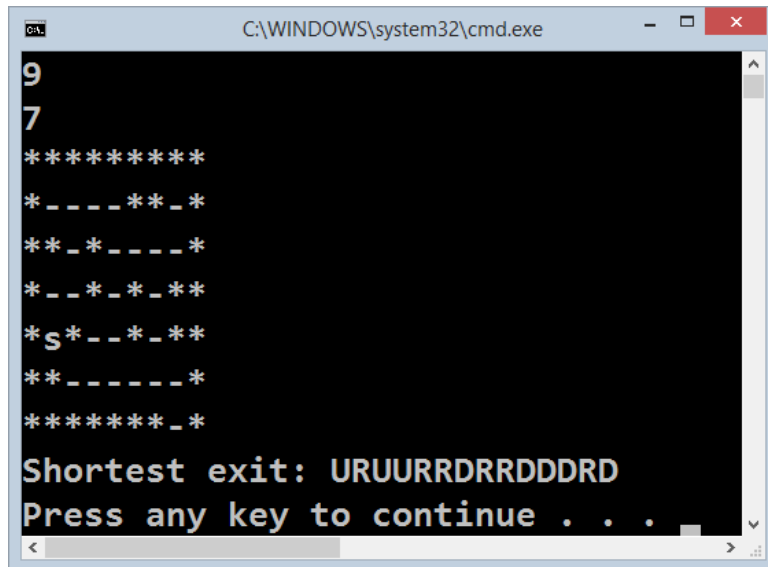
```
public static void Main()
{
    ReadLabyrinth();
    ...
}
```

Now test the program. Run it ([Ctrl] + [F5]). Enter a sample graph data and check the output:

Input	Labyrinth	Output
9 7 * * * * * * * * * * _ _ _ _ * * _ * * * _ * _ _ _ _ * * _ _ * _ * _ * * * s * _ _ * _ * * * * _ _ _ _ _ _ * * * * * * * _ *		Shortest exit: URUURRDRRDDDRD

Seems like it runs correctly:

```
9
7
*********
*____**_*
**_*____*
*__*_*_**
*s*__*_**
**_____*
*******_*
Shortest exit: URUURRDRRDDDRD
Press any key to continue . . .
```
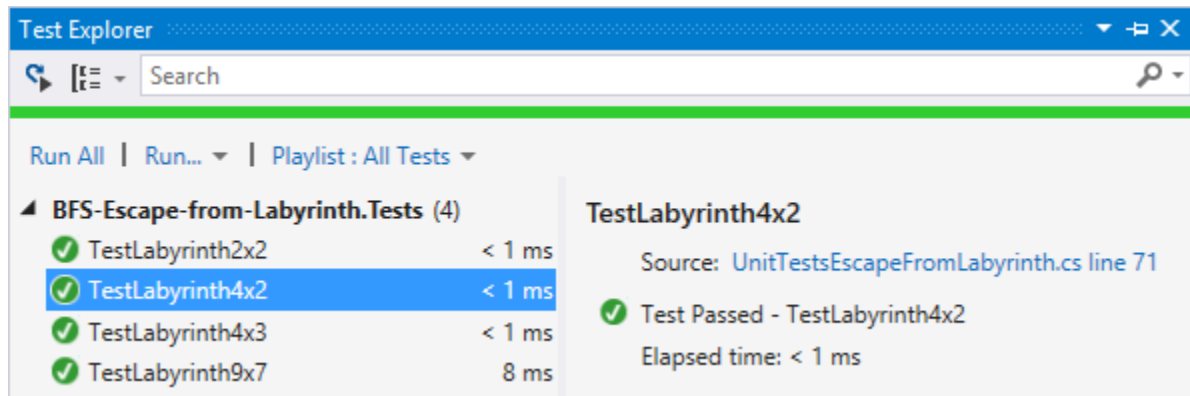
We are ready for the unit tests.

Problem 12. Run the Unit Tests

Seems like we solved the labyrinth-escaping problem. Let's run the unit tests that come with the program skeleton:

Follow us:

All tests pass. Experienced developers will tell: "**_hold the line green to keep the code clean_**".

Congratulations! You have implemented the BFS-based escape from labyrinth algorithm.

Follow us: