

Exercises: Trees Representation and Traversal (BFS, DFS)

Problems for exercises and homework for the "Data Structures and Algorithms Basics" course from the official "Applied Programmer" curriculum.

You can check your solutions here: <https://judge.softuni.bg/Contests/2933/Trees-BFS-and-DFS-Exercises>

Use the provided skeleton!

1. Find File on Hard Drive with DFS

Use the **DFS algorithm** to traverse the **hard drive** on your computer and search for a **file** with a given name. For example, paste this exercise's **.docx** file from in a folder called **SoftUni**.

You can use the code from "**DFS Traverse Folders and Files**" task from **Trees-BFS-DFS-Lab.docx**. If you don't know how to implement the DFS algorithms, use the steps from the document.

Use **file.Name** property for searching for a file with a given name. If you find the file, print "**{file.Name} is found in {dir.FullName}.**" on the console.

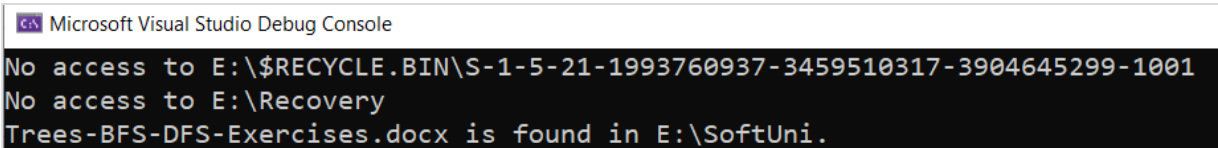
Also, note that our console app doesn't have **access** to all of our folders. Add a **try-catch** block, so that exceptions don't stop the program from running, like this:

```
private static void TraverseDirDFS(DirectoryInfo dir, string fileName)
{
    try
    {
        //Write logic
    }
    catch
    {
        Console.WriteLine($"No access to {dir}");
    }
}
```

If we try to find the "**Trees-BFS-DFS-Exercises.docx**" file in the **SoftUni** folder, we should give the name of the **hard drive** (yours may be different from the given one) and the **file's name** to our method. In our case this is the **TraverseDirDFS()** method:

```
static void Main()
{
    TraverseDirDFS(@"E:\", "06.Trees-BFS-DFS-Exercises.docx");
}
```

The result looks like this:



```
Microsoft Visual Studio Debug Console
No access to E:\$RECYCLE.BIN\S-1-5-21-1993760937-3459510317-3904645299-1001
No access to E:\Recovery
Trees-BFS-DFS-Exercises.docx is found in E:\SoftUni.
```

As you can see, we **couldn't access** folders twice, but then we found our file.

Test your code for your **hard drive** and **files**. The searched file should be found, if present in the file system, and result should be printed on the console.

2. Find the Nearest Exit from a Labyrinth

This task 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: **URUURDRRRUR** (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 **URUURDRRRUR** is the shortest.

*	*	*	*	*	*	*	*	*
*	-	-	-	-	*	*	-	-
*	*	-	*	-	-	-	-	*
*	-	-	*	-	*	-	*	*
*	s	*	-	-	*	-	*	*
*	*	-	-	-	-	-	-	*
*	*	*	*	*	*	*	-	*

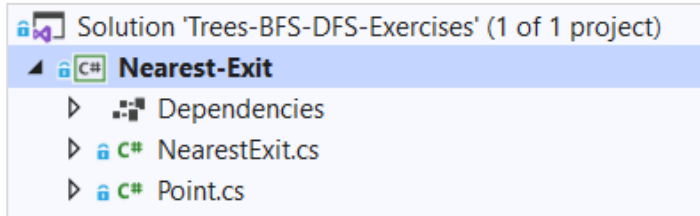
The input comes from the console. The first line holds the labyrinth **width**. The second line holds the labyrinth **height**. The next height lines hold the labyrinth cells – characters '*' (wall), '-' (empty cell) or 's' (start cell).

Examples

Input	Labyrinth	Output																																																															
9 7 ***** * _ _ _ * _ _ * _ * _ _ _ * * _ * _ _ _ * * _ * _ _ _ * * S _ _ _ _ * * _ * _ _ _ * ***** _ *	<table><tr><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td></tr><tr><td>*</td><td>-</td><td>-</td><td>-</td><td>-</td><td>*</td><td>*</td><td>-</td><td>-</td></tr><tr><td>*</td><td>*</td><td>-</td><td>*</td><td>-</td><td>-</td><td>-</td><td>-</td><td>*</td></tr><tr><td>*</td><td>-</td><td>-</td><td>*</td><td>-</td><td>*</td><td>-</td><td>*</td><td>*</td></tr><tr><td>*</td><td>S</td><td>*</td><td>-</td><td>-</td><td>*</td><td>-</td><td>*</td><td>*</td></tr><tr><td>*</td><td>*</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td><td>*</td></tr><tr><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>-</td><td>*</td></tr></table>	*	*	*	*	*	*	*	*	*	*	-	-	-	-	*	*	-	-	*	*	-	*	-	-	-	-	*	*	-	-	*	-	*	-	*	*	*	S	*	-	-	*	-	*	*	*	*	-	-	-	-	-	-	*	*	*	*	*	*	*	*	-	*	Shortest exit: URUURDRRRUR
*	*	*	*	*	*	*	*	*																																																									
*	-	-	-	-	*	*	-	-																																																									
*	*	-	*	-	-	-	-	*																																																									
*	-	-	*	-	*	-	*	*																																																									
*	S	*	-	-	*	-	*	*																																																									
*	*	-	-	-	-	-	-	*																																																									
*	*	*	*	*	*	*	-	*																																																									
4 3 **** * - S * ****	<table><tr><td>*</td><td>*</td><td>*</td><td>*</td></tr><tr><td>*</td><td>-</td><td>S</td><td>*</td></tr><tr><td>*</td><td>*</td><td>*</td><td>*</td></tr></table>	*	*	*	*	*	-	S	*	*	*	*	*	No exit!																																																			
*	*	*	*																																																														
*	-	S	*																																																														
*	*	*	*																																																														
4 2 **** ***S	<table><tr><td>*</td><td>*</td><td>*</td><td>*</td></tr><tr><td>*</td><td>*</td><td>*</td><td>S</td></tr></table>	*	*	*	*	*	*	*	S	Start is at the exit.																																																							
*	*	*	*																																																														
*	*	*	S																																																														

Escape from Labyrinth – Project Skeleton

You are given a **Visual Studio project skeleton** (unfinished project) holding the Point class and the unfinished class **EscapeFromLabyrinth**. The project holds the following assets:



The unfinished **NearestExit** class stays in the file **NearestExit.cs**.

Define a Sample Labyrinth

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

```
public class NearestExit
{
    static int width = 9;
    static int height = 7;
    static char[,] labyrinth =
    {
        { '*', '*', '*', '*', '*', '*', '*', '*', '*' },
        { '*', '-', '-', '-', '-', '*', '*', '-', '*' },
        { '*', '*', '-', '*', '-', '-', '-', '-', '*' },
        { '*', '-', '-', '*', '-', '*', '-', '-', '*' },
        { '*', 'S', '*', '-', '-', '*', '-', '-', '*' },
        { '*', '*', '-', '-', '-', '-', '-', '-', '*' },
        { '*', '*', '*', '*', '*', '*', '*', '-', '*' }
    };
    static char VisitedCell = 's';

    0 references
    public static void Main()
```

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

Examine the Point Class

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
{
    5 references
    public int X { get; set; }
    5 references
    public int Y { get; set; }
    2 references
    public string Direction { get; set; }
    3 references
    public Point PreviousPoint { get; set; }
}
```

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();
        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).

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;
}
```

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;
}
```

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);
    }
}
```

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();
}
```

Test the BFS Algorithm

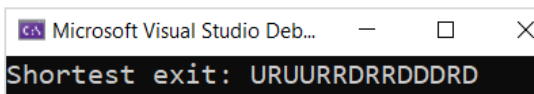
Now, **test** whether the BFS algorithm implementation for finding the exit from a labyrinth:

```
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**]):



```
Shortest exit: URUURDRRDDDDRD
```

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):

```
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.

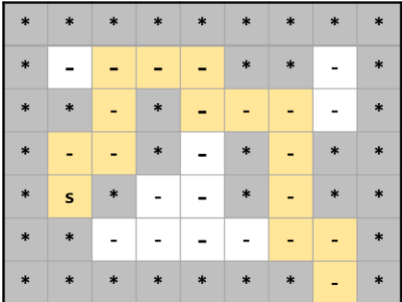
Then, remove the hard-coded values of **fields**, connected to the labyrinth:

```
static int width;
static int height;
static char[,] labyrinth;
static char VisitedCell = 's';
```

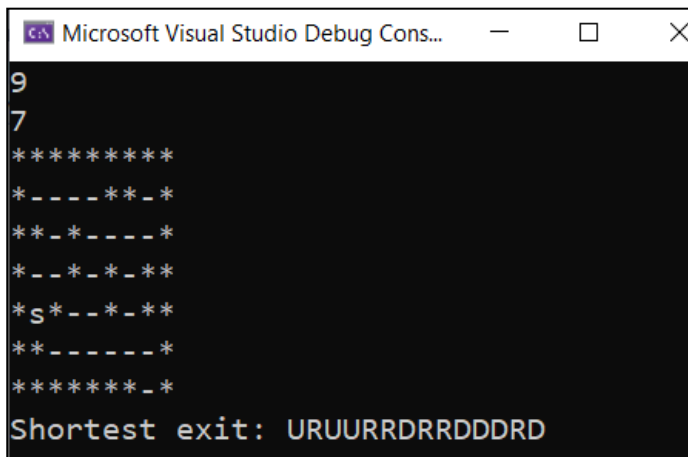
Modify the **Main()** method to **read** the labyrinth from the console instead of using the hard-coded labyrinth:

```
public static void Main()
{
    ReadLabyrinth();
    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);
    }
}
```

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: URUURDRRDDDRD

Seems like it runs correctly:



```

9
7
*****
*-----*
**_*-----*
*--*_*_*--*
*s*--*_*_*
*-----*
*****_*
Shortest exit: URUURDRRDDDRD
  
```

Test with other sample inputs and outputs, as well.

3. Largest Connected Area

Find the **largest connected area** in a matrix, using the **DFS** algorithm (recursive depth-first search). The largest connected area is composed of '1', which are **next** to each other (on the right, left, up or down, but not in diagonal). For better understanding, look at the example.

Input

You should read **several lines** from the console:

- On the first line, the **height** (number of rows) of the matrix
- On the second line, the **width** (number of columns) of the matrix
- On the next lines, the **matrix** itself

Output

Output is on a single line: "The largest connected area of the matrix is: { max connected area size }"

Examples

Input	Matrix	Output																																																																						
7 91111..1. ..1.1111. .11.1.1.. .1.11.1.. ..111111.1.	<table><tr><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td></tr><tr><td>.</td><td>1</td><td>1</td><td>1</td><td>1</td><td>.</td><td>.</td><td>1</td><td>.</td><td>.</td></tr><tr><td>.</td><td>.</td><td>1</td><td>.</td><td>1</td><td>1</td><td>1</td><td>1</td><td>.</td><td>.</td></tr><tr><td>.</td><td>1</td><td>1</td><td>.</td><td>1</td><td>.</td><td>1</td><td>.</td><td>.</td><td>.</td></tr><tr><td>.</td><td>1</td><td>.</td><td>1</td><td>1</td><td>.</td><td>1</td><td>.</td><td>.</td><td>.</td></tr><tr><td>.</td><td>.</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>.</td><td>.</td></tr><tr><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>1</td><td>.</td><td>.</td></tr></table>	1	1	1	1	.	.	1	1	.	1	1	1	1	.	.	.	1	1	.	1	.	1	1	.	1	1	.	1	1	1	1	1	1	1	1	.	.	The largest connected area of the matrix is: 25
.																																																															
.	1	1	1	1	.	.	1	.	.																																																															
.	.	1	.	1	1	1	1	.	.																																																															
.	1	1	.	1	.	1	.	.	.																																																															
.	1	.	1	1	.	1	.	.	.																																																															
.	.	1	1	1	1	1	1	.	.																																																															
.	1	.	.																																																															

6			1	1						The largest connected area of the matrix is: 12
8										
..11....		1	1					1		
.11...1.										
..11....			1	1						
...1....				1						
...11.1.				1	1			1		
1..111..	1			1	1	1				

Hints

You can try to **implement** the algorithm on your own. However, if you have difficulties, you can use this **pseudocode** to write a solution. However, pseudocode only summarizes program's **flow**, but it is not real code and cannot be used directly.

```

Main() {
    ReadMatrix()
    for row = 0...size-1
        for col = 0...size-1
            int areaSize = FindArea(row, col)
            maxSize = max(maxSize, areaSize)

    print "The largest connected area of the matrix is: " + maxSize
}

ReadMatrix() {
    //Read the matrix from the console
}

FindArea(row, col) {
    if row >= size || row < 0 || col >= size || col < 0
        return 0;
    if (visited[row, col])
        return 0;
    if (matrix[row, col] != '-')
        return 0;

    visited[row, col] = true;
    var areaSize = 1;
    areaSize += FindArea(row+1, col);
    areaSize += FindArea(row-1, col);
    areaSize += FindArea(row, col+1);
    areaSize += FindArea(row, col-1);

    return areaSize;
}

```