Exercises: Trees and Graphs

You can check your solutions here: https://judge.softuni.bg/Contests/3187/Graphs-Trees.
Use the provided skeleton!

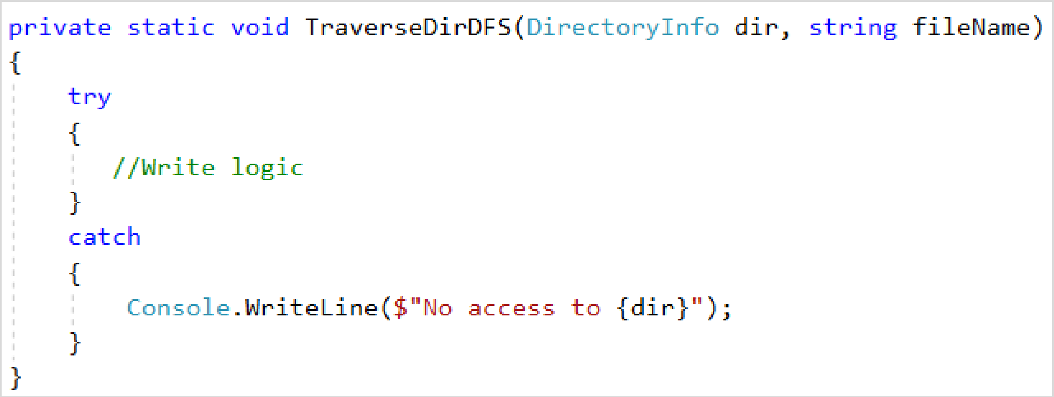
1. Find File On Hard Drive with DFS

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.

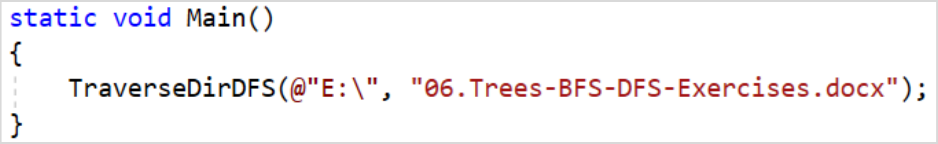
Use **`file.Name`** property for searching for a file with a given name. If you find the file, print "**`{file.Name} is found in {dir.FullName}.`**" on the console.

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:



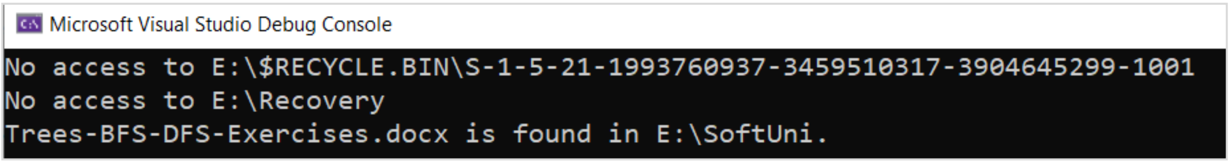
```
private static void TraverseDirDFS(DirectoryInfo dir, string fileName)
{
    try
    {
      //Write logic
    }
    catch
    {
       Console.WriteLine($"No access to {dir}");
    }
}
```

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:



```
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.

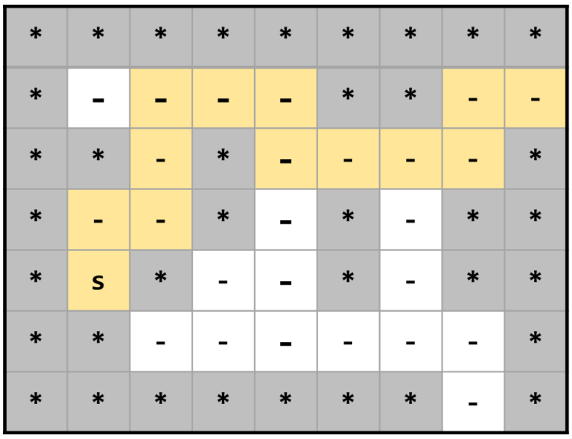
Test your code for your **hard drive** and **files**. The searched file should be found, if present in the file system, and result should be printed on the console.

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: **URUURRDRRRUR** (where

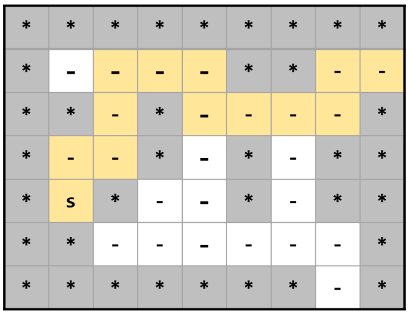
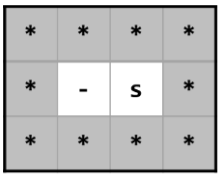
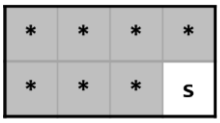
'**U**' means up, '**R**' means right, '**D**' means down and '**L**' means left). There are two exits and several other paths to these exits, but the path **URUURRDRRRUR** is the shortest.



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

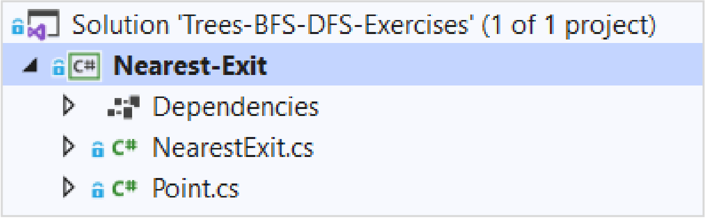
The input comes from the console. The first line holds the labyrinth **`width`**. The second line holds the labyrinth **`height`**. The next height lines hold the labyrinth cells – characters '**`*`**' (wall), '**`-`**' (empty cell) or '**`s`**' (start cell).

Examples

Input	Labyrinth	Output
9 7 ********* *----**-- **-*----* *--*-*-** *s*--*-** **------* *******-*		`Shortest exit: URUURRDRRRUR`
4 3 **** *-s* ****		`No exit!`
4 2 **** ***s		`Start is at the exit.`

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:

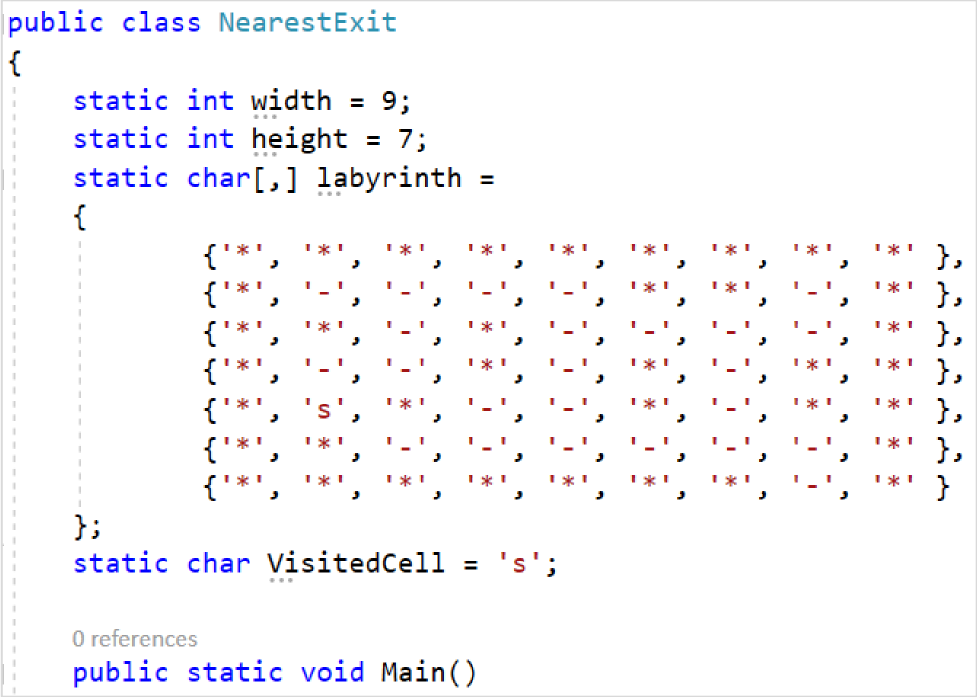


Solution 'Trees-BFS-DFS-Exercises' (1 of 1 project)
- Nearest-Exit
  - Dependencies
  - NearestExit.cs
  - Point.cs

The unfinished **`NearesExit`** class stays in the file **`NearesExit.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:

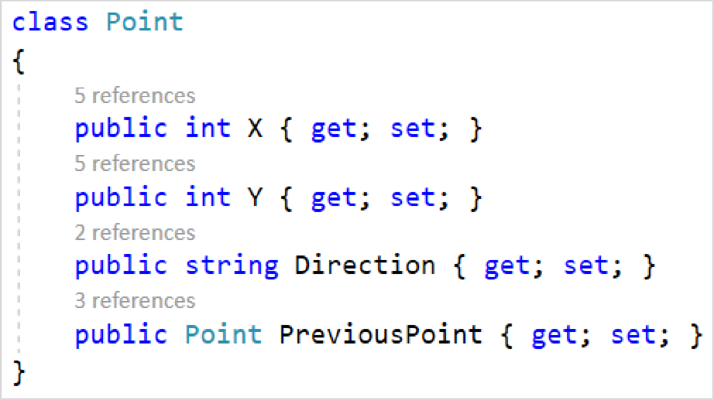


```
public class NearestExit
{
    static int width = 9;
    static int height = 7;
    static char[,] labyrinth =
    {
            {'*', '*', '*', '*', '*', '*', '*', '*', '*' },
            {'*', '-', '-', '-', '-', '*', '*', '-', '*' },
            {'*', '*', '-', '*', '-', '-', '-', '-', '*' },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

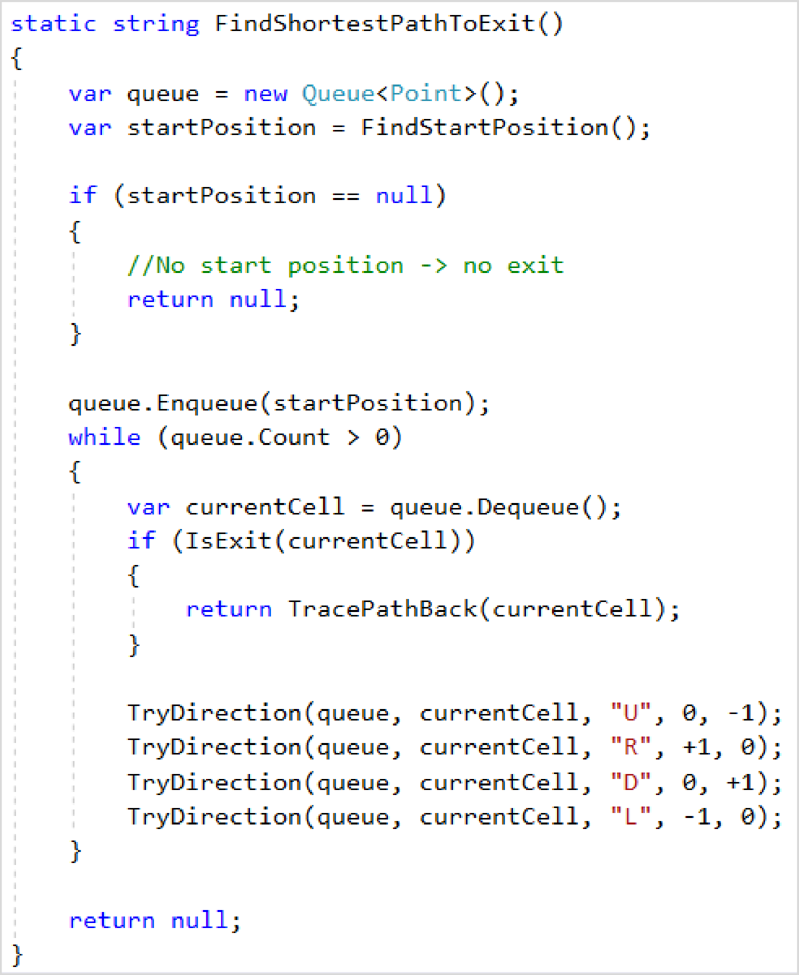
We will define the class **Point** to hold a cell in the labyrinth (**x** and **y** coordinates). It will also hold the **direction** of move (Left / Right / Up / Down) used to come to this cell, as well as the previous cell. In fact, the class **Point** is a **linked list** that holds a cell in the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



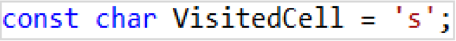
```
static string FindShortestPathToExit()
{
    var queue = new Queue<Point>();
    var startPosition = FindStartPosition();

    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.

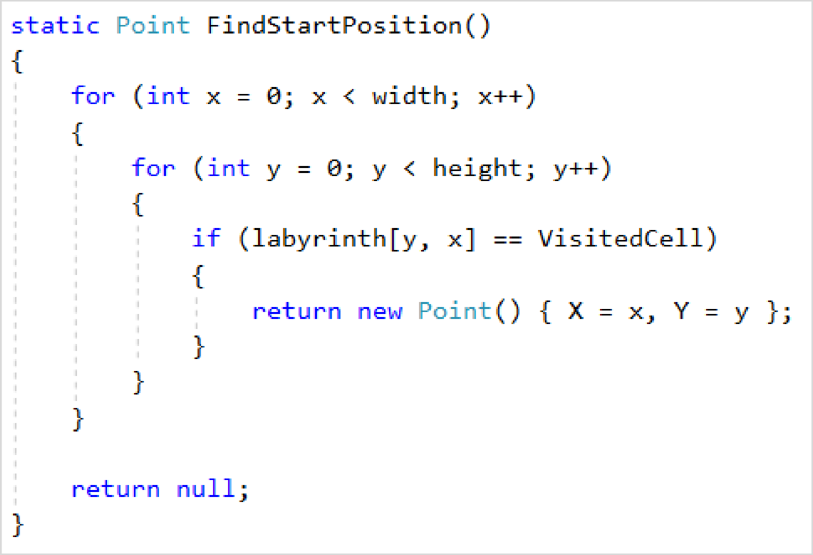
The above code has several missing pieces: finding the start position, checking if a cell is an exit, adding a neighbor cell to the queue, and printing the path found (a sequence of cells).

Find the Start Cell

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



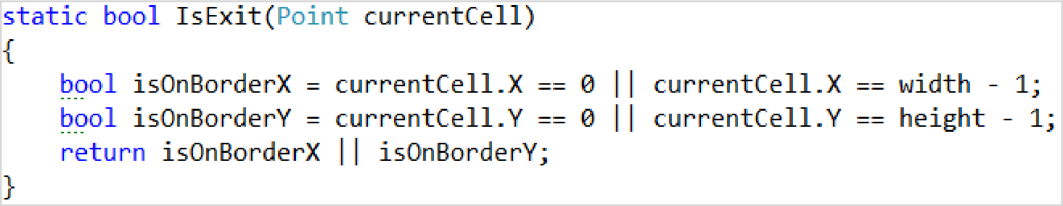
```
const char VisitedCell = 's';
```



```
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

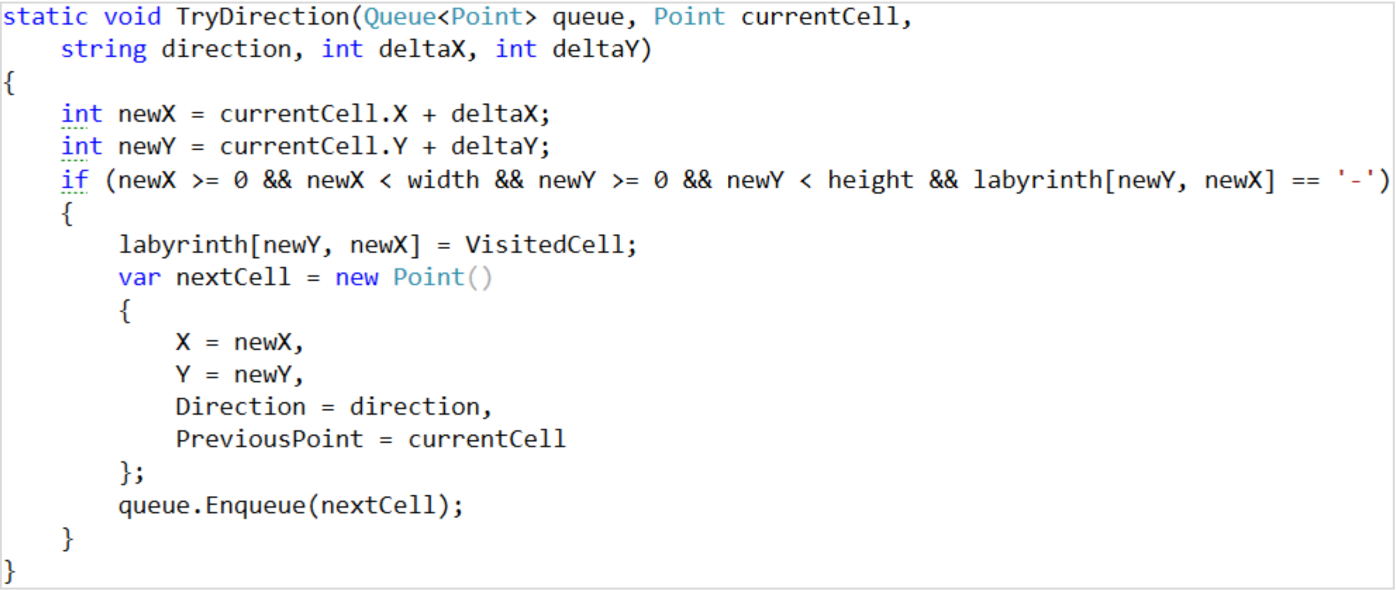
Checking whether a cell is at the **exit** from the labyrinth is simple. We just check whether the cell is at the left, right, top or bottom **sides**:



```
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

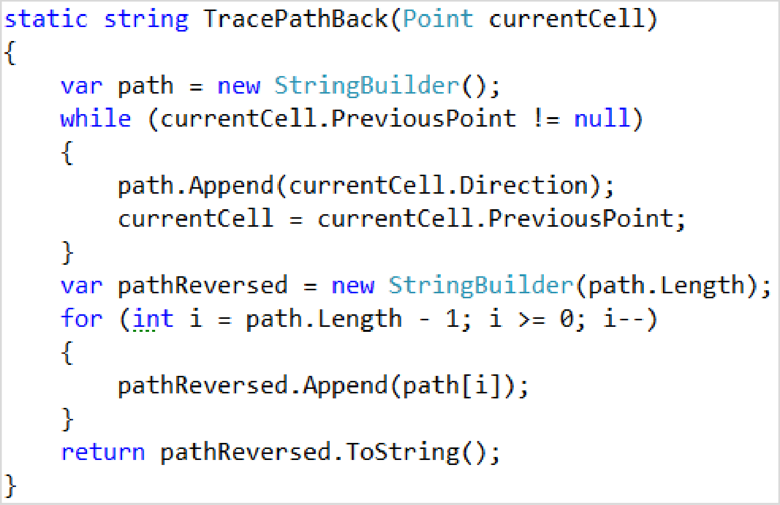
Now, write the code to try to visit the **neighbor cell** in given **direction**. The method takes an **existing cell** (e.g. {3, 5}), a **direction** (e.g. right {+1, 0}). It checks whether the cell in the specified direction exists and is empty '**-**'. Then, the cell is changed to "not empty" and 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

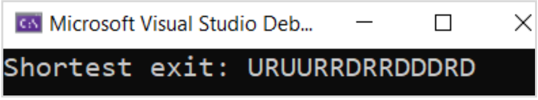
Now, **test** whether the BFS algorithm implementation for finding the exit from a labyrinth:

```
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: URUURRDRRDDDRD
```

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.

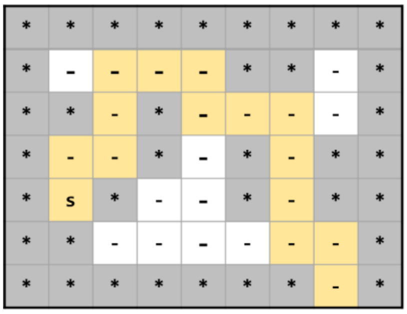
Then, remove the hard-coded values of **fields**, connected to the labyrinth:

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

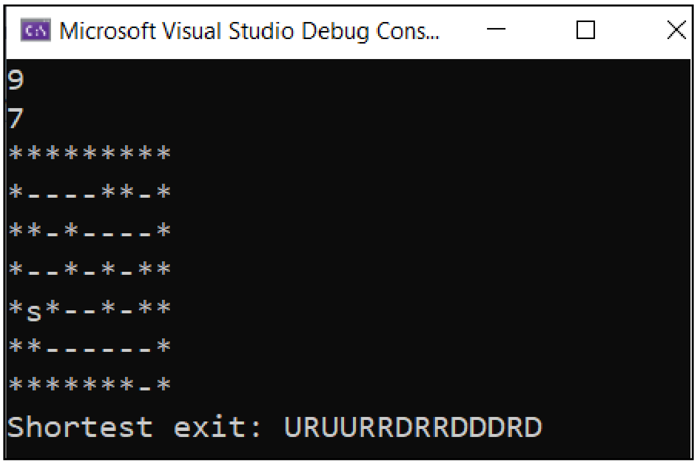
Modify the **`Main()`** method to **read** the labyrinth from the console instead of using the hard-coded labyrinth:

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

Seems like it runs correctly:



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

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
9
....
.1111..1.
..1.1111.
.11.1.1..
.1.11.1..
..111111.
.......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 .
	. . . . 1 .
		`The largest connected area of the matrix is: 25`

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

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;
}
```

4. Connected Components

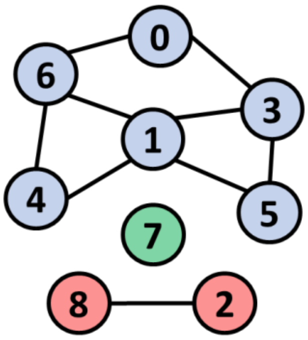
The first part of this exercise aims to implement the **DFS algorithm** (Depth-First-Search) to **traverse a graph** and find its **connected components** (nodes connected to each other either directly, or through other nodes). The graph nodes are numbered from **0** to **n-1**. The graph comes from the console in the following format:

- First line: number of lines **n**
- Next **n** lines: list of child nodes for the nodes **0** … **n-1** (separated by a space)

Print the connected components in the same format as in the examples below:

Example

Input	Graph	Output

9
3 6
3 4 5 6
8
0 1 5
1 6
1 3
0 1 4
 	
2
		Connected component: 6 4 5 1 3 0
		Connected component: 8 2
		Connected component: 7

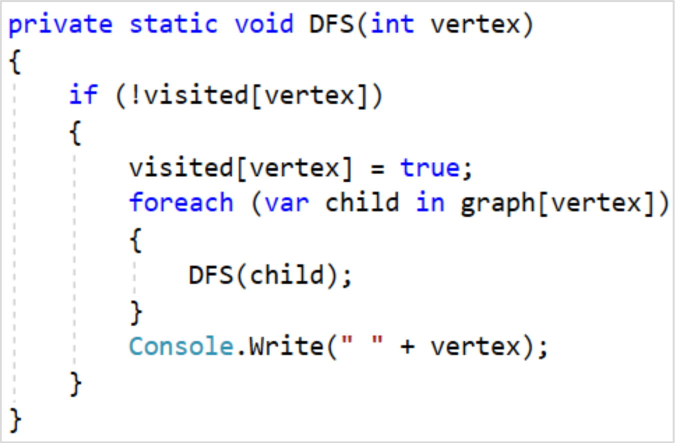
1 0		Connected component: 0

DFS Algorithm

First, create a **boolean array** that will be with the size of your graph.

```
private static bool[] visited;
```

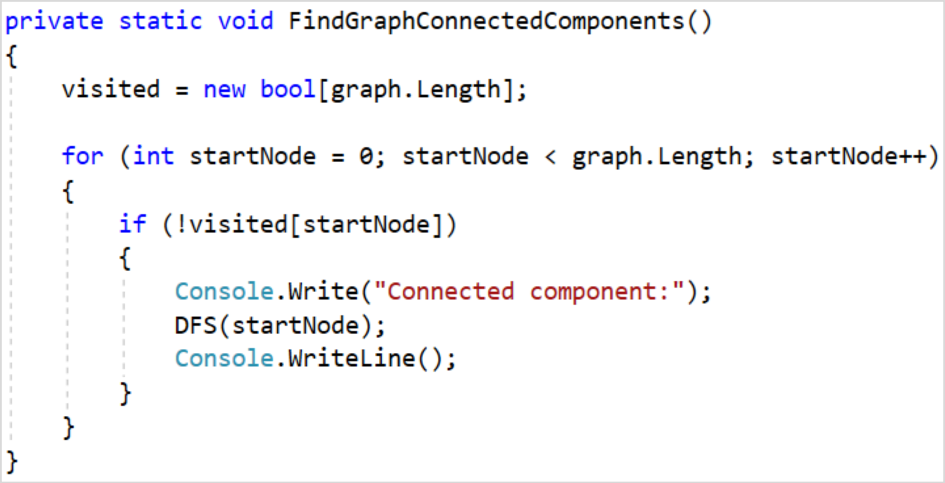
Next, implement the **DFS algorithm** (Depth-First-Search) to traverse all nodes connected to the specified start node:



```
private static void DFS(int vertex)
{
    if (!visited[vertex])
    {
        visited[vertex] = true;
        foreach (var child in graph[vertex])
        {
            DFS(child);
        }
        Console.Write(" " + vertex);
    }
}
```

Find All Components

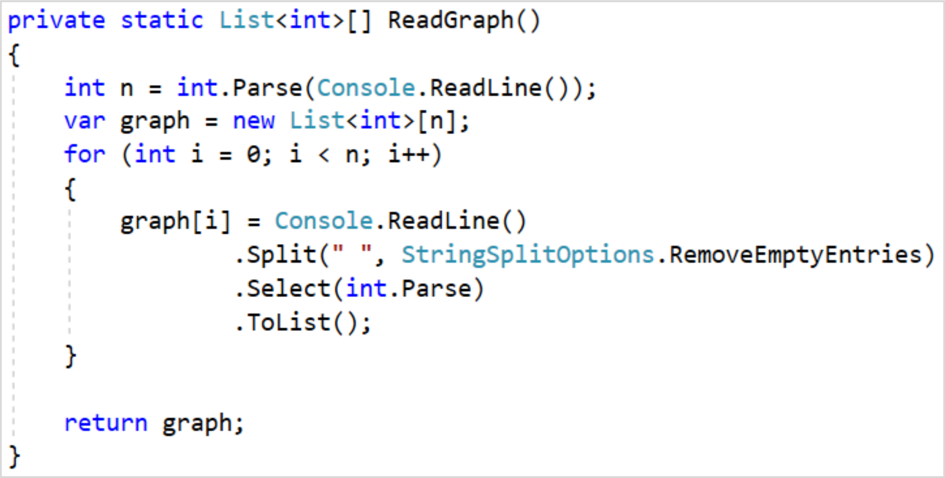
We want to **find all connected components**. We can just run the DFS algorithm for each node taken as a start (which was not visited already):



```
private static void FindGraphConnectedComponents()
{
    visited = new bool[graph.Length];

    for (int startNode = 0; startNode < graph.Length; startNode++)
    {
        if (!visited[startNode])
        {
            Console.Write("Connected component:");
            DFS(startNode);
            Console.WriteLine();
        }
    }
}
```

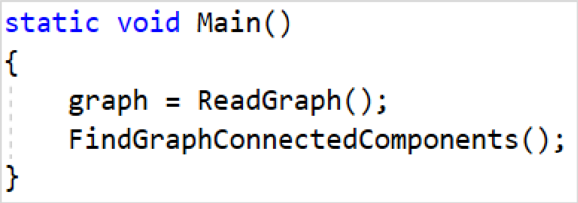
Read Input

Let's implement the data entry logic (**read graph** from the console). We already have the method below:



```
private static List<int>[] ReadGraph()
{
    int n = int.Parse(Console.ReadLine());
    var graph = new List<int>[n];
    for (int i = 0; i < n; i++)
    {
        graph[i] = Console.ReadLine()
                .Split(" ", StringSplitOptions.RemoveEmptyEntries)
                .Select(int.Parse)
                .ToList();
    }

    return graph;
}
```

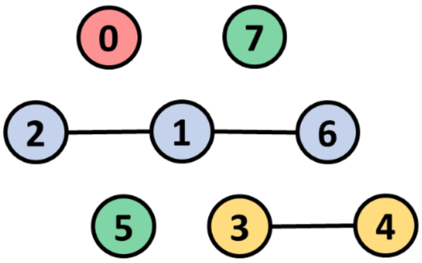
The main method should **read the graph from the console**:



```
static void Main()
{
    graph = ReadGraph();
    FindGraphConnectedComponents();
}
```

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

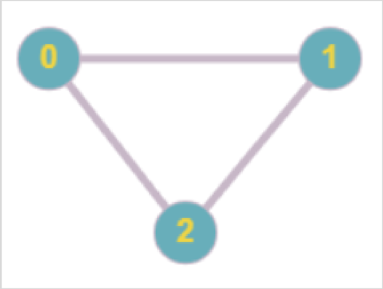
Input	Graph	Expected Output

7
 	
2 6
1
4
3
 	
1
		`Connected component: 0`
		`Connected component: 2 6 1`
		`Connected component: 4 3`
		`Connected component: 5`
		`Connected component: 7`

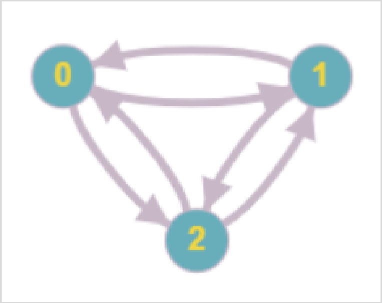
Test with other examples from above, as well.

5. Cycles in a Graph

Our task is to find if there is a **cycle** in a given **undirected graph**, using the **DFS** algorithm. An **undirected** graph is graph, where all the **edges are bidirectional**. For example, in the graph shown below there is an undirected edge between vertex 0 and vertex 1 – this means that there is a directed edge from vertex 0 to vertex 1 and from vertex 1 to vertex 0.



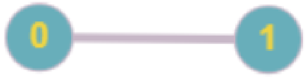
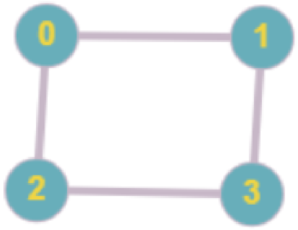
In this case, if we create a **directed graph** from the undirected one shown above, it will look like this:



Now, let's find out how to check whether the graph contains a **cycle** (a non-empty trail in which the only repeated vertices are the first and last vertices). For more clearance, look at the examples below.

Examples

Input	Picture	Output

2 1 0-1		`Graph doesn't contain cycle`
4 4 0-1 1-2 2-3 0-2		`Graph contains cycle`

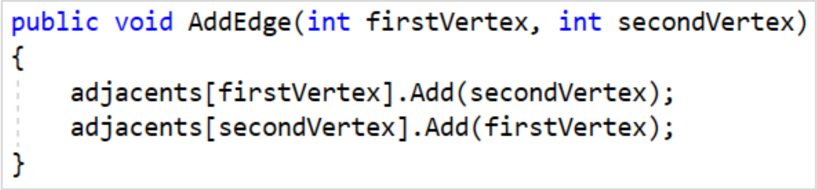
Create Graph

First, create a class **Graph** with **fields** for the count of vertices and for the adjacents. You should also create a **constructor** for the Graph initialization. We have done this before and you should know how to do it. If you have difficulties, look for instructions in "**Graphs-Lab.docx**" document.

Read the Graph

In this task, we need to **read** the graph with its vertices and edges from the console. Create a **ReadGraph()** method and use it to initialize the graph from the **Main()**. The method should read the count of vertices, the count of edges and the edges (the first vertex and the second vertex are separated by "**-**") from the console. We have done this as well in the "**Graphs-Lab.docx**".

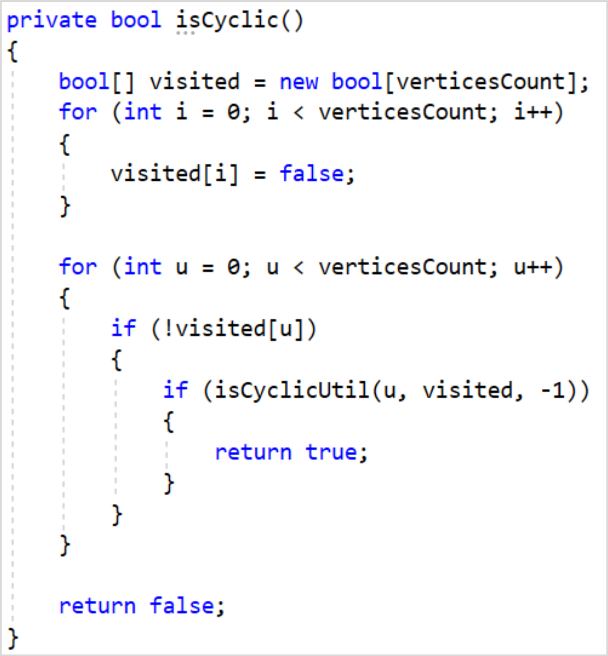
Note that our **ReadGraph()** method should invoke the **AddEdge()** method for adding edges to the graph. However, it is a bit different when we have an **undirected** graph. In the **AddEdge()** method, we should create an edge from the first to the second vertex and from the second to the first one (this way we have an undirected edge). The method should look like this:



```
public void AddEdge(int firstVertex, int secondVertex)
{
    adjacents[firstVertex].Add(secondVertex);
    adjacents[secondVertex].Add(firstVertex);
}
```

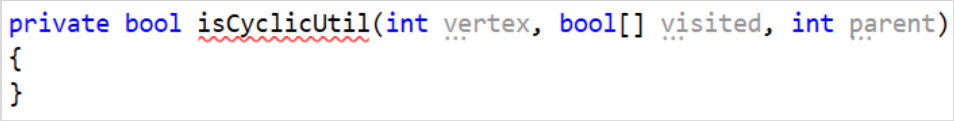
Create the Algorithm

First, create an **isCyclic()** method, which returns a **boolean** (true/false whether there is a cycle or not). In it, create a **boolean array** for visited vertices and mark all vertices as **not visited**. Then, loop through vertices and invoke a **helper method** to detect cycle in different DFS trees for the ones, which have not been visited:



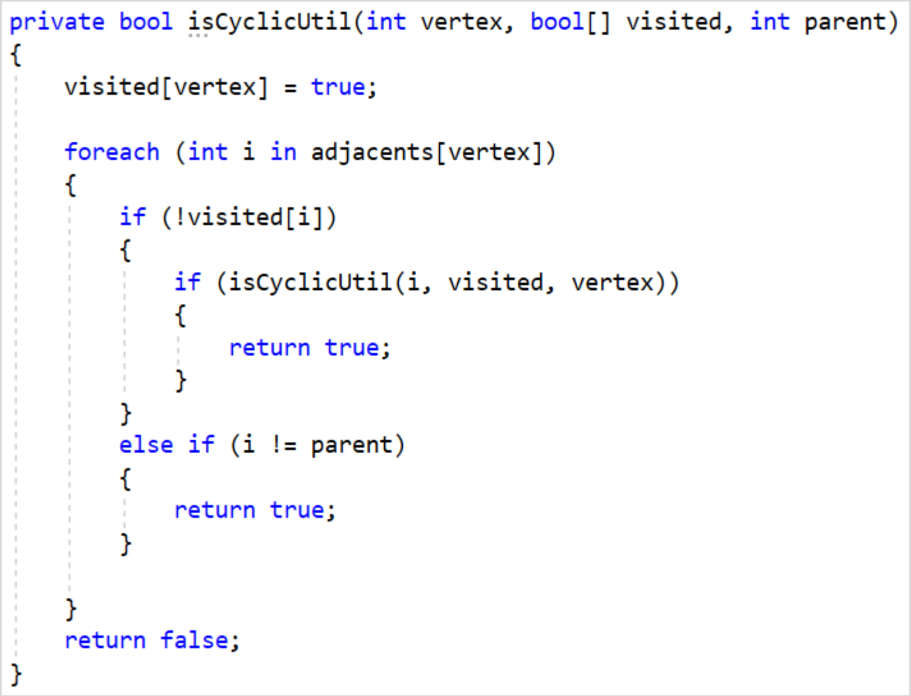
```
private bool isCyclic()
{
    bool[] visited = new bool[verticesCount];
    for (int i = 0; i < verticesCount; i++)
    {
        visited[i] = false;
    }

    for (int u = 0; u < verticesCount; u++)
    {
        if (!visited[u])
        {
            if (isCyclicUtil(u, visited, -1))
            {
                return true;
            }
        }
    }

    return false;
}
```

Create the **`isCyclicUtil()`** helper method. It accepts vertex, bool array of visited vertices and a parent vertex:



```
private bool isCyclicUtil(int vertex, bool[] visited, int parent)
{
}
```

In the method, mark the **current vertex** as **visited**. Then, loop through the **adjacent vertices** and check if the vertex has been visited. If an adjacent is **not visited**, then **recur** for that adjacent. If an adjacent is **visited** and **not a parent** of current vertex, then there is a **cycle**. These conditions should return a bool whether there is a cycle or not. The method should look like this:



```
private bool isCyclicUtil(int vertex, bool[] visited, int parent)
{
    visited[vertex] = true;

    foreach (int i in adjacents[vertex])
    {
        if (!visited[i])
        {
            if (isCyclicUtil(i, visited, vertex))
            {
                return true;
            }
        }
        else if (i != parent)
        {
            return true;
        }

    }
    return false;
}
```

Test Code

First, you should create the graph, using the **`ReadGraph()`** method. Then, check if the graph contains a cycle, using the **`isCyclic()`** method. If the method returns **`true`**, print "**`Graph contains cycle`**" on the console. If it returns **`false`**, print "**`Graph doesn't contain cycle`**".

Press **[Ctrl]** + **[F5]** to run the program and **test** if it works properly with the examples above.

6. Shortest Path

You are given an **undirected graph**. Write program to find the **shortest path length between two given nodes**. The shortest path length is the minimum number of **edges** between the nodes. Use the **DFS** algorithm.

Input

You need to **read the graph** from the console with the **start** and **end vertices** for the **path**:

- On the first line, you will receive **N**- number of **vertices**
- On the second line, read **M**- number of undirected **edges** between vertices
- On the next **M** lines, read the two vertices, connected by an edge in the format **"{firstVertex} {secondVertex}"**
- After that, on the next line, read the **startVertex** of the path
- On the last line, read the **endVertex** of the path

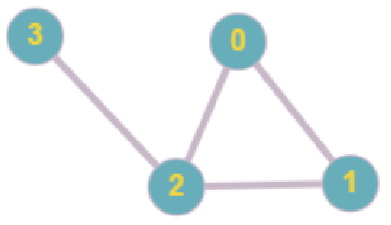
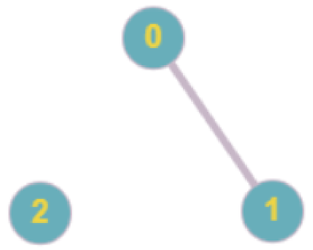
Find the **shortest path length** (number of edges) between the **startVertex** and the **endVertex**.

Output

On the first line, print "**Shortest path length from {startVertex} to {endVertex}:**" on the console. On the next line:

- If you find a **path**, print "**Path found. Length: {shortest path length}**".
- If there is **no path**, print "**No path exists**".

Examples

Input	Graph	Expected Output	Explanation
4 4 0 1 0 2 1 2 2 3 0 3		Shortest path length from 0 to 3: Path found. Length: 2	There are **4** vertices, connected by **4** edges. The path starts from **vertex 0** and ends in **vertex 3**. The shortest path is **0-2-3** and its length is **2** edges.
3 1 0 1 0 2		Shortest path length from 0 to 2: No path exists	There is no path between **vertex 0** and **vertex 2**.

Hints

You can use this article (https://www.geeksforgeeks.org/find-paths-given-source-destination), in which it is shown how to **print all paths** from a given source vertex to a destination one. Look at **explanations** and **modify the code** to work for the given problem.