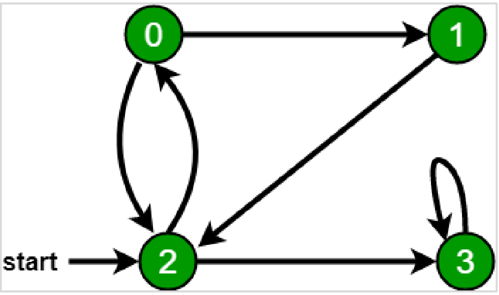
Lab: Graphs and Graph Algorithms

You can check your solutions here: https://judge.softuni.bg/Contests/3189/Additional-Exercises.

1. Traverse Graph with BFS

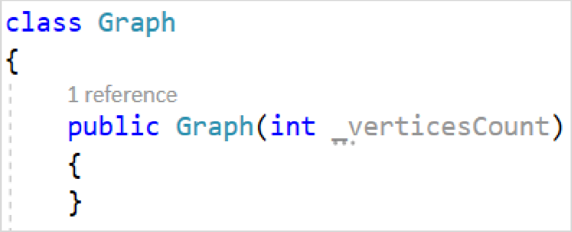
`Breadth First Search` for a graph is similar to BFS for a tree. The only catch here is, unlike trees, graphs may contain **cycles**, so we may come to the same node again. To avoid processing a node more than once, we use a boolean **`visited`** array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is **2, 0, 3, 1**.



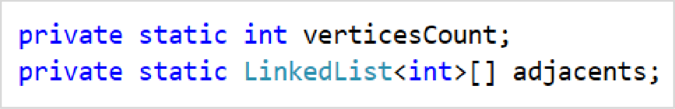
Now, let's implement the **BFS** algorithm and **traverse** the graph.

First, rename your **class** to **Graph** the following way and create a **constructor**, which accepts **vertices count**:



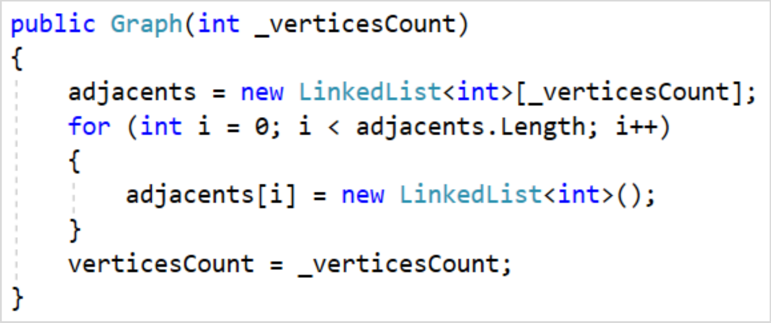
```
class Graph
{
    1 reference
    public Graph(int _verticesCount)
    {
    }
```

Also, you will need private **fields** to hold data for our graph- the **number of vertices** and a **collection of adjacents** (connected vertices).



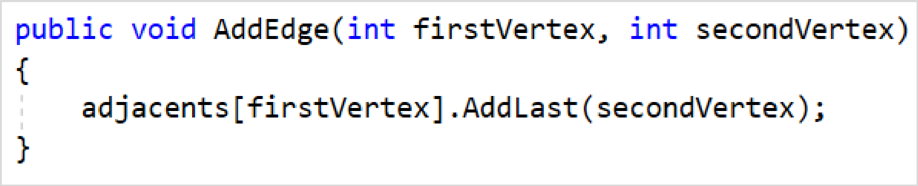
```
private static int verticesCount;
private static LinkedList<int>[] adjacents;
```

Now, create the **graph** structure in the constructor the following way:



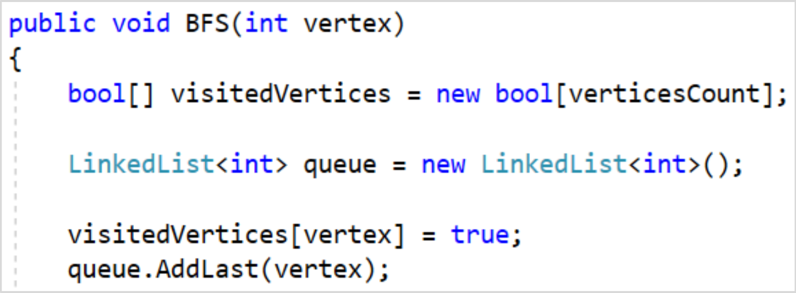
```
public Graph(int _verticesCount)
{
    adjacents = new LinkedList<int>[_verticesCount];
    for (int i = 0; i < adjacents.Length; i++)
    {
        adjacents[i] = new LinkedList<int>();
    }
    verticesCount = _verticesCount;
}
```

After we have created the graph with its vertices, we need to connect them with **edges**. Create a **method** for adding edges from a given **vertex** to a second one:



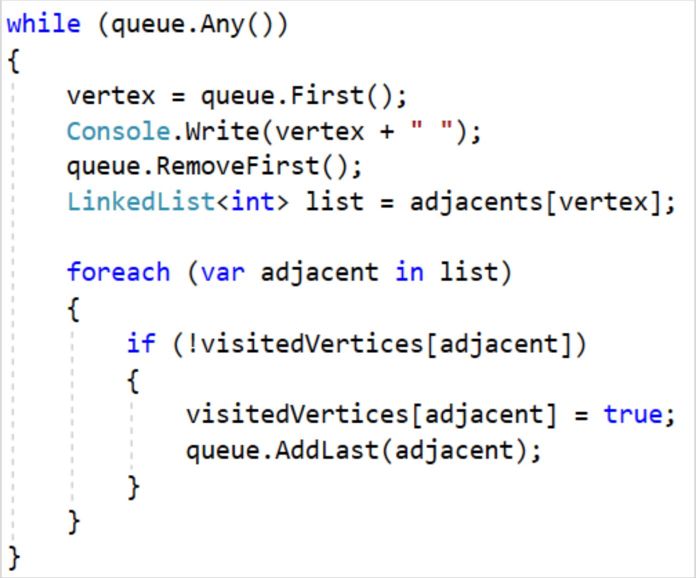
```
public void AddEdge(int firstVertex, int secondVertex)
{
    adjacents[firstVertex].AddLast(secondVertex);
}
```

Now, it's time to implement the **BFS algorithm** itself. Create a **method** and add a collection of **visited vertices** and mark all vertices as **not visited** (they are set as false by default in C#). Then, create a **queue** for the BFS algorithm, mark the **current vertex** as visited and **enqueue** it:



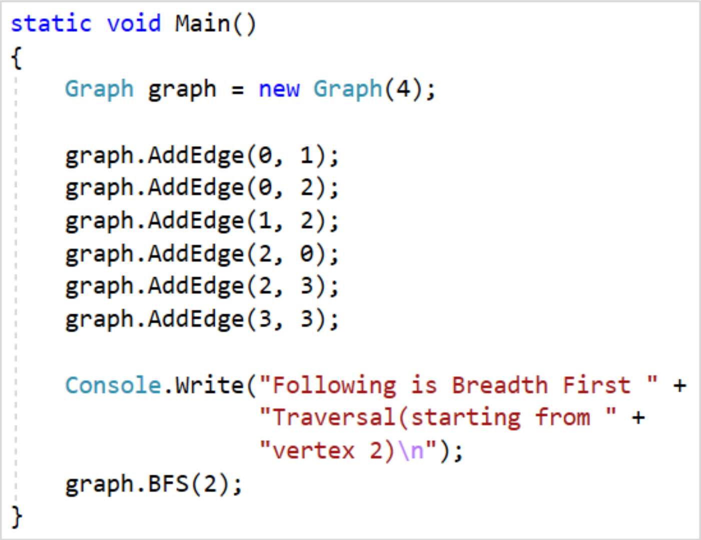
```
public void BFS(int vertex)
{
    bool[] visitedVertices = new bool[verticesCount];

    LinkedList<int> queue = new LinkedList<int>();

    visitedVertices[vertex] = true;
    queue.AddLast(vertex);
```

After that, until the queue is **empty**, you should get **vertices** from the queue with their **adjacents**. Print the current vertex. If an adjacent has not been visited, mark it as visited and add it to the queue:



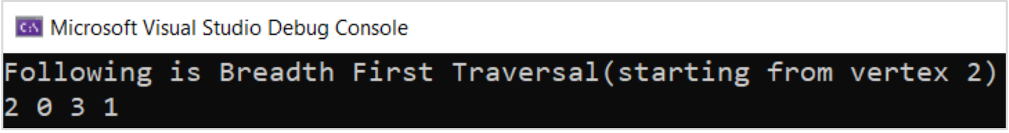
```
while (queue.Any())
{
    vertex = queue.First();
    Console.Write(vertex + " ");
    queue.RemoveFirst();
    LinkedList<int> list = adjacents[vertex];

    foreach (var adjacent in list)
    {
        if (!visitedVertices[adjacent])
        {
            visitedVertices[adjacent] = true;
            queue.AddLast(adjacent);
        }
    }
}
```

Finally, **use** the new methods from the **`Main()`**. Let's try traversing the graph from the above picture, starting from **`vertex 2`**:



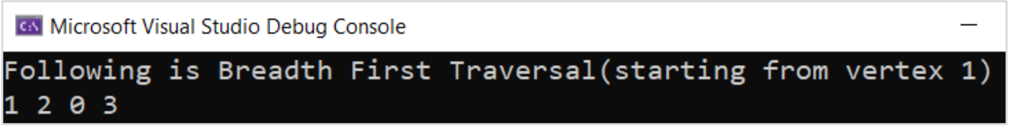
```
static void Main()
{
    Graph graph = new Graph(4);

    graph.AddEdge(0, 1);
    graph.AddEdge(0, 2);
    graph.AddEdge(1, 2);
    graph.AddEdge(2, 0);
    graph.AddEdge(2, 3);
    graph.AddEdge(3, 3);

    Console.Write("Following is Breadth First " +
                  "Traversal(starting from " +
                  "vertex 2)\n");
    graph.BFS(2);
}
```

The result should be the following:



```
Following is Breadth First Traversal(starting from vertex 2)
2 0 3 1
```

You can also try traversing the same graph, starting **`vertex 1`**. This is the result:

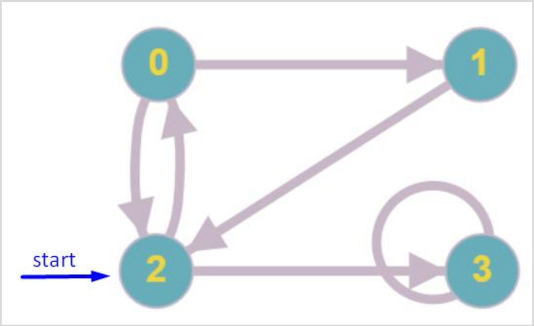


```
Following is Breadth First Traversal(starting from vertex 1)
1 2 0 3
```

2. Traverse Graph with DFS

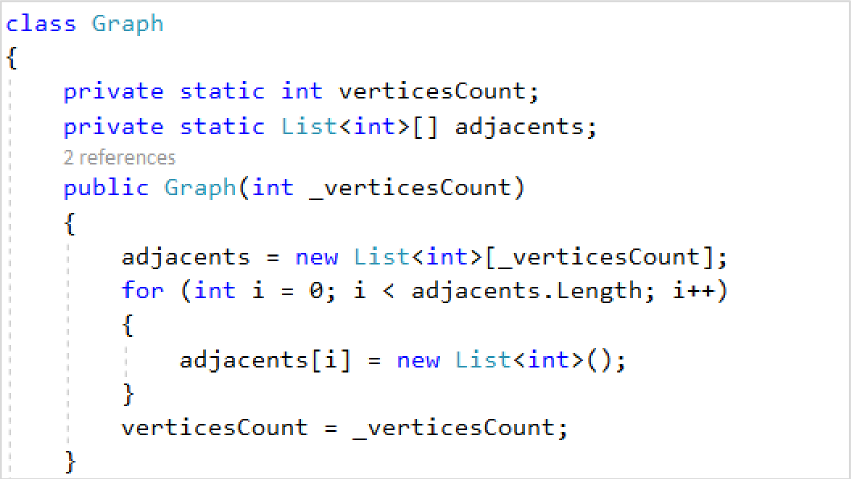
`Depth First Search` for a graph is similar to **DFS** for a tree. However, to avoid processing a node more than once, we need to use a boolean **`visited` array**, as we did in the **BFS** algorithm.

The algorithm starts at the **root** node and explores **as far as possible** along each branch before **backtracking**. So the basic idea is to start from the **root** or any **arbitrary** node and **mark** the node and move to the **adjacent unmarked** node and continue this loop until there is **no unmarked adjacent** node. Then **backtrack** and check for other unmarked nodes and **traverse** them. Finally **print** the nodes in the path. We will word with the same graph from the previous task:

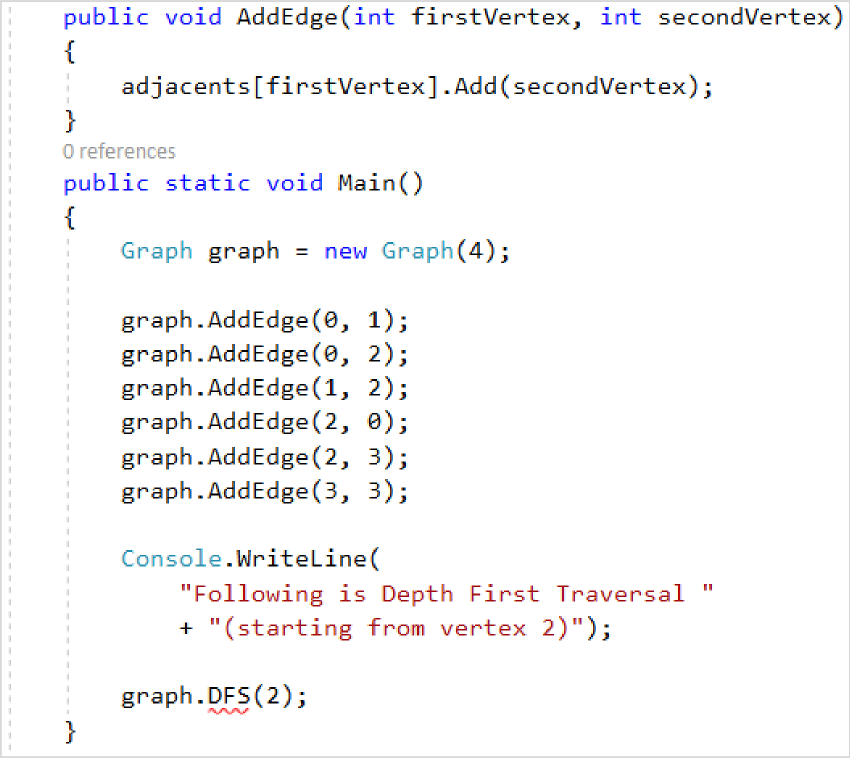


Let's implement the **DFS `algorithm`**, as well.

You can use the code from the previous task, as methods for **creating graph** are also valid here. However, you don't need **`LinkedList<T>`** already, since we do not work with a sequence, and it's a good idea to change it to **`List<T>`**. Your code should look like this:

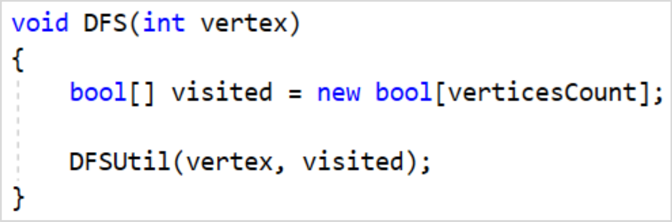


```
class Graph
{
    private static int verticesCount;
    private static List<int>[] adjacents;
    2 references
    public Graph(int _verticesCount)
    {
        adjacents = new List<int>[_verticesCount];
        for (int i = 0; i < adjacents.Length; i++)
        {
            adjacents[i] = new List<int>();
        }
        verticesCount = _verticesCount;
    }
```



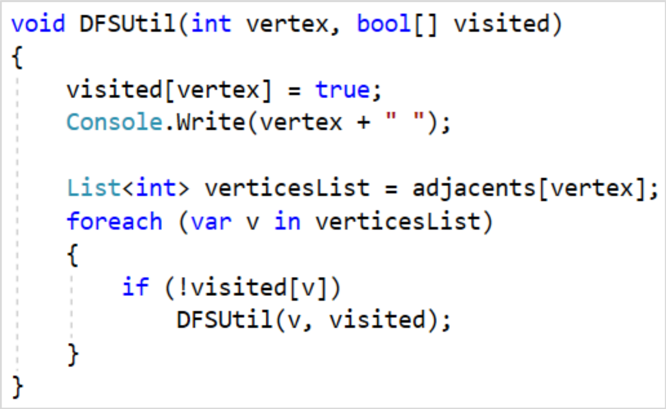
```
    public void AddEdge(int firstVertex, int secondVertex)
    {
        adjacents[firstVertex].Add(secondVertex);
    }
    0 references
    public static void Main()
    {
        Graph graph = new Graph(4);

        graph.AddEdge(0, 1);
        graph.AddEdge(0, 2);
        graph.AddEdge(1, 2);
        graph.AddEdge(2, 0);
        graph.AddEdge(2, 3);
        graph.AddEdge(3, 3);

        Console.WriteLine(
            "Following is Depth First Traversal "
            + "(starting from vertex 2)");

        graph.DFS(2);
    }
```

Now, let's implement the **DFS `algorithm`** itself. Start with the **`DFS()`** method, which should accept a vertex. We should also create a collection for **visited vertices** and invoke a **`DFSUtil()`** method, which we will implement next:



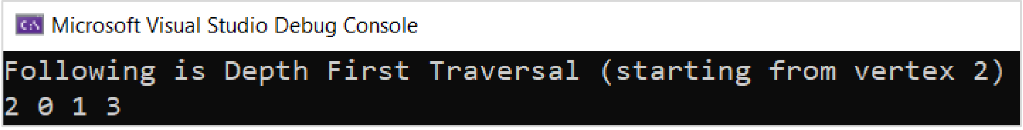
```
void DFS(int vertex)
{
    bool[] visited = new bool[verticesCount];

    DFSUtil(vertex, visited);
}
```

Finally, let's create the **`DFSUtil()`** helping method, which accepts **vertex** and a collection of **visited vertices**. It should **mark** the current vertex as **visited** and **print** it. Then, we should get all **adjacent vertices** to the current one and, if it is **not visited**, **recursively mark** it as visited and print it. Do it the following way:



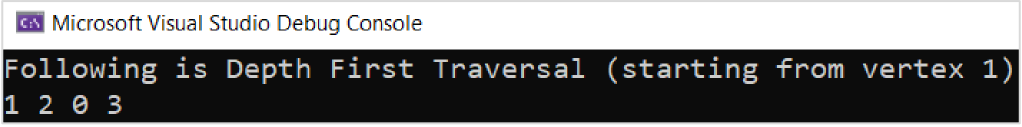
```
void DFSUtil(int vertex, bool[] visited)
{
    visited[vertex] = true;
    Console.Write(vertex + " ");

    List<int> verticesList = adjacents[vertex];
    foreach (var v in verticesList)
    {
        if (!visited[v])
            DFSUtil(v, visited);
    }
}
```

When ready, **test** the program. The result when **`vertex 2`** is the starting vertex should be the following:



```
Following is Depth First Traversal (starting from vertex 2)
2 0 1 3
```

Then, if **`vertex 1`** is the starting one, result should be this:



```
Following is Depth First Traversal (starting from vertex 1)
1 2 0 3
```

3. Read Graph

Use any of the above tasks' code. Your task is to create a **`ReadGraph()`** method to **read** the graph data from the console and create the **graph** and its **edges**. Test the method with both **BFS** and **DFS algorithms**.

Submit your solution in the **judge system**.

Input

Input consists of the following lines:

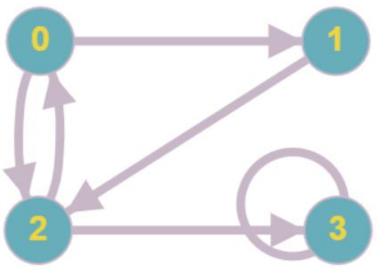
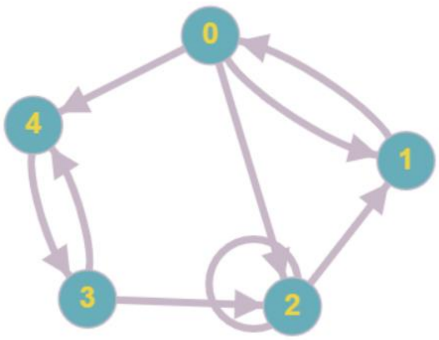
- On the first line, read **`vertices count`**
- On the second line, read the **`count of edges`** - **n**
- On the next **n** lines, read data for each edge in the following format: "**`{first vertex of edge} {second vertex of edge}`**"
- On the last line, read the **`starting vertex`** (the vertex, from which the traversal starts) – read it in the **`Main()`** method

Output

Output should consist of **two** lines:

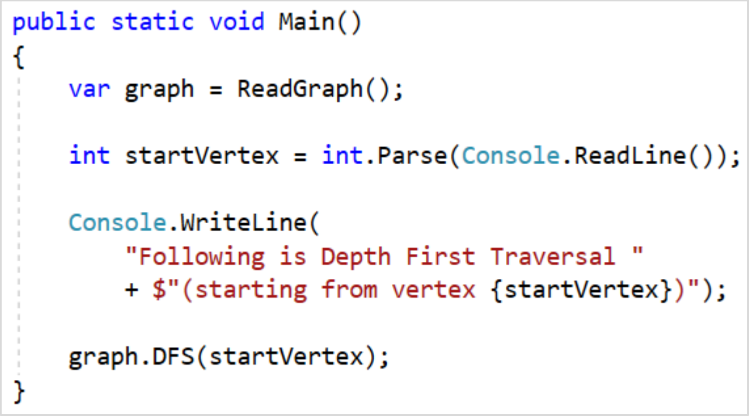
- On the first line, output should be in format: "**`Following is Depth/Breadth First Traversal (starting from vertex {start vertex})`**"
- On the second line- **`vertices`**, printed in **order of traversal**, separated by space (the **`DFSUtil()`** and the **`BFS()`** methods take care of this)

Examples

Input	Labyrinth	Output (DFS)	Output (BFS)
4 6 0 1 0 2 1 2 2 0 2 3 3 3 2		Following is Depth First Traversal (starting from vertex 2) 2 0 1 3	Following is Breadth First Traversal(starting from vertex 2) 2 0 3 1
5 9 0 1 0 2 0 4 1 0 2 1 2 2 3 2 3 4 4 3 3		Following is Depth First Traversal (starting from vertex 3) 3 2 1 0 4	Following is Breadth First Traversal(starting from vertex 3) 3 2 4 1 0

Hint

Invoke the **ReadGraph()** method from the **Main()**. At the end, your **Main()** method should look like this (for the DFS traversal):



```
public static void Main()
{
    var graph = ReadGraph();

    int startVertex = int.Parse(Console.ReadLine());

    Console.WriteLine(
        "Following is Depth First Traversal "
        + $"(starting from vertex {startVertex})");

    graph.DFS(startVertex);
}
```