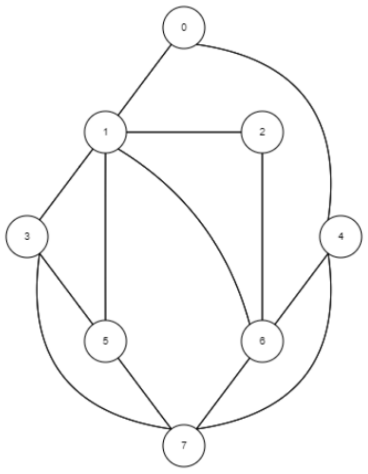
	Mühendislik Fakültesi Bilgisayar Mühendisliği Programı **BLM307E Algorithm Analysis & Design Lab Work**	Adı Soyadı: Notu:

Lab 4: Depth-First Search (Recursive and non-recursive implementation)

You will be given the adjacency list representation of a graph. The *Graph* class reads in a text file which contains the edges and creates the adjacency list representation of the associated graph.

Graph's logical representation



Adjacency List

0: 1 > 4
1: 2>3>5>6
2: 1>6
3: 5>7
4: 0>6>7
5: 1>3>7
6: 1>2>4>7
7: 3>4>5

Step1: DFS class takes an adjacency list representation of a graph to traverse through its edges:

```
Graph G= new Graph("edges2.txt", false);
DFS dfSearcher= new DFS(G.getAdjList());
```

Now, open DFS.java and you will see that a boolean array is used to keep track of the visited vertices. Implement following dfs method which traverses the graph using "**recursion**". No stack!

```
public void dfs(int i) {
        ...
}
```

Question 1: Write the adjacency list representation of this graph for selected 3 vertices (only write "1"s) (10p)

Question 2: Starting at vertex 0, traverse the graph by depth-first-search and draw the traversal stack. (15p)

Node	Stack in_order	Stack out_order
0	1	8
1	2	7
2	3	6
6	4	5
4	5	4
7	6	3
5	7	2
3	8	1

	Mühendislik Fakültesi Bilgisayar Mühendisliği Programı **BLM307E Algorithm Analysis & Design Lab Work**	Adı Soyadı: Notu:

Question 3: In the main method (in Test.java), call dfs method with a starting index of "0" (0 as keyboard input), write down graph traversal. You should paste your dfs method code below. (25p)

```
 void DFSUtil(int v,boolean visited[])
    {

        visited[v] = true;
        System.out.print(v+" ");

        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext())
        {
      int n = i.next();
            if (!visited[n])
                DFSUtil(n,visited);
        }
    }


    void DFS()
    {

        boolean visited[] = new boolean[V];


        for (int i=0; i<V; ++i)
            if (visited[i] == false)
                DFSUtil(i, visited);
    }
```

Question 3: Using DFS, can you find if a given directed graph is cyclic? If you can find it, explain it step by step (Pseudo-code). (25p)

a directed graph there may be more than one way to get from node X to node Y without there being any (directed) cycles in the graph
The number of backedges is not the number of cycles, because a single backedge can participate in more than one cycle.
In graph G1 of the depth-first search from A: it visits A->B->C, and then has a choice between D and E. Let's suppose it takes D. Then it visits E, and finds one backedge going to B. Thing is, the EB edge participates in both the BCE cycle and the BCDE cycle. if the node visited has been visited before, it becomes a cycle

Question 4: See the non-recursive version of the dfs method using the "stack" data structure. Which version of the DFS better, non-recursive version or stack version? Why? Explain it. Compare time complexities. (25p)

	Mühendislik Fakültesi Bilgisayar Mühendisliği Programı **BLM307E Algorithm Analysis & Design Lab Work**	Adı Soyadı: Notu:

```
public void dfs_no_recursion(int root) {
        Stack<Integer> s = new Stack<Integer>();
        s.push(root);
        int current_node;

        visited[root] = true;
        System.out.print(root + " ");

        int neighbor;
        while (!s.isEmpty()) {
                current_node = s.peek();
                neighbor = getUnvisitedNeighbor(current_node);
                if (neighbor != -1) { // root has a neighbor
                        visited[neighbor] = true;
                        System.out.print(neighbor + " ");
                        s.push(neighbor);
                } else
                        s.pop();
        } // end while
}
```

Repeats the followings until there are no elements in the stack:
At each iteration, looks at the last added element, and gets an unvisited neighbor of that element.

If there is such an element is found, marks that element as visited and pushes it onto the stack. This element then becomes the last added element in the stack.

If there is no such element, pops the last element from the stack.

```
private int getUnvisitedNeighbor(int root) {
        for (int neighbor : adjList[root]) {
                if (!visited[neighbor]) {
                        return neighbor;
                } // end if
        } // end for each
        return -1; // no unvisited neighbor has been found
}
```

Recursive

```
int n = adj.size();
  for (size_t i = 0; i < n; i++)
  {
    v[i].visited = false;
  }
  for (int w = 0; w < n; w++) {
    if (!v[w].visited) {
      explore(w);

    }
  }
```

Recursive dfs faster than non-recursive

In general, expect your own stack to be very slightly faster because you won't have to deal with the entire stack frame that is pushed whenever you make a function call / popped whenever said function returns. You will just push/pop the two variables you need. In java your own stack (from <stack>) is stored in a vector, so using it is cache-friendly. On the other hand, the recursive version is way more convenient, and the difference in runtime is rarely significant.

Both of complexity V+E