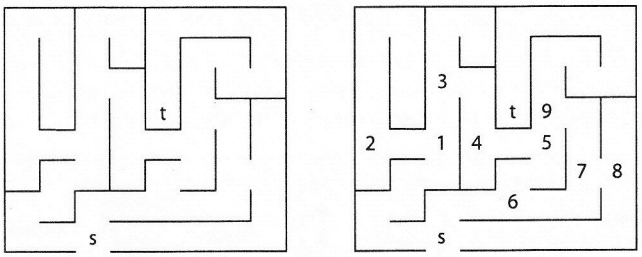
In this exercise we study graph algorithms.

Excercise 5a. (Implement maze resolving algorithm, 1p)

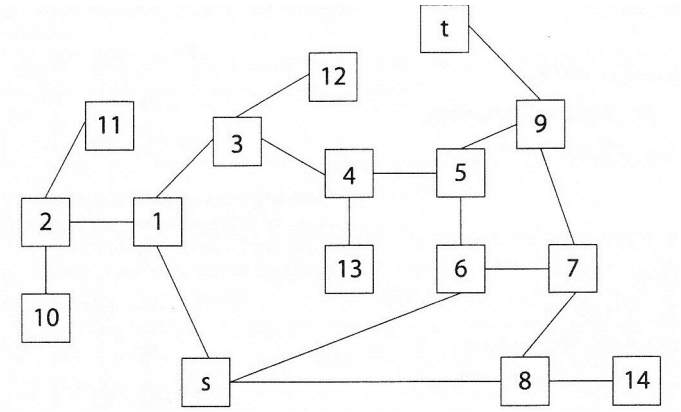
You have given a graph in a file maze.grh. In this file the first number on a line describes a node (an integer between 0 and the number of nodes – 1) and the following numbers define the list of adjacent nodes to the given node. E.g.

```
3 1 4 12
```

tells that the node number 3 has (one directional) connections to nodes 1, 4 and 12. The graph defined in this way describes the following maze (node s is 0 and t is 15 in the file).



In this maze, all the places where a selection (more than one possible path exists) can be made are numbered (as nodes) and those nodes are connected by edges if there exists a path between the given nodes. The graphical representation of the above maze is given on the below



Your task is to implement a program which resolves the given maze (or some other described by the input file). Maze can be solved by using depth-first graph search to find if there exists a path from the source node (node 0) to the target node (node 15). If that path exists, the maze can be solved. The solution path can be obtained by following the depth-first search path.

In order to make the maze solution reusable, we implement a new graph class, called as Graph. This class has the following methods

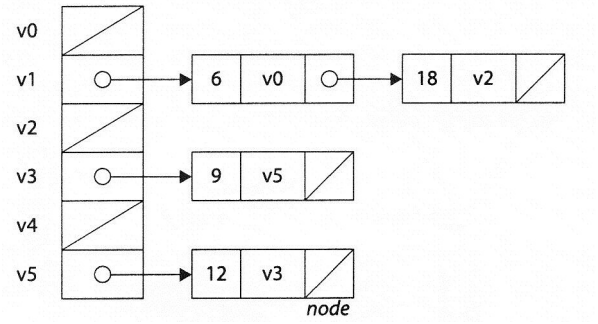
```
/* tell how many nodes a graph has */
public int nodes();

/* read a graph from the file */
public boolean readGraph(File file);

/* print a graph */
void printGraph();
```

```
/* Depth First Search
 * start is the node from where the search begins
 * visited is an array of all the visited nodes
 * pred is an describing the search path
 */
void dfs(int start, boolean visited[], int pred[]):
```

This Graph class represents the graph using adjacent linked list notation. This is an efficient notation (considering the memory usage) when the graph is sparse (as it often is). `adjList` field on the class is an array of pointers to adjacent vertices, e.g. `adjList[5]` gives the pointer to an another vertex who has an edge between them. If the pointer is NULL, that vertex does not have edges to other vertices at all. `data` field at the datatype tells the id (number) of the current vertex.



On the figure above, v0 to v5 are the vertices (`adjList` array). This array of vertices points to the `Node` type adjacent node linked list. In this Node there is also an additional field called weight which is not yet implemented in our Graph class.

`readGraph()` method creates space for the `Graph` data structure and initializes the content of the data structure using the contents of the file whose file descriptor is given as an argument. You can use `Scanner` class for reading one line from the file (method nextLine()). This line can be separated to an array of Strings with split() method of String class. There separate strings can then converted to integers using Integer.parseInt() method. `printGraph()` prints the content of the graph data structure as follows:

```
0: 8 6 1
1: 3 2 0
2: 11 10
3: 12 4 1
4: 13 3
5: 9 6 4
6: 7 5
7: 9 8
8: 14 7 0
9: 7 5 15
10: 2
11: 2
12: 3
13: 4
14: 8
15: 9
```

where the number before semicolon is the number of vertex and the number list following the semicolon is the adjacent vertices list which is terminated to the end of line. You can use this function to check that the `readGraph()` function works ok. `dfs(int start,`

`int visited[], int pred[])` does the depth-first type search process, starting from the start node, marking non-zero to all nodes visited to the `visited[]` array, and marking to the `pred[]` array which edge was leading to this vertex (pred[5]=3 means that vertex 3 has an edge to the vertex 5).

You can use the following test application to test that your ADT graph works.

```
/* find a maze solution */
private static int mazeSolution(int from, int to, int pred[], int steps[]) {
    int i, n, node;

    // first count how many edges between the end and the start
    node = to; n = 1;
    while ((node = pred[node]) != from) n++;

    // then reverse pred[] array to steps[] array
    node = to; i = n;
    while (i >= 0) {
        steps[i--] = node;
        node = pred[node];
    }

    // include also the end vertex
    return (n+1);
}

private final static String FILE = "H:/Metropolia/luennot/ADV_TIERA_Java/labs/lab05/dist/maze.grh";
private final static int    FROM = 0;
private final static int    TO   = 15;

/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
    Graph g = new Graph();

    // read the graph. and do the depth-first search
    System.out.println("Graph Adjacent list");
    g.readGraph(new File(FILE));
    g.printGraph();

    boolean visited[] = new boolean[g.nodes()];
    int     pred[]    = new int[g.nodes()];
    g.dfs(FROM, visited, pred);

    // then check if there is a solution by looking from the backwards to the start
    int steps[] = new int[g.nodes()];
    System.out.println("\nMaze solution from " + FROM + " to " + TO);;
    int n = mazeSolution(FROM, TO, pred, steps);
    for (int i = 0; i < n; i++)
        System.out.print(steps[i] + " ");
    System.out.println();
}
```