1 Maze Solving

For this assignment you will write a program for solving a maze. The program will receive as input a file with a description of the maze, and it will produce as output a path from the entrance to the exit, if any exists. A maze consists of a set of rooms connected by corridors, some of which are closed by doors. To open a door a number of coins between 0 and 9 are needed. Each door has associated a number that indicates how many coins are needed to open it. The program will be given a number of coins that it can use to open doors while trying to solve the maze.

Each coin can be used only once. Imagine that the doors have coin slots where the correct number of coins must be deposited for opening the door. Once the coins are placed inside the slot they cannot be reused.

Your program will store the maze as an undirected graph. Every node of the graph corresponds to a room, and every edge corresponds to either a corridor that can be used to go from one room to another or to a door that the program might decide to open. There are two special nodes in this graph corresponding to the entrance and the exit. A modified depth first search traversal, for example, can be used to find a solution for the maze.

2 Classes to Implement

You are to implement at least four Java classes: GraphNode, GraphEdge, Graph, and Maze. You can implement more classes if you need to, as long as you follow good program design and information-hiding principles.

For each one of the classes below, you can implement more private methods if you want to, but you cannot implement additional public methods.

2.1 Class GraphNode

This class represent a node of the graph. You must implement these public methods:

- `GraphNode(int name)`: the constructor for the class. Creates a node with the given name. The name of a node is an integer value between 0 and $n - 1$, where n is the number of nodes in the graph. A node can be marked with a value that is either `true` or `false` using the following method.
- void `mark(boolean mark)`: marks the node with the specified value.
- `boolean isMarked()`: returns the value with which the node has been marked.
- `int getName()`: returns the name of the node.

2.2 Class GraphEdge

This class represents an edge of the graph. You must implement these public methods:

- `GraphEdge(GraphNode u, GraphNode v, int type, String label)`: the constructor for the class. The first two parameters are the endpoints of the edge. The last parameters are the type and label of the edge. When representing a maze, the label of an edge could be `"corridor"` or `"door"`. When an edge represents a door, the type of the edge indicates the number of coins needed to open the door. For example, edge (u, v) with type 3 and label `"door"` represents a door of the maze between rooms u and v that requires 3 coins to open.
- `GraphNode firstEndpoint()`: returns the first endpoint of the edge.
- `GraphNode secondEndpoint()`: returns the second endpoint of the edge.
- `int getType()`: returns the type of the edge.
- void `setType(int newType)`: sets the type of the edge to the specified value.
- `String getLabel()`: returns the label of the edge.
- void `setLabel(String newLabel)`: sets the label of the edge to the specified value.

2.3 Class Graph

This class represents an undirected graph. You **must use** an adjacency matrix or an adjacency list representation for the graph. For this class, you must implement all and only the public methods specified in the `GraphADT` interface and the constructor. Therefore, this class must be declared as follows:

```
public class Graph implements GraphADT
```

The public methods in this class are described below.

- `Graph(n)`: creates an empty graph with `n` nodes and no edges. The names of the nodes are $0, 1, \ldots,$ `n` -1.
- void `insertEdge(GraphNode u, GraphNode v, int edgeType, String label)`: adds to the graph an edge connecting nodes `u` and `v`. The type and label for this new edge are as indicated by the last parameters. This method throws a `GraphException` if either node does not exist or if there is already an edge connecting the given nodes.
- `GraphNode getNode(int name)`: returns the node with the specified `name`. If no node with this name exists, the method should throw a `GraphException`.
- `Iterator incidentEdges(GraphNode u)`: returns a Java Iterator storing all the edges incident on node `u`. It returns null if `u` does not have any edges incident on it. If `u` is not a node of the graph a `GraphException` must be thrown.
- `GraphEdge getEdge(GraphNode u, GraphNode v)`: returns the edge connecting nodes `u` and `v`. This method throws a `GraphException` if there is no edge between `u` and `v` or if `u` or `v` are not nodes of the graph.
- `boolean areAdjacent(GraphNode u, GraphNode v)`: returns *true* if nodes `u` and `v` are adjacent; returns *false* otherwise. It throws a `GraphException` if `u` or `v` are not nodes of the graph.

2.4 Class Maze

This class represents the maze. As explained above, an object of the class `Graph` will be used to store the maze and to find a solution for it. You must implement the following public methods in this class:

- `Maze(String inputFile)`: constructor that reads the input file and builds the graph representing the maze. If the input file does not exist, or the format of the input file is incorrect this method should throw a `MazeException`. Read below to learn about the format of the input file.
- `Graph getGraph()`: returns a reference to the `Graph` object representing the maze. Throws a `MazeException` if the graph is null.
- `Iterator solve()`: returns a java `Iterator` containing the nodes of the path from the entrance to the exit of the maze, if such a path exists. If the path does not exist, this method returns the value `null`. For example for the maze described below the `Iterator` returned by this method should contain the nodes `0, 1, 5, 6,` and `10`.

3 Input File

The input file is a text file with the following format:

```
S
A
L
k
RHRHRH···RHR
VWVWVW···VWV
RHRHRH···RHR
VWVWVW···VWV
⋮
RHRHRH···RHR
```

Each one of the first 4 lines contain one number: `S, A, L,` or `k`.

- `S` is the scale factor used to display the maze on the screen. Your program will not use this value. If the maze appears too small on your monitor, you can increase this value. Similarly, if the maze is too large, choose a smaller value for the scale.
- `A` is the width of the maze. The rooms of the maze are arranged in a grid. The number of rooms in each row of this grid is the width of the maze.
- `L` is the length of the maze, or the number of rooms in each column of the grid.
- `k` is the number of coins that the program has available to open maze doors.

For the rest of the file, `R` is any of the following characters: 's', 'x', or 'o'. `H` could be 'w', 'c', or a digit '0', '1', ... '9'. `V` could be 'c', 'w', or a digit '0', '1', ... '9'. `W` must be 'w'. The meaning of the above characters is as follows:

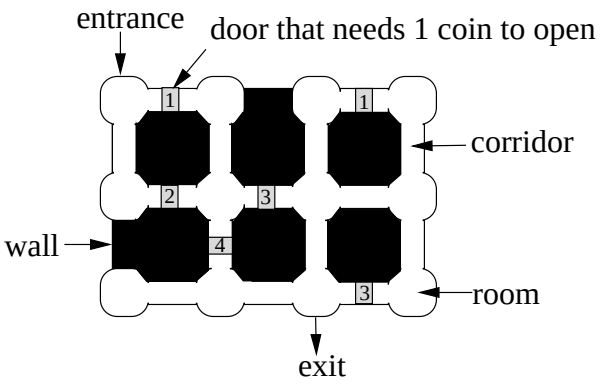
- 's': entrance to the maze
- 'x': exit of the maze
- 'o': room
- 'c': corridor

- 'w': wall
- '0', '1', ... '9': door that to be opened requires the number of coins specified by the digit; so '0' represents a door that does not need any coins to be opened, '1' represents a door that requires 1 coin, and so on.

There is only one entrance and one exit, and each line of the file (except the first four lines) must have the same length. Here is an example of an input file:

```
30
4
3
4
s1owo1o
cwcwcwc
o2o3oco
ww4wcwc
ococx3o
```

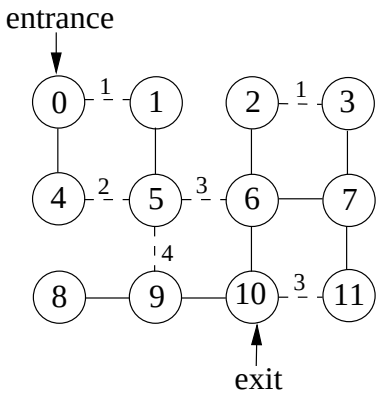
This input represents the following maze



In this maze four coins can be used to try to reach the exit.

4 Graph Construction

The rooms of the maze (or nodes of the graph) are numbered consecutively, starting at zero from the upper left room. For example, the above maze is represented with this graph:



where dotted edges represent doors and solid edges represent corridors. In the `Maze` class you need to keep a reference to the entrance and exit nodes.

5 Solving the Maze

A solution for the maze is **any** path from the entrance node to the exit node that uses at most the specified number of coins to open doors. If there are several solutions for the maze, your program can return any one of them.

The solution can be found, for example, by using a modified DFS traversal. While traversing the graph, your algorithm needs to keep track of the nodes along the path that the DFS traversal has followed. If the current path already has used all the coins to open doors, then no more edges representing doors can be added to it.

For example, consider the above graph and let the available number of coins be 4. Assume that the algorithm visits first nodes 0, 4, and 5. Note that to move from node 4 to node 5 the algorithm must 2 coins; hence, after reaching node 5 the algorithm has only 2 coins available.

As the algorithm traverses the graph, all visited nodes get marked. While at node 5, the algorithm cannot next visit nodes 6 or 9, since it does not have enough coins. Hence, the only place where the algorithm can go next is node 1. However, the exit cannot be reached from here, so the algorithm must go back to node 5, and then to nodes 4 and 0. Note that nodes 1, 5 and 4 must be unmarked when DFS traces its steps back, otherwise the algorithm will not be able to find a solution.

Next, the algorithm moves from vertex 0 to node 1 using one coin. From node 1 the algorithm moves to node 5 and then it uses the remaining 3 coins to go to node 6. At this point the algorithm has used all its coins, so no more doors could be open. Luckily, from node 6 the exit at node 10 can be reached without having to open any additional doors. So, the solution produced by the algorithm is: `0, 1, 5, 6,` and `10`. Note that the path 0, 1, 5, 9, 10 is not a valid solution as it requires the use of 5 coins.

You do not have to implement the above algorithm if you do not want to. Please feel free to design your own solution for the problem.

6 Code Provided

You can download from the course's website the following java classes: GraphException.java, MazeException.java, Board.java, GraphADT.java, DrawMaze.java and Solve.java. Class DrawMaze provides methods that are used to display the maze and the solution computed by your algorithm. Read carefully the code of class `Solve.java` to learn how to invoke the methods from the `Maze` class to find the solution for the maze. Class `Solve.java` shows how to use the iterator returned by the `solve()` method to draw the solution found by your algorithm on the screen. This class also contains the `main` method, so to execute the program you must type

```
java Solve inputFile
```

where `inputFile` is the name of the input file containing a description of the maze. You can use `Solve.java` to test your implementation of the `Maze.java` class.

You can also download from the course's website some required image files and examples of input files that you can use to test your program. You can also download `TestGraph.java` to test your implementation for the `Graph` class.

7 Hints

You might find the `ArrayList` and `Stack` classes useful. However, you do not have to use them if you do not want to. Recall that the java class `Iterator` is an interface, so you cannot create objects of type `Iterator`. The methods provided by this interface are `hasNext(), next()`, and `remove()`. An Iterator can be obtained from an `ArrayList` or `Stack` object by using the method `iterator()`. For example, if your algorithm stores the path from the entrance of the maze to the exit in a `Stack S`, then an iterator can be obtained from `S` by invoking `S.iterator()`.

You might find classes `StringTokenizer` and `BufferedReader`, and methods `Character.isDigit`, `Character.getNumericValue` and `Integer.parsetInt` useful.

8 Coding Style

Your mark will be based partly on your coding style.

- Variable and method names should be chosen to reflect their purpose in the program.
- Comments, indenting, and white spaces should be used to improve readability.
- No instance variable must be used unless they contain data which is to be maintained in the object from call to call. In other words, variables which are needed only inside methods, whose values do not have to be remembered until the next method call, should be declared inside those methods.
- All instance variables should be declared `private` to maximize information hiding. Any access to the variables should be done with accessor methods.

9 Marking

Your mark will be computed as follows.

- Program compiles, produces meaningful output: 2 marks.
- Tests for the `Graph` class: 4 marks.
- Tests for the `Labyrinth` class: 4 marks.
- Coding style: 2 marks.
- `Graph` implementation: 4 marks.
- `Labyrinth` implementation: 4 marks.