Data Structures and Algorithms **Spring-Summer 2012/13**

Assignment 7

**Valeria Fionda, Mouna Kacimi,
Werner Nutt, Simon Razniewski**

Graphs and Dynamic Programming

Instructions: Your assignment should represent your own effort. However, you are not expected to work alone. It is fine to discuss the exercises and try to find solutions together, but each student shall write down and submit his/her solutions separately. It is good academic standard to acknowledge collaborators, so if you worked together with other students, please list their names.
You must be prepared to present your solution at the lab. If you are not able to explain your solution, this will be considered as if you had not done your work at all.
You can write up your answers by hand (provided your handwriting is legible) or use a word processing system like Latex or Word. Experience shows that Word is in general difficult to use for this kind of task.
For a programming task, your solution must contain (i) an explanation of your solution to the problem, (ii) the Java code, in a form that we can run it, (iii) instructions how to run it. Also put the source code into your solution document. For all programming tasks, it is not allowed to use any external libraries ("import") or advanced built-in API functions (for example, `String.indexof("a")` or `String.substring(1,5)`), if not stated otherwise.
Please, include name, matriculation number and email address in your submission.

1. Least Dangerous Climbing Route

Imagine a climber trying to climb on top of a wall. A wall is constructed out of square blocks of equal size, each of which provides one handhold. Some handholds are more dangerous than others. From each block, the climber can reach three blocks of the row right above: one right on top, one to the right and one to the left (unless right or left are not available because the block is the end of the wall). The goal is to find the least dangerous path from the bottom of the wall to the top where the danger rating of a path is the maximum of the danger ratings of blocks used on that path.

We represent the problem as follows. The input is an $n \times m$ grid in which each cell has a rating $R(i, j)$ associated with it. The bottom row is row 1, the top row is row n. From a cell (i, j) a climber can reach in one step cell $(i + 1, j - 1)$ (if $j > 1$), cell $(i + 1, j)$, and cell $(i + 1, j + 1)$ (if $j < m$).
Here is an example of an input grid:

6	3	4	1	2	1
5	8	2	3	1	5
4	0	6	4	9	7
3	6	5	8	3	4
2	2	7	1	9	5
1	6	8	3	5	1
	1	2	3	4	5

Figure 1: Climbing Grid

For instance, on that 6×5 climbing grid, $R(1, 3) = 3$ and the climber can move to $(2, 2)$, $(2, 3)$, $(2, 4)$, each having a cost of 7, 1, 9, respectively. For a climbing path from the bottom to the top, the *danger rating* is the maximum of all values on the blocks that make up the path.

1. Write a recursive algorithm in Java that finds a path from bottom to top with minimal danger factor and prints it. Test the algorithm with the grid in Figure 1.

2. Write a dynamic programming version of the recursive "best path" algorithm from Task 1 in Java. Compare the running time of both algorithms.

Hint 1: First, solve the optimization problem. Then create an additional data structure that keeps track of the choices made when finding the optimum.
Hint 2: For the optimization problem, consider the more specific problem to find for a given cell in the top row the best path leading to that cell.

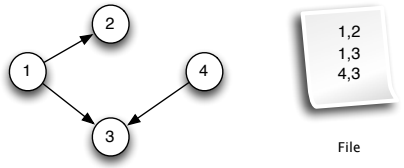
(10 Points)

2. Implementation of Directed Graphs

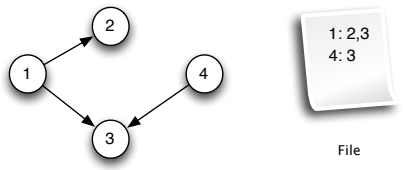
In this exercise you are asked to write Java classes that implement directed graphs (digraphs) according to the "adjacency list approach" from the lecture.

- A class `Node` having the following fields:
  - `int id`: the id of the node;
  - `char color`: that can be 'w', 'g' or 'b';
  - `Node pred`: the predecessor node (in a spanning tree);
  - `int distance`: the distance from the starting node;
  - `NodeList adj`: the nodes reachable from the current node traversing an edge;
- A class `Graph` having the following field:
  - `NodeList nodes`: representing the set of nodes of the graph;

  and at least the following methods:
  - `void addEdge (int i, int j)`: add an edge from the node having id `i` and the node having id `j`; if the nodes do not exist they are created;
  - `void readFromFile(String file)`: constract the graph from a file storing each edge on a different line; for example, the graph on the left is encoded by the file in the right.



  - `void printOnFile(String file)`: print the current graph on the specified file; the output format is shown in the following figure:



- A class `NodeList` that reimplements the class `List` from previous exercises to handle this new kind of `Node` objects. Implement at least the following method:
  - `findOrMake(int i)`: search in the `NodeList` for a `Node` whose `id` is `i` and return it; if such a node is not in the list, create a new `Node` having `id` = `i` add it the list, and return it.

(10 Points)

3. Graph Traversal

Add to your `Graph` class the following methods for graph traversal:

- `Graph BSF()`: implementing breadth-first search (BFS) over the graph and returning a new `Graph` representing the spanning tree of the BFS traversal;
- `Graph itDSF`: implementing depth-first search (DSF) algorithm over the graph by an algorithm that maintains an explicit stack; the method returns a new `Graph` representing the spanning tree and prints the topological sort of the nodes as a side effect;
- `Graph recDSF`: implementing the DSF with a recursive algorithm over the graph; the method returns a new `Graph` representing the spanning tree and prints the topological sort of the nodes as a side effect.

Hint: For a DSF that returns a topological sort, you may have to add some fields to the class `Node`.

(10 Points)

Submission: Until Wed, 12 June 2013, 11:59 pm, to

`dsa-submissions AT inf DOT unibz DOT it.`

Submit your work in two files, one PDF document and one .tar or .jar file with your code.