Dr. Hasmik Gharibyan
CSC/CPE349

Project 3: Dynamic Programming

Deadlines: submit your files **electronically** by **midnight** (end of day) on **Friday, 11/02/18.**

Late submission: you can submit your work within 24 hours after deadline (penalty applies):
- by 2 a.m. on Saturday, 11/03/18 for **25 point penalty**
- from 2:01 a.m. to midnight (end of day) on Saturday, 11/03/18 for **50 point penalty**

Programming Language: ***Java***.

PART 1 (50 points): Implement a class named *FactoryProblem*. //DON'T change the name

Write a program to implement a **dynamic programming** algorithm for solving the Assembly-line Scheduling Problem. A photocopy of pages with problem description and implementation details can be found *either* through a link posted on the course website right next to this assignment's link, *or* directly at: http://users.csc.calpoly.edu/~hghariby/AssemblyLineSchedulingProblem.pdf.

IMPORTANT: Do **NOT** start implementing your algorithm **until you read the last section** of the photocopied pages called *"Step 2: A Recursive Solution"* (on last two pages) – this section contains **essential** information and main formulas needed for the implementation.

You are required to use the **BOTTOM-UP method (i.e. iterative approach)** for your algorithm; you should NOT use recursion for finding/computing values of *f* and *l* functions. However, when constructing/printing the actual optimal solution, you may use recursion (in that case you will need a separate supporting method carrying the recursion).

Attention: you are NOT allowed to use naïve recursion or memoized top-down approach when computing the *value* of the optimal solution.

Your *FactoryProblem* class should have **(i)** all pieces of code (all necessary methods) for the BOTTOM-UP implementation of your dynamic programming algorithm for the solution of the Assembly-line Scheduling Problem (*you can call the methods whatever you want*), and **(ii)** a ***main*** method that reads input data from a text file, executes the algorithm on inputted data, and outputs the computed optimal solution on the screen.

In the *main* method you need to do the following:

1. Prompt the user to enter the name of the file containing the data for your problem, **input the file-name** and set up a scanner to read from that file.
2. Read values from the input-file and save in appropriate variables and arrays. These are the data you will give to your algorithm as parameter-values.
   The input-file content will be as follows (values on the same line are separated by ≥1 **spaces**):
   - the first line contains one integer: this is the value of $\boldsymbol{n}$
   - the second line contains two integers: these are the values of $\mathbf{e_1}$ and $\mathbf{e_2}$
   - the third line contains two integers: these are the values of $\mathbf{x_1}$ and $\mathbf{x_2}$
   - the fourth line contains *n* integers: these are the values of $\mathbf{a_{1,1}, a_{1,2}, \ldots, a_{1,n}}$
   - the fifth line contains *n* integers: these are the values of $\mathbf{a_{2,1}, a_{2,2}, \ldots, a_{2,n}}$
   - the sixth line contains *n-1* integers: these are the values of $\mathbf{t_{1,1}, t_{1,2}, \ldots, t_{1,n-1}}$
   - the seventh line contains *n*-1 integers: these are the values of $\mathbf{t_{2,1}, t_{2,2}, \ldots, t_{2,n-1}}$

   Note: 1. Variable names are as defined in problem description (**see posted pages** on website).
   2. Assume the file-content is correct (do **not** worry about validity or format checking).

Example: for the data used in Figure 15.2 (**see posted pages**), your file content will be this:

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

3. Invoke your algorithm for solving the problem and get the results. Output the fastest time to get the chassis from starting point to exit (*the first line in the example output below*). Then invoke your method for printing the optimal solution: it should print the route corresponding to an optimal solution (*the remaining lines in the example output below*).

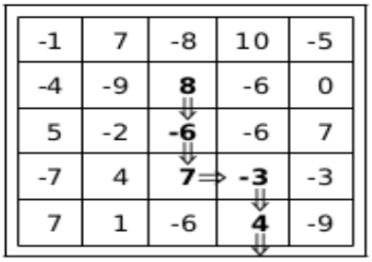
   Example: for the example in Figure 15.2, your output on the screen will look like this:

Fastest time is: 38

The optimal route is:
station 1, line 1
station 2, line 2
station 3, line 1
station 4, line 2
station 5, line 2
station 6, line 1

PART 2 (50 points): Implement a class named *GameProblem*. //DON'T change the name

Problem description ***(adapted from Theresa Migler; original is by Glencora Borradaile)***
You will design a dynamic programming algorithm for the following game that is played on an ***n*×*m*** grid **A** (it is not necessarily square). You start on any square of the grid. Then on each turn, you move either one square to the right or one square down. The game ends when you move off the edge of the board (either the bottom side or the right side). Each square of the grid has a numerical value. Your score is given by the sum of the values of the grid squares that you visit. The object is to maximize your score. For example, in the grid below, the player can score 8 - 6 + 7 - 3 + 4 = 10 by following the highlighted route. (This is not the best solution. The best solution has value 15 – can you find it?)



Write a program to implement **dynamic programming** algorithm for solving the above described problem. You are required to use the **BOTTOM-UP method (i.e. iterative approach)** for your algorithm -- you should NOT use recursion when computing/finding the optimal *value*. However, when constructing/printing the actual optimal solution, you may use recursion (in that case you will need a separate supporting method carrying the recursion).

Attention: you are **NOT** allowed to use naïve recursion or memoized top-down approach when computing the *value* of the optimal solution.

Your *GameProblem* class should have **two *public* methods**: *main* and *game* – both *void* methods. You may have private methods that support public methods (you can define them as appropriate).

- ***main*** **method:** in this method you need to do the following.

1. Prompt the user to enter the name of the file containing the data for your problem, **input the file-name** and set up a scanner to read from that file.
2. Read values from the file: these are the data you will pass to *game* method as parameter-values. Your file content will be as follows (values on the same line will be separated by ≥1 **spaces**):
   - the first line contains two integers: values of ***n*** and ***m*** (i.e. number of rows and columns of A)
   - then *n* lines will follow, each containing *m* integers (**a line** represents **one row of A**)

   Note: Assume the file has a correct content (*don't worry about checking validity or format*).

   Example: for the data used in the above picture, your input file content will be this:

```
 5  5
-1  7 -8  10 -5
-4 -9  8  -6  0
 5 -2 -6  -6  7
-7  4  7  -3 -3
 7  1 -6   4 -9
```

3. Invoke your *game* method passing values for *n*, *m* and *A* parameters. The *game* method will do all the work (it will output the max score and the optimal route).

- ***game*** **method:** this method should have the following signature.

```
public static void game(int n, int m, int[][] A)
        //The parameters represent:  number of rows, number of columns, and the A grid
```

Your *game* method should implement a ***dynamic programming*** **algorithm** (***bottom-up*** **method)** to find and output (i) the **maximum score** that can be obtained for an instance of the problem, and (ii) **a route** that brings that maximum score.

Details of the implementation: You are going to define a table S[1..n,1..m] to hold max scores; value of S[i, j] will be the max score for a path starting at [i,j] square. Your goal is to find a square (i.e. a **pair of index-values *x* and *y***) that has the max score and be able to track the route that starts at [*x*,*y*] square. So, for the *x*,*y* index-values S[*x*,*y*] =$\max_{1\le i\le n,1\le j\le m}\{S[i,j]\}$.

First, let's see how to compute the S-values: here is a definition that will help you:

$$S[i,j] = \begin{cases} A[n,m] & if\ i=n, j=m\ (this\ is\ the\ bottom-right\ square: you\ can\ only\ exit) \\ \max\{S[i+1,m],0\}+A[i,m] & if\ j=m\ (this\ is\ the\ last\ column: move\ down\ or\ exit) \\ \max\{S[n,j+1],0\}+A[n,j] & if\ i=n\ (this\ is\ the\ last\ row: move\ right\ or\ exit) \\ \max\{S[i+1,j],S[i,j+1]\}+A[i,j] & otherwise\ (check\ the\ better: move\ down\ or\ right) \end{cases}$$

Attention: in what order you will fill the S-values in the table is important: think about it.

To be able to track the rout, you need to define an auxiliary table R: in each R[i,j] cell you will save info about the choice made in computing the value of respective S[i,j] – e.g. you can save "d" if the choice is to move down, "r" if the choice is to move right, and "e" if you have to exit.

Note: When you find an *x*,*y* pair of indexes that provide the max score (max S-value), you can then create your output of the optimal route: start with the [*x*,*y*] cell in the R-table and follow the "directions" registered in the R-table – move right, down, or exit (this is similar to what we did in class when producing the output of LCS problem's optimal solution).

ATTENTION: in your output the indexing should be natural (**indexing must start with 1**).

Example: for the above example, your output on the screen will look like this:

Best score: 15
Best route: [3,1] to [3,2] to [4,2] to [4,3] to [4,4] to [5,4] to exit

Submitting your work:

Turn in the *FactoryProblem* and *GameProblem* source files electronically via "*handin*" procedure by the deadline (see the top of first page of this document for the due date and time):

The account *id* is: ***hg-cpe103***
The *name* of the assignment is: ***Project3***
The *name* of the assignment for late submissions is: ***Project3_late.***

Important requirements:

1. Your programs must **compile and run from command line** on our departmental servers. So, before submitting, make sure your programs compile and run on lab machines.
2. Do not create packages for your source code, do not zip, and do not use folders for your submission – all these complicate the automated grading process.
3. Each file must have a **comment-header** which should include both authors' **names** and **ids** (as in *id@calpoly.edu*), as well as the **date** and the **assignment name** (e.g. Project 3). **Reminder**: only **one** submission needs to be made for a pair/team.
4. The header of each file must **start at the first line** (no empty lines or *import* statements before it) and should be followed by at least one empty line before the first line of the code.