Dr. Hasmik Gharibyan
CSC/CPE349

Project 3: Dynamic Programming

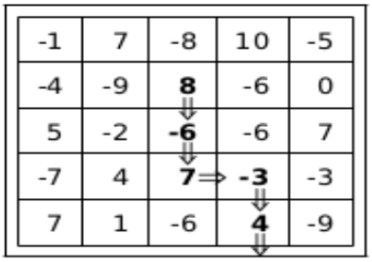
Deadlines: submit your files **electronically** by **midnight** (end of day) on **Wednesday, 05/05/21.**

Late submission: you can submit your work within 24 hours after deadline (penalty applies):
-by 2 a.m. on 05/06/21 for **10-point penalty**,
-from 2:01 a.m. till *noon* on 05/06/21 for **25-point penalty**,
-from 12:01 p.m. till *midnight* (end of the day) on 05/06/21 for **50-point penalty**

Programming Language: ***Java***.

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** (the grid 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 a **dynamic programming** algorithm for solving the above described problem (see details of implementation on the next page, in game method's description). You are required to use the **BOTTOM-UP method (i.e. iterative approach)** for your algorithm.

Attention: you are **NOT** allowed to use naïve recursion or memoized top-down approach. There should **NOT** be any recursion in your program – not in computing/finding the optimal value, and not in constructing/printing of the actual optimal solution.

The *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 line 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 the below-described *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 leads to 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 each S[i, j] will be the max score for a path starting at [i,j] square. Your goal is to find a square in S (i.e. a **pair of index-values *x* and *y***) that has the max score, and then 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:

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

Attention: 1. the order in which you fill the values in the S-table is important: **think about it**.
2. be mindful of the efficiency of the algorithm – do NOT include unnecessary checks that will slow down the algorithm (particularly, inside any of *i*- or *j*- loops do **NOT** check if *i*==*n* or *j*==*m*).
Hint: do **NOT** implement all 4 lines of the formula in one nested loop. Implement each line of the formula separately/individually (in an ***independent*** segment).

To be able to track the rout, you need to define an auxiliary table R which is fill parallel to S: 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 represent the max score (i.e. the max S-value), you can then create the 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, on the screen your output should 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 *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 program 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. Your source 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 the source 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.