CSCI 221: Computer Programming II

Programming Assignment 5

Due Tuesday March 29, 11:59pm

Leniency 24 hours later

Collaboration Policy: Work alone. Write your own code. You may not use any code found on the internet or anywhere else in this assignment. You may not use any code provided by anyone other than the instructor, including tutors in CSL. You may not share code with others.

Programming Assignment: Mountain Trek

In this assignment you will read a set of topographic (land elevation) data into a 2D array and write some methods to compute some paths through the mountains. The code to visualize the paths is provided

Background:

There are many contexts in which you want to know the most efficient way to travel over land. When traveling through mountains (let's say you're walking) perhaps you want to take the route that requires the least total change in elevation with each step you take – call it the path of least resistance. Given some topographic data it should be possible to calculate a "greedy lowest-elevation-change walk" from one side of a map to the other.
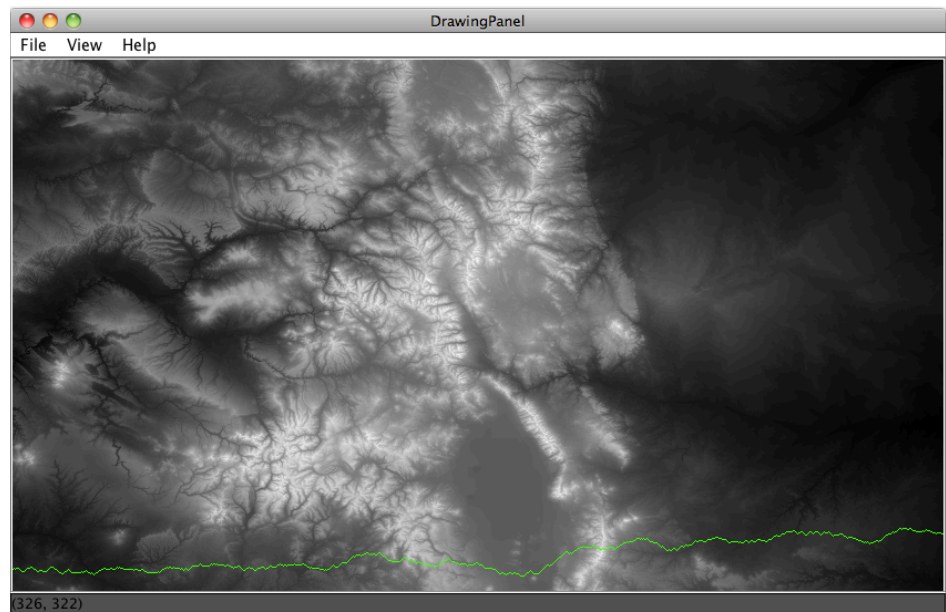
In the map above, brighter shades mean higher elevation. The green line shows a lowest-elevation-change route for this map, traveling west-to-east.

A Greedy Walk

A "greedy" algorithm is one in which, in the face of too many possible choices, you make a choice that seems best at that moment. For the maps we are dealing with there are 7.24×10^{405} possible paths you could take starting from western side of the map and taking one step forward until you reach the eastern side. (It would be computationally, too expensive, to compute all possible paths and then pick the best.) This greedy approach may not find **the** best path, but it is relatively efficient to compute.

| 3011	2900	2852	2808	2791	2818
2972	2937	2886	2860	2830	2748
2937	2959	2913	2864	2791	2742
2999	2888	2986	2910	2821	2754
2909	2816	2893	2997	2962	2798

Shows a portion of the data. Greedy path shown in green.

Since our map is in a 2D grid, we can envision a "walk" as starting in some in some cell at the left-most edge of the map (column 0) and proceeding forward by taking a "step" into one of the 3 adjacent cells in the next column over (column 1). Our "greedy walk" will assume that you will choose the cell whose elevation is closest to the elevation of the cell you're standing in. (NOTE: this might mean walking uphill or downhill).
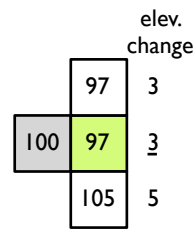
The diagrams below show a few scenarios for choosing where to take the next step. In the case of a tie with the forward position, you should <u>always</u> choose to go straight forward. In the case of a tie between the two non-forward locations, you should flip a coin to choose where to go.
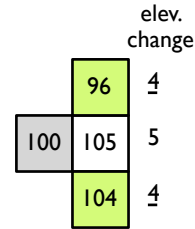
elev.
change

109	9
100	107	7
105	5
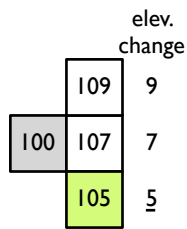
Case 1: smallest change is 5, go fwd-down

elev.
change

109	9
100	97	3
105	5
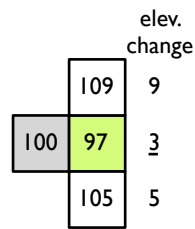
Case 2: smallest change is 3, go fwd

elev.
change

97	3
100	97	3
105	5

Case 3: smallest change is a tie (3), fwd is an option, so go fwd

elev.
change

96	4
100	105	5
104	4

Case 4: smallest change is a tie (4), choose randomly between fwd-up or fwd-down

There are other ways to choose a path through the mountains that can be explored in the "above and beyond" section of this assignment.

Assignment Requirements
The minimum requirements for the assignment are that you write code to produce something like the map shown in the picture above. To do that you need to complete the following 5 steps:

Step 1: Read the data into a 2D array (10 points)
Step 2: Find min, max elevations, and other calculations on the data (25 points)
Step 3: Draw the map (code provided)
Step 4: Draw a lowest-elevation-change route starting from some row. (code provided) (35 points)
Step 5: Find and draw (code from Step 4) the lowest-elevation-change route in the map (10 points)
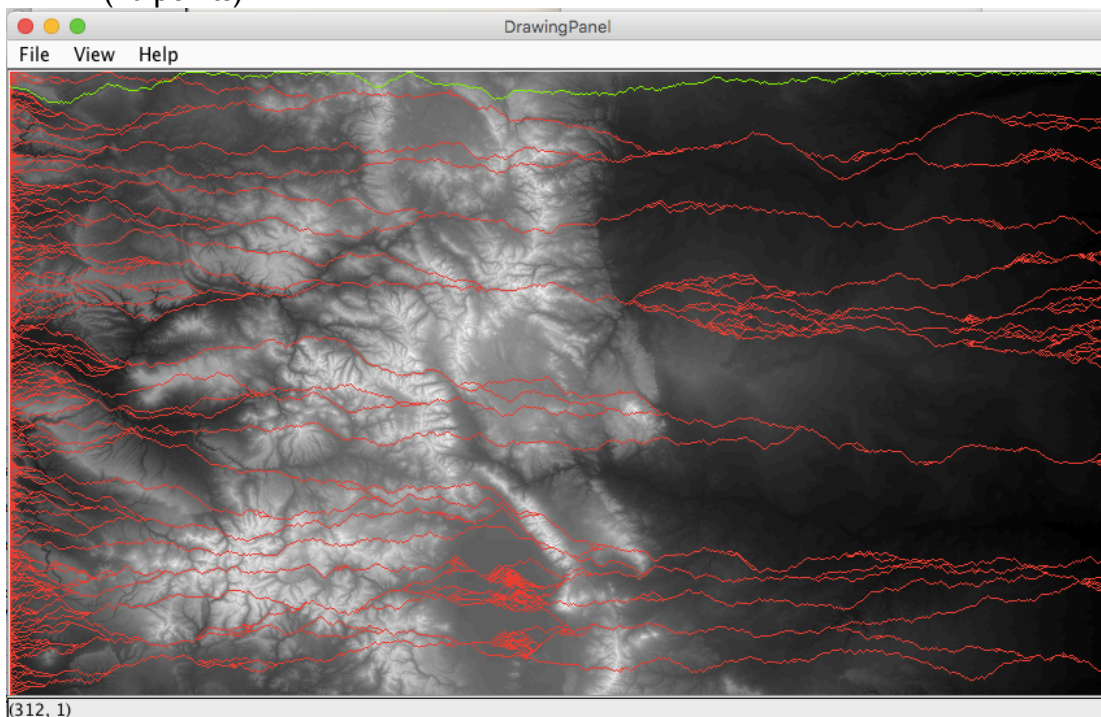
To fulfill the requirements you are provided with a class and method stubs for completing parts of the project. Here is the sketch:

```
class MapDataDrawer
  //a class for maintaining a 2D array of ints
  // representing a topographic map

     MapDataDrawer(String filename, int rows, int cols)
        //read data from given file into a 2D array

     int findMin()
        //return the minimum value in the map

     int findMax()
        //return the max value in the map

     void drawMap(Graphics g) // complete method provided do not edit
        //draw this map in B&W using given graphics context

     int indexOfMinRow(int col)
        //given a column, find the index of the row with min elevation
        // return row number of

     int drawLowestElevPath(Graphics g, int row)
        //draw the lowest elevation path starting from the given row
        // return total elev change from the path

     int indexOfLowestElevPath(Graphics g)
        //find the lowest elev change path in the map
        //return the row it starts on
```

Step 1 – Read the data into a 2D array

Your first goal is to read the values into a private 2D array of ints in the MapDataDrawer class. The constructor of the class should read the given data file into a new 2D array of ints maintained as a private variable of the class.

The data

The data is a plain ascii text file from NOAA representing the average elevations of patches of land (roughly 700x700 meters) in the US. [1] The data file included with the project, <u>colorado.dat</u>, is the elevation data for most of the state of Colorado (mountains!). The data comes as one large, space-separated list of integers. There are 403,200 integers representing a 480-row by 844-col grid. Each integer is the average elevation in meters of each cell in the grid.

In the constructor:
- Instantiate a new 2D array of ints of the appropriate size
- Read the ints out of the data file (the data are listed in row-major order).
- WARNING: The data are given as a continuous stream of numbers - there are no line breaks in the file, so you can't use nextLine(). You need to use Scanner's nextInt() method to read each subsequent in. You also probably want to write nested loops to cover the exact rows/cols (480/844) needed.

---

[1] <u>http://maps.ngdc.noaa.gov/viewers/wcs-client/</u>

After you've written the constructor, **test it on the object bench.** Make sure that you're actually reading data into the 2D array. Do a sanity check by looking at a specific row and col and comparing with a friend, or against the original data itself.

Step 2 – Find the min and max values
In order to draw the map you need to find the min and max values in the map first. Implement the **findMin()** and **findMax()** methods. These should return the smallest and largest values respectively in the 2D array. You'll need to write code that scans the entire array and keeps track of the smallest and largest things found so far.
Test these functions independently first to make sure you're getting decent values. Maybe check with a friend to see if s/he is getting the same thing.

Step 3 – Draw the map – done for you

Step 4 – Draw a lowest-elevation-change path from some row
Implement the **`drawLowestElevPath`**(Graphics, startRow) method. Note that this method does two things: 1) it draws the path, 2) it calculates and returns to the total elevation change on that path.
The method already includes the code to draw the West-to-East path, starting from the given row using the greedy lowest-elevation-change technique described in earlier pages.
You will need to do the following things in some order.
1. Starting from the given row, and column 0, color it green (or whatever color you want to draw your path). The call: **g.fillRect(0,row,**1, 1**)**; colors column 0, row "row" whatever color has been set (in the calling/driver) routine.

2. Write a loop that generates every possible column across the map from 1 to 840 (the number of columns) (you start at column 0, but column 1 is the first column you need to make choice about where to step). For each column you will decide which row to take your next "step" to – fwd, fwd-and-up, or fwd-and-down - using the greedy choice strategy described.

3. Color the chosen location to step green (or whatever color you choose). **g.fillRect(columnVar,rowVar,** 1**,** 1**)**; will color the new location, assuming that columnVar is the column index and rowVar is the row index in the grid.

4. Use a variable that keeps track of the 'current row' you're on, and update it each time you take a step forward – row may stay the same, or go up or down by 1 depending how you walk.

5. Use Math.abs(…) to get the absolute value of the difference between two elevations.

6. Continue finding the lowest neighboring cell and coloring it as you go.

7. Keep a running total of the total elevation change that would be 'experienced' by a person walking this path. Since we consider an elevation change the absolute value (i.e. going 'uphill' 10 meters is the same amount of change as going 'downhill' 10 meters) this running total will be non-decreasing and will end up being a pretty large positive number

   When you're done you should see a line tracing the lowest elevation-change path from west to east for the given row.

Step 5 – find the lowest elevation path for the whole map

Implement the **indexOfLowestElevPath**() method which finds the overall lowest-elevation-change path and returns the row it starts on.

You will find the lowest-elevation-change route by noting that the method you wrote in step 4 returns the total elevation change for a path starting at a given row.   If you write a loop that generates every possible starting row from 0 to 480, and call your method using each possible starting row, you should find the lowest route for the whole map. (Since that method also draws all of the routes it's good feedback that the method is running properly.) You might end up with something that looks like the figure on page 7.

KLUGE NOTE: Even though the indexOfLowestPath method accepts a Graphics object, this method should not use it to draw anything, but rather simply pass it onto drawLowestElevPath method to that it can use it to draw. (This is possibly a flaw in the program design, but a flaw designed to protect you from some complicated Object coupling that we could avoid.)
If you want to, you can clear the map after this calculation and simply draw the one line. The result should look *something* close to the image above or the one shown on the first page of the assignment.
Sanity Check:
NOTE: The greedy walk should always produce a *slightly* different path due to the fact that you are flipping a coin in some cases to determine which direction to go.

OPTIONAL: Above and Beyond (for the true mountain climbers)
### Path Finding
Note that our greedy lowest-elevation-change algorithm is flawed. It is not guaranteed to find the absolute lowest elevation change route from west to east since our decisions are limited to what's in front of us. In CSCI 230 and 310 and Math 207/307 you are likely to learn of more accurate algorithms. Some include

- Another, only slightly more complicated way to do a greedy walk is to start at an arbitrary x,y location in the map and do a greedy walk to the east from that point and a greedy walk to the west from that point. Using this method you could calculate the lowest elevation-change route that passes through every possible x,y coordinate. (Note, this will take some time to calculate…might not want to draw it live).

- A different kind of path you might want to follow in the mountains is to travel the path that stays as low as possible, elevation-wise, regardless of the change. Think of this as a greedy algorithm that always prefers to go downhill if possible, and uphill as little as possible. You can use a greedy method for this by always choosing to step to the location with the lowest elevation, not necessarily the lowest change in elevation. Show a comparison of these two paths.

- Write a method that finds, and highlights, the lowest elevation point for each possible column in the map. Compare that to the lowest elevation path you calculated for the problem and see If any of your paths pass through that point. If you do a greedy walk going east and west, from each of those points, do you end up finding a better elevation-change route?

- Do your walk considering more than the three forward locations. You could consider the 5, or even 7 surrounding locations. This can get pretty tricky: to do this you need

to keep track of which direction you are heading, or where your last step was, so that you don't back track.

- Find the best possible west-to-east walk using a version of the [Floyd–Warshall Shortest Path algorithm](). You may discuss how to do this with your instructor but you should see what you can figure out from the Wikipedia page.
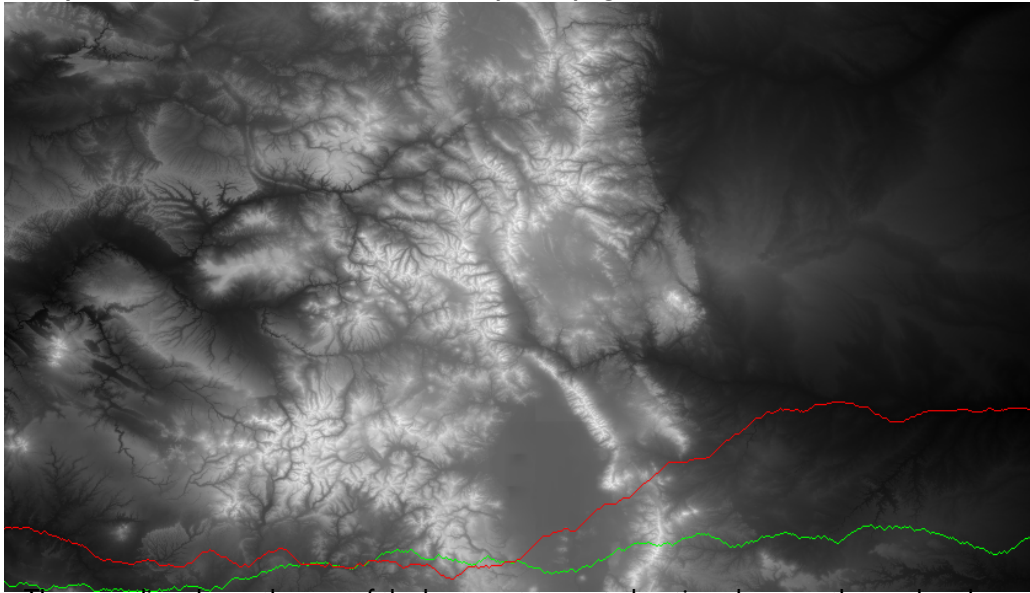
In

The green line shows the one of the best west-to-east elevation-change paths produced by a greedy walk. The red line is actually the best possible path. It was found using a modified version of the Floyd-Warshall algorithm

essence, iIt is possible to compute <u>all</u> of the possible paths from west-to-east, using a separate 2D array to keep track of the best cumulative elevation change possible for each cell in the grid. You construct this grid column-by-column "moving" from west-to-east and choosing the best of three possible values to put into each cell - since each cell can be arrived at from up to three different locations in the previous column, you need to choose which of the three elevation-changes affects the cumulative total the least, and put that value in the cell.

Since that grid of best values does not tell you the path you need to follow to realize those small elevation changes, you need to maintain <u>another</u> 2D array in parallel, in which you store the row index that should be traveled through to achieve the best path. The best path can then be reconstructed by walking backwards and following the row values.

The best possible path gives a total elevation change of 7,056 meters (fully half of the best greedy walk) and should start around row 428.

- Can you think of a different algorithm for choosing a "good" path through the mountains? It depends on your definition of 'good' of course, but if you can think of an interesting idea, go for it (just run it by Dr. McCauley first).

Submission check:
- Make sure MapDataDrawer is adequately documented. Plus add your name to Driver.
- Store the **3** required java files **and** the data file in a folder named <yourlastname>HW5 and zip it up. Submit as usual on OAKS.

NOTE: if your code does not compile without interaction from us, your grade will be a 20 at most, assuming your documentation is up to standards and you submit properly. If you submit improperly, your grade will be 0 – no exceptions.