

## CSCI165 Computer Science II

### Lab Assignment: Bit Mapped Path Finder

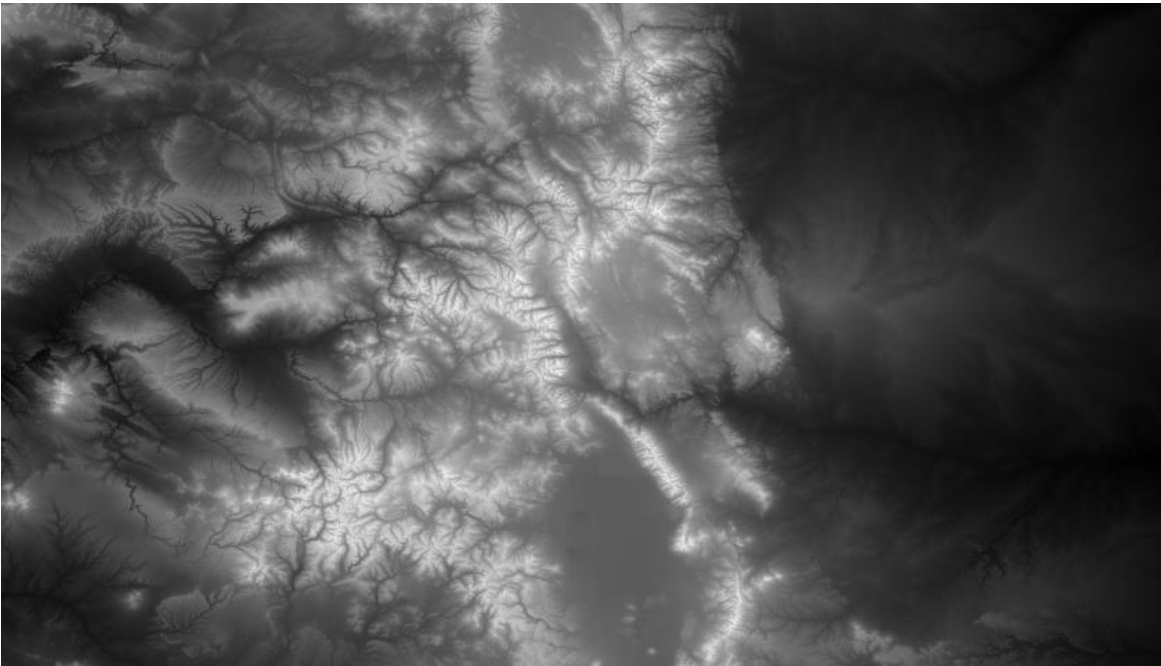
#### Week Six

This lab is an extension of the discussion problem from last week wherein you drew the map based on the topographic data.

#### Objectives

- Understand, implement and process Java 2D arrays
- Understand and implement simple Java 2D graphics
- Nested loops
- Understand and implement a greedy algorithm:  
[https://en.wikipedia.org/wiki/Greedy\\_algorithm](https://en.wikipedia.org/wiki/Greedy_algorithm)

Once the map has been generated you will write some methods to compute and visualize paths through the mountains. You will then instruct the program to select the path that has the lowest elevation change.



In the map above, brighter shades mean higher elevation.

#### 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.

## 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  $7.24 \times 10^{405}$  possible paths you could take starting from western side of the map and taking one step forward until you reach the eastern side.

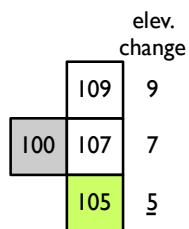
3011	2900	2852	2808	2791	2818
2972	2937	2886	2860	2830	2748
2937	2959	2913	2864	2791	2742
2999	2888	2986	2910	2821	2754
2909	2816	2893	2997	2962	2798

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

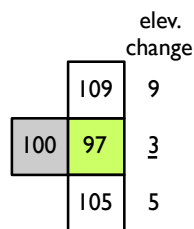
Since our map is in a 2D grid, we can envision a "walk" as starting in some cell at the left-most edge of the map (column 0) and proceeding forward by taking a "step" into one of the 3 adjacent cells in the next column over (column 1). Our "greedy walk" will assume that you will choose the cell whose elevation is closest to the elevation of the cell you're standing in.

(NOTE: this might mean walking uphill or downhill).

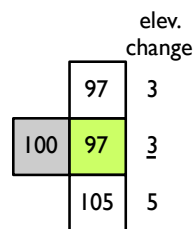
The diagrams below show a few scenarios for choosing where to take the next step. In the case of a tie with the forward position, you should always choose to go straight forward. In the case of a tie between the two non-forward locations, you should flip a coin to choose where to go.



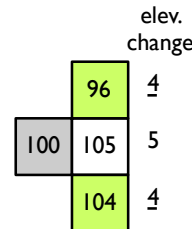
**Case 1:** smallest change is 5, go fwd-down



**Case 2:** smallest change is 3, go fwd



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



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

## Assignment Requirements

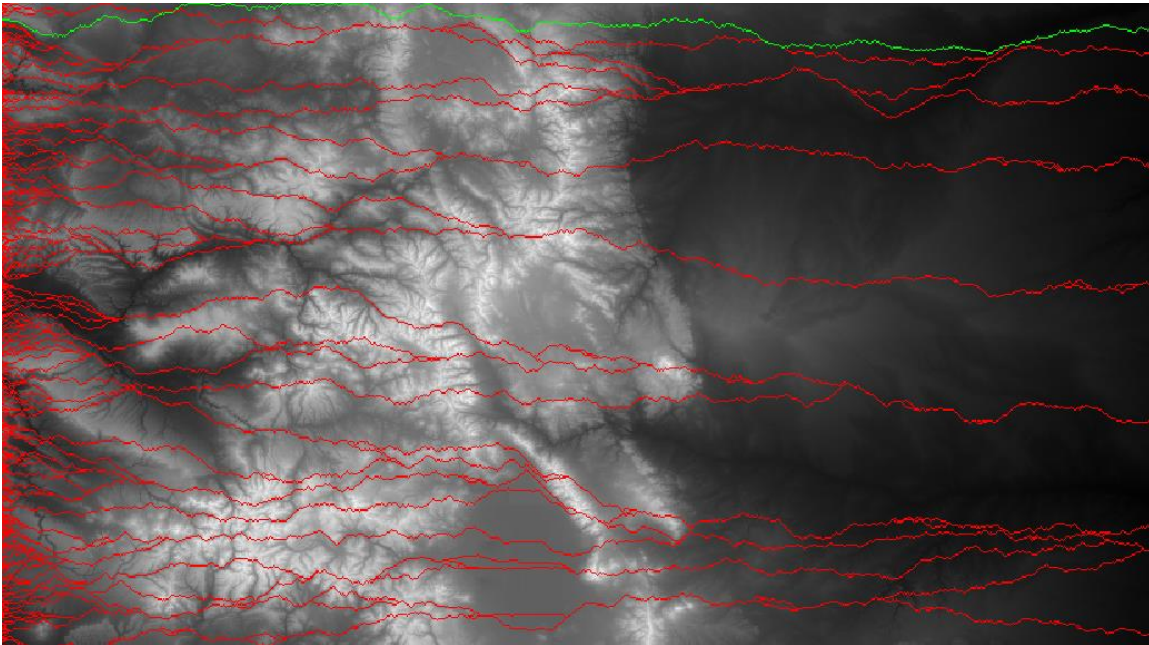
Step 1: Read the data into a 2D array

Step 2: Find min, max elevations, and other calculations on the data

Step 3: Draw the map

Step 4: Draw a lowest-elevation-change route starting from some row

Step 5: Find and draw the lowest-elevation-change route in the map (shown below in green)



To fulfill the requirements you are provided with a class and method stubs for completing parts of the project. You can make a copy of the *MapDrawer.java* file. The arguments can be modified to fit your needs. You do not have to adhere to this structure 100%.

**Task: Draw a lowest-elevation-change path from some row**

Implement the *int 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 should draw a line by drawing 1x1 filled rects in a different color on top of the existing drawing. The path should be drawn going West-to-East, starting from the given row using the greedy lowest-elevation-change technique described earlier.

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).
2. Write a loop that generates every possible column across the map from 1 to 840 (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 to green (or whatever color you choose).
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.

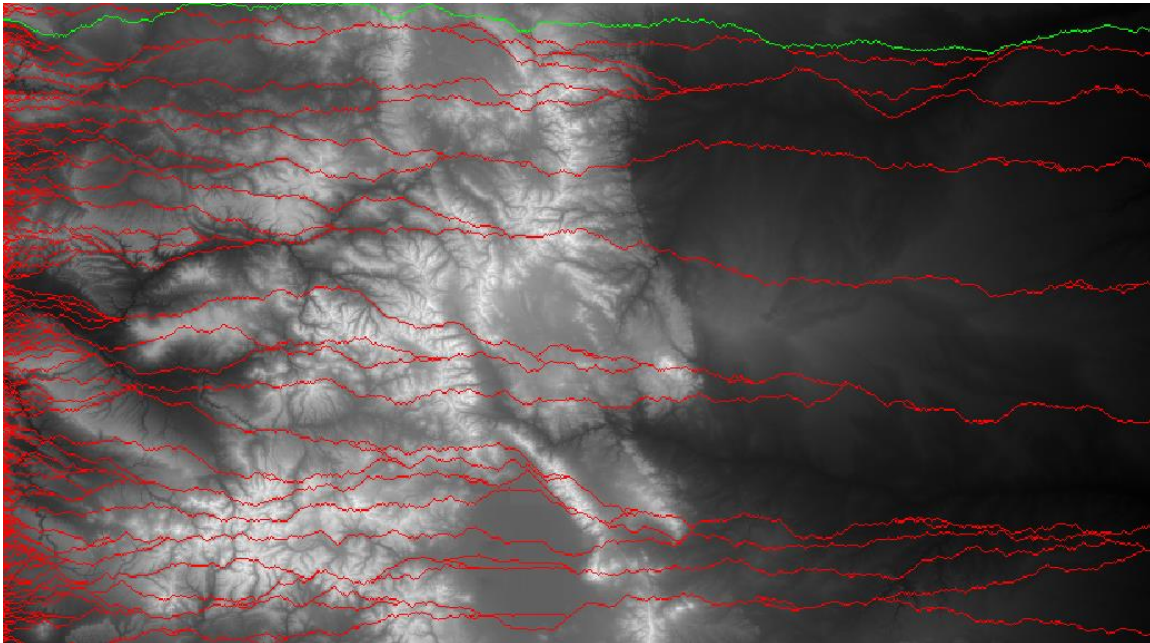
6. Continue finding the lowest neighboring cell and coloring it green as you go.
7. Keep a running total of the total elevation change that would be 'experienced' by a person walking this path. Since we consider an elevation change the absolute value (i.e. going 'uphill' 10 meters is the same amount 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.

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

Implement the *int 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 this:



### 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.

The best greedy walk for the Colorado map should return a total elevation change *somewhere around* 14,000 +/- 1000 (meters). The best path should start *somewhere close* to the southern edge of the map, something between row 5 and 20 or so.

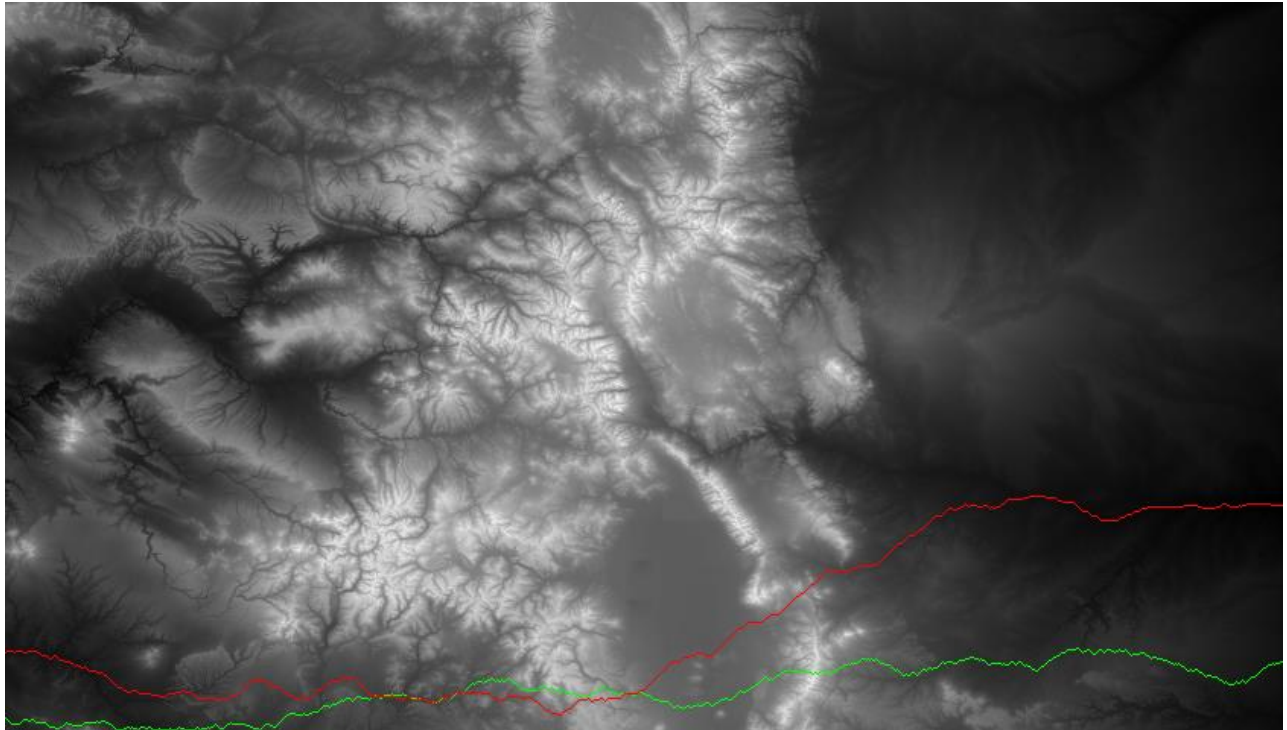
## Above and Beyond: Cool extensions

### 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.

- 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. There are many shortest path algorithms that are beyond the scope of this class. If you are curious, research them.
- In essence, it is possible to compute all 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 another 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.





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

## Graphics

- You could also shade the pixels you draw for the path to indicate their relative elevations. You could get the shading for “free” by using the alpha channel of the color you select.
- Do a drawing of the mountain in colors other than a monochromatic scale. This would give you more than 256 shades of color to work with. For example, what if you shaded the lower elevations green (for lush valleys) and the highest elevations some sort of rock color brown/tan. To do this arithmetically you need to look up how to 'interpolate' colors.