**Mountain Climber**

Two-dimensional (2D) arrays can be a bit mind-bending at first, but with a little practice, you'll see that they're not much more complicated than 1D arrays. Remember – an array is a single variable containing a list of elements of the same data type. 2D arrays are just a list of lists!

It might also help to think of a 2D array as a matrix. For a variable int[][] nums = new int[3][3], the sub-arrays nums[0], nums[1], and nums[2] would be the "rows" of the matrix.

Thus, row 0 consists of the elements nums[0][0], nums[0][1], and nums[0][2]. Column 0 contains the elements nums[1][0], nums[2][0], and nums[3][0]. The table below may help:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Col 0 | Col 1 | Col 2 |
| Row 0 | nums[0][0] | nums[0][1] | nums[0][2] |
| Row 1 | nums[1][0] | nums[1][1] | nums[1][2] |
| Row 2 | nums[2][0] | nums[2][1] | nums[2][2] |
| Row 3 | nums[3][0] | nums[3][1] | nums[3][2] |

1. Make a class TwoDRunner with a main() method, which you'll use to test the following methods.
2. Make a class TwoDArrays that will perform some useful operations on 2D arrays of integers.
   1. TwoDArrays should have one (private) instance variable, int[][] nums (the 2D array). An object of this class will represent a two-dimensional array, and any of the methods below may be called on a particular object.
   2. There should be one constructor, with an int[][] parameter that will be used to initialize nums (to the 2D array supplied to the constructor).
3. Complete the method public int sum(), that returns the sum of all values in nums.

new TwoDArrays(new int[][] {{1, 2}, {10, 11}}).sum() >>> 24

1. Complete the method public boolean isSquare(), that returns true if nums is square (i.e. has nbynrows and columns).

new TwoDArrays(new int[][] {{2, 3, 1}, {5, 4, 6}}).isSquare() >>> false

/\* The example above is not square because it has two elements, each three elements long. The array parameter is a "2 by 3", or rectangular (i.e. not square) array \*/

1. Complete the method public boolean inSequence(), that checks if the array is square and contains each of the digits from 1 to nums.length2, in numerical order. Utilize the method you just wrote to determine if nums is square.

new TwoDArrays(new int[][] {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}).inSequence() >>> true

1. Complete the method public int[] closestToZero(), that returnsan array containing the *index* of the value in nums that is closest to zero*.* You can assume there will be only one value closest to zero.

new TwoDArrays(new int[][] {{3, 1, 5, 7}, {4, 12, -3, 8}}).closestToZero() >>> {0, 1}

1. Complete the method public void addMatrix(int[][] other), that adds all values in other to nums, then prints the result*.* You can assume the arrays are the same size. See below for help printing a 2D array.

new TwoDArrays(new int[][] {{1,2,3}, {3,2,1}}).addMatrix(new int[][] {{2,3,1}, {3,1,2}}) >>> {{3,5,4}, {6,3,3}}

* 1. You hopefully remember that the method Arrays.toString() will return a String representation of the array parameter. However, *every element in a 2D array is an array*, so a call to this method with a 2D array of integers that has a length of three as a parameter will return something like this:

[[I@1909752, [I@1f96302, [I@14eac69]]

Add a method private void print() that will print the state of nums in a more useful fashion. To do this:

* + 1. Use a for-each loop to iterate through all the lists in the list of lists (check the powerpoints for more information on for-each loops).
    2. Print what the Arrays.toString() method returns with the "current" list as a parameter, then move to the next line.
  1. Add a call to the print() method at the end of the addMatrix() method.

1. (Riddle) The man was afraid to go home because the man with the mask was there. Why?
2. Complete the method public int columnSum(int col), that returns the sum of all the elements in the given column of nums – "empty" columns have a value of 0.

new TwoDArrays(new int[][] {{1, 2, 3}, {4, 5, 6}, {6}}).columnSum(2) >>> 9

/\* Be careful of rows of different lengths! Because a 2D array is a list of lists, here you can't assume *nums* will be square. \*/

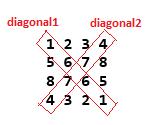
1. Complete the method public boolean isColumnMagic(), that will return true if nums is "column-magic". A matrix is column-magic if every column has the same column sum. You can't assume numswill be square or rectangular.

new TwoDArrays(new int[][] {{1, 2, 3}, {1}, {2, 2, 1}}).isColumnMagic() >>> true

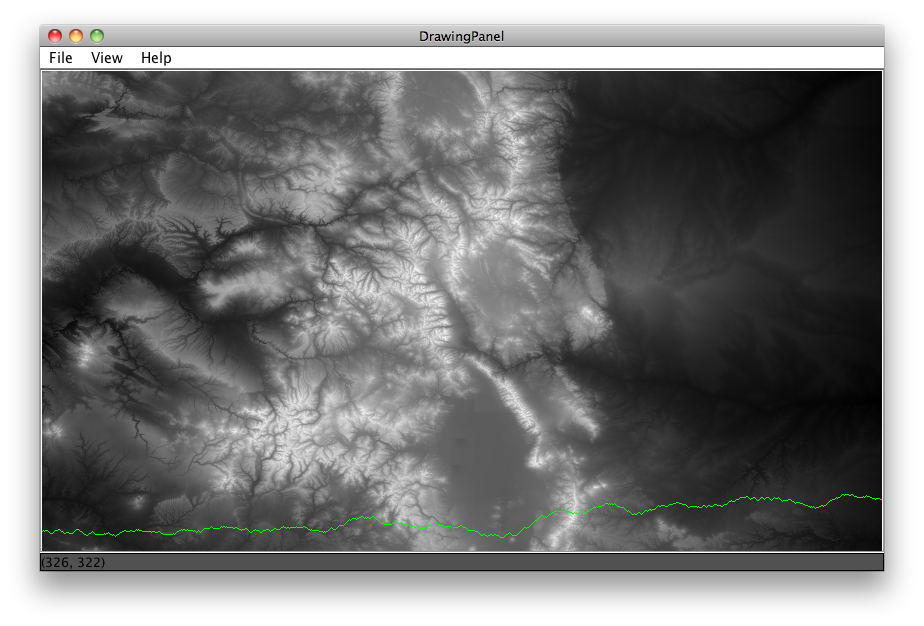
1. Complete the method public boolean sameDiagonalSums(), that returns true if the two major diagonal sums of nums (assumed square) are equal. For a challenge, do this with one loop.

new TwoDArrays(new int[][] {{1, 2, 3}, {3, 2, 1}, {7, 2, 2}}).sameDiagonalSums() >>> false

new TwoDArrays(new int[][] {{4, 9, 2}, {3, 5, 7}, {8, 1, 6}}).sameDiagonalSums() >>> true



**Mountain Climber**

In this lab, you will read a set of ***topographic*** (land elevation) data into a two-dimensional array and write some methods to compute paths through the mountains, as well as visualize them.

*In the map above, brighter shades represent higher elevation. The green line shows a lowest-elevation-change route for this map, traveling west-to-east.*

There are many contexts in which you may want to know the most efficient way to travel over land.

When traveling through mountains (assuming 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*** ***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 x 10405 possible paths you could take starting from western side of the map and taking one step forward until you reach the eastern side.

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" assumes that you will choose the cell whose elevation is closest to the elevation of the cell you're standing in (this might mean walking uphill or downhill).

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

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 forward. In the case of a tie between the two non-forward locations, you should choose randomly where to go. Note that there are other ways to choose a path through the mountains – these can be explored in the **(Advanced)** section of this assignment.



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

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

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

*Case 4: smallest is a tie, choose randomly between up or down*

A runner class and a class that provides a graphical view (GUI) have been provided. You will be adding code to the **MapDataDrawer** class (the below is just a description of the methods, read on for instructions).

**Instance variables**

|  |  |
| --- | --- |
| int[][] grid | two-dimensional array of integers, storing elevations at various points |

**Methods**

|  |  |
| --- | --- |
| MapDataDrawer(String fileName) | constructor, reads data from given file into the 2D array |
| int getRows()andint getCols() | getters, return the number of rows / columns in the map |
| int findMin() andint findMax() | returns the min / max elevation in the map |
| void drawMap(Graphics g) | draws this map in black & white using given the graphics context (paintbrush) |
| int drawLowestElevPath(Graphics g, int row) | draws the lowest elevation path, starting from the given row. Also returns the total elevation change for the path |
| int indexOfLowestElevPath(Graphics g) | finds and draws the lowest elevation change path in the entire map. Returns the starting row |

**Implement the labs as follows:**

The data is a plain text file from the [NOAA](http://www.noaa.gov/), representing the average elevations of patches of land in the US. The data file included with the project, "**Colorado\_480x480.txt**", is the elevation data for much of the state of Colorado (mountains!). The data comes as one large, space-separated list of integers. Each integer is the average elevation, in meters, of each cell in the grid.

1. Import the necessary files into your IDE of choice.
   1. **BlueJ:** download and extract the starter code folder. Copy the files into a new folder titled Lab06-MountainClimber on your H: drive. Double-click the **package.bluej** file to open.
   2. **Eclipse**: download and extract the starter code folder. Make a new Java project. Drag/drop the extracted .java files into the ***src*** folder, and the text file into the main project folder.
2. Complete the **MapDataDrawer** class' constructor as follows:
3. The first two integers in the text file represent the number of rows and columns in the file (you'll need a Scanner). It may be helpful to save these values. Next, instantiate grid as a new 2D array of integers, given the size specified in the file.
4. The data are given as a continuous stream of space-separated numbers; there are no line breaks in the file, so you can't use nextLine() and split() (and Integer.parseInt()). Use Scanner's nextInt() method to parse the values into your program.
5. Test your constructor before proceeding to the next step!Make sure that you're actually reading data into the 2D array. Do a sanity check by looking at a specific row and column and comparing with a friend, or against the original data itself.
6. Complete the getRows() and getCols() getter methods.
7. To (later) draw the map, you will need the min and max elevation values in the map. Implement the findMin()and findMax()methods. These methods should return the smallest and largest values, respectively, in the 2D array.
   1. Test these functions independently to make sure you're getting decent values. Maybe check with a friend to see if s/he is getting the same thing.
8. Implement the drawMap() method. This method is passed a Graphics object as a parameter. You don't need to know what this is; think of the g object as the graphics paintbrush. You will call drawing methods on this object, and what it draws will appear on screen.

This method will "draw" the 2D array of integers as a series of filled rectangles, each 1 by 1 pixel in size, with the color set to a gray-scale value between white and black.

* 1. The shade of gray should be scaled based on the elevation of the map at a particular location. In the red-blue-green (RGB) color model, if each RGB value is the same, you'll get a shade of gray. Thus, there are 256 possible shades of gray from black (0, 0, 0) to middle gray (128, 128, 128), to white (255, 255, 255). More info on color can be found in the lab folder.

To make the shade of gray**,** use the min and max values in the 2D array (you just wrote helper methods to find these!) to scale each integer to a value between 0 and 255, inclusive (yes, this requires math). You also need to set the fill color of the paintbrush before drawing the rectangle. Example:

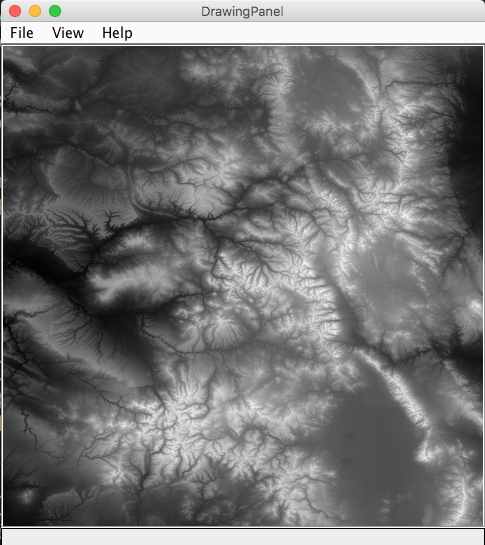
int color = //calculated grayscale RGB value

g.setColor(new Color(color, color, color)); //supply a new Color object

g.fillRect(x, y, 1, 1) //draws a filled rectangle, using the current Color

**NOTE!** Thegrid array stores the data in *row-major* ("normal") order - that's how it came out of the data file. However, graphics are drawn in *column-major* order. You need to flip the row and column (only) when drawing. If you're using grid[r][c], the call should be g.fillRect(c, r, 1, 1).

Done correctly, your map should look like this:

****

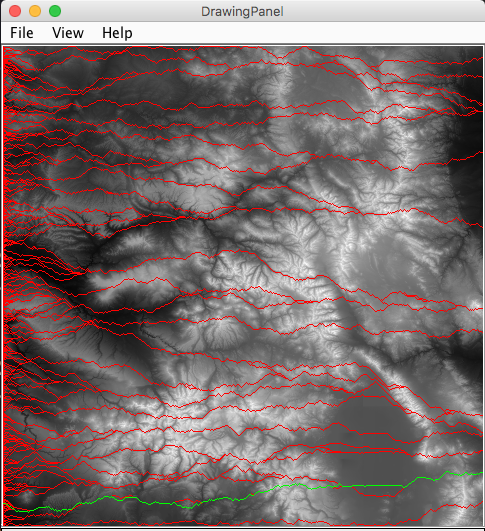
***If your map is taking a long time to draw, see*** [***here***](https://youtu.be/HtX1vrpdS1s)

1. Implement the drawLowestElevPath() method. Note that this method does two things: it draws a path "through the mountains" and returns to the total elevation change on that path.

The method "draws a line" by drawing filled rectangles, each 1 by 1 pixel, in a different color on top of the existing drawing (using the g object). The path should be drawn going West-to-East, starting from the given row using the "greedy" lowest-elevation-change technique described earlier.

1. Starting from the given row and column 0, draw a filled rectangle 1 by 1 pixel (the line color is set by the runner class, though you can change it if you'd like).
2. Write a loop that iterates through every column across the map, starting with column 1 (column 1 is the first column you need to make choice about where to "step"). For each column, you will decide where to take your next step – fwd, fwd-and-up, or fwd-and-down – using the greedy choice strategy described previously.
   * 1. Use Math.abs() to get the absolute value of the difference between two elevations.
3. Color in the location you're moving to (draw a filled rectangle).
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. Continue finding the lowest neighboring cell and coloring it in as you go.
6. Maintain a running total of the 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 ends up being a pretty large positive number. Check the **(Advanced)** section for better path-finding techniques (e.g. going downhill is NOT the same as going uphill).

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

1. Implement the indexOfLowestElevPath() method which finds the *overall* lowest-elevation-change path and returns the row it starts on.
2. Utilize the method you wrote in step 4 that returns the total elevation change for a path starting at a given row.

Write a loop that generates every possible starting row from 0 to grid.length - 1 and call your method using each possible starting row; you should find the lowest route for the entire map (as that method also draws all the routes, it's good feedback that the method is running properly).

You should end up with something that looks like the image shown to the right:

*Though the indexOfLowestPath() method accepts a Graphics object, this method should not use it to draw anything, but rather pass it on to the drawLowestElevPath() method that does the actual drawing. This is possibly a flaw in the program design, but a flaw designed to protect you from some complicated business.*

You can also clear the map by uncommenting the corresponding line in the Driver class after this calculation and simply draw the "best" path line. The result should look something close to the image above or the one shown on the first page of the assignment.

*The greedy walk should always produce a slightly different path as you are occasionally choosing a random direction, in the case of a tie in elevation change.*

The best greedy walk for the "Colorado\_480x480.txt" map should return a total elevation change (experienced by the walker, i.e. not just the sum of the elevations of all cells) somewhere around 11,000 +/- 1000 (meters), starting roughly on row 450.

**(Advanced) Better Path Finding**

The "greedy" lowest-elevation-change algorithm is fairly naïve. 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. A current "best choice" could lead to a series of terrible future choices.

* 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 that 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](http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm). The Wikipedia link should help.

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.

As 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 *significantly* lower (approximately half) than that of the standard greedy algorithm described previously.

**(Advanced) Better 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.

Adapted from the "Mountain Paths" project, by Baker Franke

[*nifty.stanford.edu/2016/franke-mountain-paths/*](http://nifty.stanford.edu/2016/franke-mountain-paths/)