**Prog 2: Take a Hike**

|  |  |  |  |
| --- | --- | --- | --- |
| 9-15 Update shown in **Blue**: Program including computing your optimal solution on the given dataset must run in 30 seconds or less on an "average" laptop.  9-20 Sample of final output and description shown in Purple.  **Contents**   1. [**1**Background](https://sites.google.com/site/uic342/prog-2-path-finder#TOC-Background) 2. [**2**Your Solution](https://sites.google.com/site/uic342/prog-2-path-finder#TOC-Your-Solution) 3. [**3**What is a "Greedy Walk"?](https://sites.google.com/site/uic342/prog-2-path-finder#TOC-What-is-a-Greedy-Walk-)    1. [**3.1**1. Read in the Data File](https://sites.google.com/site/uic342/prog-2-path-finder#TOC-1.-Read-in-the-Data-File)    2. [**3.2**2. Find min and max values](https://sites.google.com/site/uic342/prog-2-path-finder#TOC-2.-Find-min-and-max-values)    3. [**3.3**3. Draw the map.](https://sites.google.com/site/uic342/prog-2-path-finder#TOC-3.-Draw-the-map.-)    4. [**3.4**4. Draw a path.](https://sites.google.com/site/uic342/prog-2-path-finder#TOC-4.-Draw-a-path.-)    5. [**3.5**5. Find Best Greedy Solution](https://sites.google.com/site/uic342/prog-2-path-finder#TOC-5.-Find-Best-Greedy-Solution-)    6. [**3.6**6. The Fun/Challenging Part](https://sites.google.com/site/uic342/prog-2-path-finder#TOC-6.-The-Fun-Challenging-Part-)   **Background**  The ill-fated [Donner party](https://en.wikipedia.org/wiki/Donner_Party) attempted to cross the California Sierra Nevada mountains, unaware of how bad it was going to be.  Almost half of them (39 of 87) died in part because of a snow storm dumping [5 feet of snow](http://snowbrains.com/six-incredible-sierra-nevada-ca-snow-records/) on them.  This group was following the California Trail, which settlers in the 1800's used in covered wagons to follow sources of water across the mostly arid plains.  As the [California Trail](https://familysearch.org/wiki/en/California_Trail) site explains:   |  |  | | --- | --- | | Emigrants usually formed into wagon trains for security. Almost everyone preferred to walk rather than ride in dusty, bumpy wagons. They had to average 11 miles (18 km) to 17 miles (27 km) per day to reach California in four to six months. To leave too early risked muddy trails and too little grass for livestock. To arrive late risked traveling in winter weather. Thunderstorms and fierce winds were common. In good weather they often slept under the stars. On the prairie buffalo chips were gathered for use as cooking fuel. Wash day was about every two weeks. | [California Trail](https://familysearch.org/wiki/en/California_Trail)[Map from <https://familysearch.org/wiki/en/California_Trail>] |   It was difficult and dangerous to move heavy wagons [up and down](http://tomlaidlaw.com/otkiosks/otcc/hilgard.html)  [steep hills](http://twentytwowords.com/how-covered-wagons-were-able-to-be-pulled-up-steep-hills/).  If someone from San Francisco wanted to establish a trail through the mountains, what route should have been established between the California valley on the left through the mountains to Nevada on the right?  This path should minimize vertical elevation change (either up or down) on the path.  [This assignment is slightly modified from Baker Franke's Mountain Paths [assignment](http://nifty.stanford.edu/2016/franke-mountain-paths/), using the *DrawingPanel* class from [Building Java Programs](http://www.buildingjavaprograms.com/) by Reges and Stepp.]  Other problems of this sort are:   * 1. In building a new Interstate highway, where should it go?  We would want to minimize the need for excavating and filling in.   2. Which mountain(s) should be chosen to develop a new ski resort?  We would want to maximum rather than minimize slope.   3. In the Chicago area after heavy rains, as the waters rise on the Des Plaines river (or the North Branch of the Chicago River), forecast where workers should sandbag next.  For this you would need higher resolution data, such as from [here](https://www.sciencebase.gov/catalog/item/57356de1e4b0dae0d5de646e).   4. A bike route through a hilly region should prefer flatter terrain.   **Your Solution**  Below we can see the Google map of the area in question.  (Donner pass is just above and to the left of Lake Tahoe.) Below the Google map is a visualization of the dataset we will be using, with data scaled as 0..255 grayscale values.  Higher elevations are lighter gray.  Each pixel equals approximately 1 square mile.  **Google Map:**  [[https://sites.google.com/site/uic342/_/rsrc/1473367512320/prog-2-path-finder/GoogleMapOfArea.png?height=296&width=400](https://www.google.com/maps/@39.039497,-120.2520595,8.65z/data=!5m1!1e4)](https://www.google.com/maps/@39.039497,-120.2520595,8.65z/data=!5m1!1e4)  **Figure A**: Display map with no paths  [https://sites.google.com/site/uic342/_/rsrc/1473367405627/prog-2-path-finder/OurDataSet.png](https://sites.google.com/site/uic342/prog-2-path-finder/OurDataSet.png?attredirects=0)  In these images the origin (0,0) is at the upper-left.  Starting on the left-most column we can find the lowest point, which for this dataset turns out is row 91.  We can follow a greedy walk strategy where we always move to the right one column, to the nearest adjacent elevation which is either the same row or up/down one row (explained further below).  Setting the pixels on that path to red we get a path starting at row 91with total elevation change of 12,492 meters:  **Figure B**: Display map with a single path starting from lowest elevation in the left column.  This is shown in red below.  [Greedy Trail starting from Lowest Elevation in Left Column](https://sites.google.com/site/uic342/prog-2-path-finder/GreedyTrailFromLowestStart%20from%2091%20for%2012492.png?attredirects=0)  Similarly we can find the greedy paths from all starting locations, giving:  **Figure C:** Greedy Trails from all starting points.  [Greedy Trails from all starting locations](https://sites.google.com/site/uic342/prog-2-path-finder/All%20Greedy%20Trails.png?attredirects=0)  We can keep track of and print only the best path, which starts at row 15 and gives a total change of 7,101 meters.  Displaying this best greedy path in green along with our original path in red should look something like:  **Figure D**: Display map with best path using the greedy strategy (green), along with previous path (red)  [Best greedy overall, and from lowest starting location](https://sites.google.com/site/uic342/prog-2-path-finder/Best%20Greedy%20path%20overall%20and%20best%20from%20lowest%20start.png?attredirects=0)  Display the best possible path you can find after trying various strategies to give you **Figure E**(not shown).  In addition to the previously-found red and green paths, display your own optimal path in blue.  This should give you a single image with all three paths on it.  The total elevation change values for each of the paths should be shown in the console. One students's solution (Ashour Danka) is shown below, illustrating the three paths.  Everyone does not have to get the same solution for their optimal path, as this depends on the approach you use.  Figure E: Display map with best path using the greedy strategy (green), along with previous path (red), and proposed optimal path (blue) [Low-start greedy (red), Optimal greedy (green), Overall best (blue) by Ashour Danka](http://i.imgur.com/3II2joC.png)  **What is a "Greedy Walk"?**  A "greedy" algorithm is one in which you repeatedly make the local choice that seems best. 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. Our "greedy walk" will assume that in your walk 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.  The diagrams below show a few scenarios for choosing where to take the next step. Assume that in the case of a tie you flip a coin to choose where to go.   |  | | --- | | http://nifty.stanford.edu/2016/franke-mountain-paths/images/greedy2.png |   Note that each step you choose the cell with the minimum *elevation change*, not the cell with the minimum *value*.  **Steps to Take**  Start by downloading and unpacking [MountainPaths\_stuDistro.zip](https://sites.google.com/site/uic342/prog-2-path-finder/MountainPaths_stuDistro.zip?attredirects=0) (755K) and making a project out of those files. Write the code in class MapDataDrawer.java to complete the following general steps, which are each explained in more detail below:   * + Read the data into a 2D array of ints from within the MapDataDrawer class.   + Find the min and max elevations in the entire map   + Draw the map   + Find the lowest elevation in the far left column.  From that location plot the greedy walk path going to the far right column.   Class MapDataDrawer starts out as a skeleton, looking like this:  **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)          //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  Below are some additional explanations and point values (out of 55 total) for each step.  **1. Read in the Data File**  Write this code in the MapDataDrawer class constructor.  We will be using the [NevadaToCalifornia.txt](https://sites.google.com/site/uic342/prog-2-path-finder/NevadaToCalifornia.txt?attredirects=0) (113K) data file, though your program should work with any datafile of the proper format (such as [Colorado.txt](https://sites.google.com/site/uic342/prog-2-path-finder/Colorado.txt?attredirects=0) which is 2.8MB).  The first two lines of our data file look like the following:  117 212  803 778 860 1066 1131 1143 1240 1350 1500 1626 1773 1832 1942 1813 1668 1645 1746 1801 1698 1522 . . .  The first integer is the number of *rows* in the file, and the second integer is the number of *columns* in the file.  These are followed by all the values, which in this case are 117 x 212 = 24,804 integer values.  There are no line breaks in the data, so you have to keep track of which values go in each row,col as you read them in.  You should use the Scanner nextInt() method to read in the values.  You should create a private class variable that is a 2-dimensional array of ints to store this data.  Each integer represents the average elevation in meters for the corresponding square mile on the map.  If you want to create additional data sets of your own, visit [NOAA Grid Extract](http://maps.ngdc.noaa.gov/viewers/wcs-client/).  To extract a dataset you need to :   * 1. Zoom in on the area desired   2. Select an area by clicking on the "i" button in the upper-left, then selecting the region within the displayed map   3. For the Output Format select "ArcGIS ASCII Grid".  It takes a moment to download the data.   4. Replace the metadata at the top of the file with just the row and column values.  (Note that in the metadata it is in column,row order, not row,column).   **2. Find min and max values**  These should be found across all file values.  Write this code in the findMin() and findMax() methods  **3. Draw the map.**  **(10 points)**Write this code in the drawMap( Graphics) method.  Most of the functionality to do this is already written.  You just need to determine the color, using this information to draw a small rectangle corresponding to each cell in the map at the row,col location.  First use the findMin() and findMax() values to help figure out how to scale the map values into the 0..255 RGB range.  The same computed value will be used for all three of R, G, and B, thus making it a grayscale image.  The 256 shades of gray go from black (0,0,0) to middle gray (128,128,128) to white (255,255,255).  Set the color just before drawing each small rectangle using the g.fillRect() method.  Your code will be something like the following:  int c = ???;    //calculate the grayscale value g.setColor(new Color(c, c, c));    // set all 3 of the RGB colors to be the same 0..255 value // While data was stored as row,col, the graphing expects it to come in as col,row, so reverse it here: g.fillRect(col,row,1,1);         // Draw a 1x1 rectangle corresponding to row,col  Once you have finished this step and call it from within the Driver class, you should see something similar to Figure A (shown above) displayed.  **4. Draw a path.**  **(10 points)**Write this code in the drawLowestElevPath( Graphics, startRow) method.  You will have a loop to go through each column, starting from column 0 on the far left.  At each step you will draw the rectangle representing the current (row,col) location, then move one column to the right, moving either straight across, or up/down one row, accumulating the difference in elevation as an absolute value.       Note that the Driver code will have already set the drawing color to RED, so when you draw a rectangle here it will show up as RED. Note that there is a difference between moving to the *lowest elevation* rectangle to the right, and moving to the rectangle to the right that has the *smallest elevation difference* from the current rectangle.  The former represents the strategy where you always try to travel downhill, while the latter represents the minimum-vertical-change approach.       Once you are done you should have a map with a red path going from left to right, such as the one shown in **Figure B**above.  This code should also return the total accumulated elevation change.       Once you are done you should have a map with a red path going from left to right, such as the one shown in **Figure B**above.  This code should also return the total accumulated elevation change.  In the console output you should print the starting row and the total elevation change.  **5. Find Best Greedy Solution**  **(10 points)**Now repeat step 4, starting at every possible row in the first column.  Your output at this point might look like **Figure C**above.  Keep track of the best solution, and print just that one out at the end, such as the one shown in **Figure D** above.   In the console output again you should print the starting row and the total elevation change for the best solution.       If your results change each time you run your program, make sure you seed the instance of Random with the value 1 only once, then share that instance of random with the code where it is used.  **6. The Fun/Challenging Part**  **(25 points)**Now you can think about how to improve your results.  The interesting question is whether some *other* non-greedy strategy can in fact give a better solution.  You must start somewhere in the left-most column and end up in the right-most column, but otherwise can move wherever you want, with no penalty for longer paths, allowing moving back towards the left.  The evaluation function only considers *total vertical distance* travelled.  For our sample map, in each of the 212 columns of data you choose among 3 alternatives, giving a total of ~3212 possible paths (roughly 1.4 x 10101) so exhaustive search is clearly not an option.   Here are some alternatives you could try:   * 1. Start in the middle and work outwards in both directions   2. Start at the right and move to the left   3. Create multiple columns, finding lowest points in each column, and try to connect points in between   4. Consider more than just the 3 locations to the right.   5. Allow "backtracking" to move backwards temporarily (e.g. through a valley) so that you can subsequently go through a mountain pass.   6. Consider if shortest-path algorithms such as Floyd-Warshall or Dijkstra's might help you with one of your strategies.   Your program including computing your optimal solution on the given dataset must run in 30 seconds or less on an "average" laptop. |