Lab 3: Mountain Climbing

Course: COMP1110/1140/6710 — Structured Programming

Lab Objectives:

Introduce writing non-trivial array programs.

Practice using Enumerations.

Develop algorithmic intuitions for search procedures and binary search.

Understanding 2D arrays in Java.

Your goal should be to complete all activities during the two-hour lab. Make sure to discuss your solutions with your tutor, and to commit and push them to gitlab! If you don't finish the activities within the allocated lab time, you have until 23:59 on Friday make a reasonable attempt at all activities. Make sure to commit and push by then!

Git set-up

By now, if you have done previous lab exercises, you should be familiar with the procedure required to work on and submit a lab exercise. If you are still unsure, see the instructions for the week 2 lab exercises or the test project first.

In this lab, you will step into the role of a surveyor navigating a digital mountain range. You will be working with with a *heightmap*: a 2D grid of integers where each value represents the elevation of a specific coordinate. Somewhere within this grid lies a single, golden peak: the highest point on the map. Your objective is to develop a series of algorithms that can programmatically "climb" this terrain. You will move from cell to cell, analysing local gradients to determine which direction leads upward, eventually optimising your path to reach the summit in the fewest steps possible.

2D Array Primer: The Heightmap

We represent the heightmap as a Java 2D array, that is an array of arrays: one array for each row of the map. You can generate heightmaps using the Mountain Lab page, which lets you generate valid heightmaps of various sizes.

1. Creating the Grid

To create a 2D array in Java, you first specify the number of *rows* and then the number of *columns*.

Declaration:

```
// Create a grid that is 24 rows high and 12 columns wide int[][] heightmap = new int[24][12];
```

Confusingly, the type `int[24][12]` is actually 24 instances of the `int[12]` type: 24 rows of 12 columns each. But, Java reverses the order of the dimensions to make it resemble indexing (see below).

Direct Initialization:

When you copy data from the Mountain Lab, the values are wrapped in nested curly braces. Each inner set of braces represents one horizontal row.

```
int[][] smallMap = { {10, 20, 10}, // Row 0 {20, 50, 20}, // Row 1 {10, 20, 10} // Row 2 };
```

2. How to Access Data

To get a specific height, use two sets of square brackets: `map[row][column]`.

The first number is the Vertical position (which row you are in).

The second number is the Horizontal position (which column you are in).

Note: Java arrays are *0-indexed*. The top-left corner of your map is always `map[0][0]`.

3. Thinking in "Y" and "X"

When navigating the heightmap:

Row (y): Changing the first index moves you North or South.

Column (x): Changing the second index moves you East or West.

4. Finding Neighbours

To "look around" your current position `[y][x]` you adjust the indices:

North (Up): `map[y - 1][x]`

South (Down): `map[y + 1][x]`

East (Right): `map[y][x + 1]`

West (Left): `map[y][x - 1]`

It may be assumed that heights beyond the edges of the map are zero, or some other very low number.

Activity 1: Getting Directions

The repository already contains a file called `Direction.java`. It contains a definition for an `enum` with 5 values: `NORTH`, `EAST`, `SOUTH`, `WEST`, and `PEAK`. Do not edit this file!. So long as your Java file is in the same directory as the one above, you can refer to the `enum` type it defines. You do not need to define your own version of `Direction`.

Create a function

```
Direction lookAround(int y, int x, int[][] heightmap)
```

that looks at the neighbours of the given coordinate `[y][x]`. The function should compare the height of the current cell to the cells directly to the North, South, East, and West.

Return a `Direction` value representing the direction of the highest neighbour. This is either `Direction.NORTH`, `Direction.EAST`, `Direction.SOUTH`, `Direction.WEST`, or, if the current cell is higher than all its neighbours, `Direction.PEAK`.

Write suitable pre- and postconditions for this function.

Testing

You must thoroughly test your function in `main` using `assert` statements. Use the generator tool to produce valid heightmaps for your test cases. If you turn on the "flow arrows" option, you can see what direction should be returned for each coordinate in the heightmap.

Your testing should aim for full branch coverage: ensure to have enough tests such that every `Direction` value is returned at least once. Pay specific attention to *edge cases*, such as coordinates located on the literal edges of the array, where certain neighbours do not exist. Your function must be robust enough to handle these edges without crashing.

Activity 2: Climbing to the Summit

Note that in all the heightmaps generated by the Mountain Lab, *local optima* produce the *global optimum* – that is, the directions given by a correct implementation of `lookAround` will *always lead you strictly closer to the peak*. This means we can find the coordinates of the peak just by following the directions we get from `lookAround`, starting from position `[0][0]`.

Write a function:

```
int[] climb(int[][] heightmap)
```

This function, starting from `[0][0]`, "walks" to the peak of the mountain in the way described above. In this version, you will move one cell at a time, always stepping toward the highest neighbour until no higher ground can be found. The function should return a new array of 2 integers `{y,x}`: the coordinates of the peak.

Make a note of how many "steps" it takes to find the peak and have `climb` print a message with that number once it finds the peak, for example:

```
'climb' found the peak in 39 steps!
```

Write suitable pre- and postconditions for this function.

Testing

To test this function, we recommend using a variety of heightmaps with the peak at a variety of locations. You can find the expected peak in the Mountain Lab by looking for the golden highlighted cell.

Activity 3: Rectangular Binary Search

A one-step-at-a-time climb is reliable but slow. Because the heightmaps always lead you strictly towards the peak and never away from it, we can apply logic similar to binary search to find the location of the peak much faster.

For a map of size `m` by `n`, In the first iteration of the loop, we know the peak lies somewhere in the rectangle `[0][0]` to `[m][n]`. We could take the midpoint of this rectangle, `[m/2][n/2]` and `lookAround`. Based on the direction we get back, we could divide the rectangle we are looking at in half, either horizontally or vertically. This logic is similar to binary search, but in the 2D case. The function has the same inputs and outputs as `climb`, but should get the same answer in significantly fewer steps:

```
int[] fastTravel(int[][] heightmap)
```

Make a note of how many "steps" it takes to find the peak and have `fastTravel` print a message with that number once it finds the peak, for example:

```
'fastTravel' found the peak in 4 steps!
```

Testing

All of your test cases for `climb` should apply just as easily to `fastTravel`.

Committing and pushing

After verifying your program works, use `git` as before to add, commit, and push your changes:

```
$ git add Mountain.java # or whatever file name $ git commit -m "Completed mountain lab" $ git push
```

Lab complete!

Congratulations! You've now completed this lab exercise. Ensure you have talked about your code with your tutor before you leave the lab session, and that you have saved, `add`ed, `commit`ed, and `push`ed all your work to GitLab!