Computer Laboratory 2

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development

1 Essential information

- This assignment is due Tuesday September 27th at noon and will be turned in on gradescope
- The assignment is intended to be done in pairs, although you're allowed to work on your own if you want.
- You can start the assignment before your lab period, although I recommend not getting too far – the lab TAs often have useful hints, and I don't want you wasting too much time struggling with something that the TAs might announce in-class.
- For more rules see the lab rules document on canvas.

2 Introduction

This laboratory lab is intended to focus on a few core skills:

- Further practice writing python functions and loops
- Practice with simple python list syntax (loops by element, loops by index)
- Practice using some common python loop and list patterns.
- Begin self-designing algorithms to solve non-trivial problems
- Practice using python's Math library – especially practice looking details up about python's math library

2.1 Lists

Before beginning, it should be noted that this lab will heavily feature lists, which we will only have begun discussing in lab. Given how new this material is, don't be afraid to look up examples of the python syntax in the book. In particular, you should make sure you know:

- What a python list is
- How to create a list in a loop using the append method.

- How to access values from a list by index.
- How to loop over a list.
- How to get the size of a list.

If you don't know how to do one of these things stop now Chapter 4.1 in the textbook covers many of these, at least briefly. You are also free to discuss these ideas with the students around you, or the lab TAs (so long as your discussion is about the core syntax here, not lab02 itself.)

2.2 Problem Solving

Outside of lists, this lab is also designed to challenge your skills in crafting new computations, rather than implementing ones you've seen before. This is a core computer science skill, and one you should have some practice with. This is a skill that takes years to develop, don't worry if this type of thinking is still rather hard.

To help you along, a few general problem-solving hints:

- Look over the examples for each function in the tester code. Sometimes these make things clear that are hard to understand from the reading.
- Make sure that **you the human reading this** can perform each task by-hand. It might be worth grabbing pencil-paper (and a calculator) and doing an example of each problem before starting to program.
- Remember that you can write "testing code" of your own. As an example, many functions in this lab require a specific loop to loop over adjacent pairs of loop values rather than each individual element. Feel free to write code to just practice that loop. (Just don't forget to comment-out or delete this code later)
- Don't start programming a final-draft function until you feel you have a *Complete* solution to the problem. "partial" algorithms are often missing key decisions and lead to broken code.
- Remember, while problems will be phrased at a high-level, they relate to specific python programming ideas. Make sure to think carefully about how each problem "connects" to python data and numbers.
- Many programmers find it helpful to take notes on the design of a function in comments. This is known as pseudo code and it's highly recommended. This also helps you remember what your plan is in case you need to take a break.

3 Software environment setup

If you need a reminder on how to set up a project for lab02, we recommend opening the PDF for lab01. For brevity I'll skip the instructions here, and focus on a few core reminders:

- Each lab should have it's OWN PROJECT in PyCharm. This means each lab should be in it's own project, and the "create a project" setup from last lab will need to be repeated weekly.

- Make sure you are *organizing* your files. Not only will this help you in the future, if your disorganization causes you to *lose* homework files you may get a 0 on the assignment.
- Each assignment in this class should be it's own pycharm project. If pycharm is showing you lab02 and lab01 files at the same time you may have misconfigured your projects.
- It's always smart to double-check that your project is configured for python3, not python2.
- You are, of course, free to use software other than pycharm, but we cannot promise to help you with any issues that arise due to this decision. – we can only support you in programming on the pycharm platform.

4 Files

This lab will involve the following files:

- (REQUIRED) altitools.py – you will program this. This file will contain the altitools code library.
- (PROVIDED) altitools_tester.py – we will provide this. This file will contain code to help you test if your functions meet our basic expectations

5 Before you get started

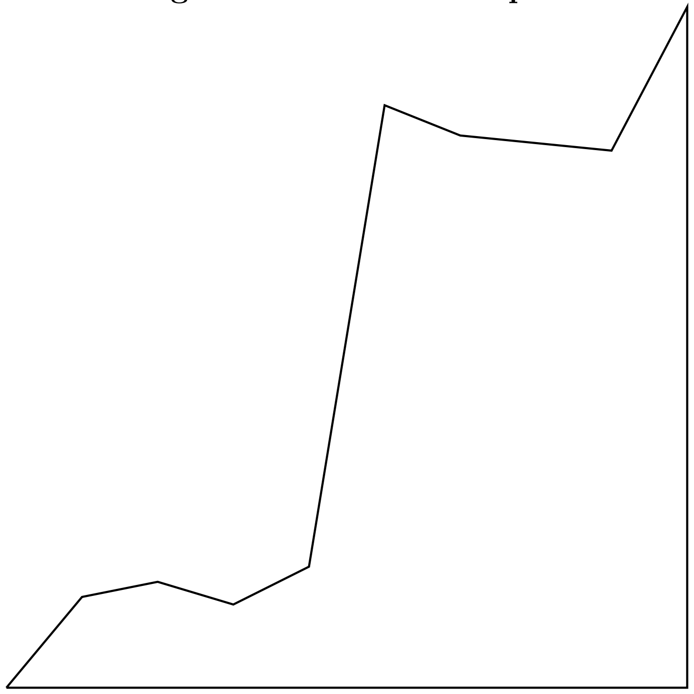
A few tasks to get "set up" before we get into the problem itself.

1. Make a pycharm project for this lab
2. Download the provided altitools_tester.py file and move the file so it's part of your pycharm project
3. Create a file named altitools.py There will be no template file from here on, you will be entirely in-charge of making your python files.
4. Add a comment to the top of that file with your name – remember, this is a required element of codestyle. skipping this step means losing points.
5. Read the PDF and make an *empty function* for each required function. Give the empty function a *plausible* return value (0 or `[]`) and then move on.
6. Run the tester code. It should output incorrect values *but* it shouldn't crash. Once you get to this point you are ready

6 Altitools

In this lab we will build "altitools.py" (The name altitools is a portmanteau of "altitude" and "tools" – altitools.) Because altitools.py is not, on it's own, a runable python file, we don't call it a program, instead we should refer to altitools.py as a software library (a collection of related, general-purpose, useful functions). In this case, the common theme of altitools will be functions to understand mountain ranges, or other paths with varied altitudes. (We'll be referring to these as mountain ranges, but the tool could easily support the relatively-small change in elevation when planning a bike route, or pedestrian path – anything that changes elevation over time it's length) A library like this might be used in software designed to support a range of navigation tasks, from hikers to bikers.

The first problem we face when building these tools is "**how will we represent a mountain range or other non-level path**". Consider the following "mountain range"



We will represent this mountain range with the following python tuple:

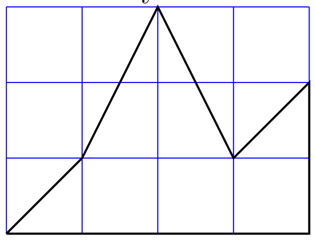
```
(0, 12, 14, 11, 16, 77, 73, 72, 71, 90)
```

The idea here is simple, at each horizontal mile we will measure the vertical height of the mountain and record it. So a range that is 4 horizontal miles long might be represented with a tuple of *5* numbers (position 0 being the 0th horizontal mile, position 1 being at the 1 horizontal mile mark, etc.)

As another example, consider the following tuple:

```
(0, 1, 3, 1, 2)
```

This would represent a hill 4 horizontal miles long, and is 3 miles tall at it's peak. Take a moment to try to draw this mountain range before going to the next page.



A few observations:

- The mountain may not start and stop at the same height.
- The numbers represent measures equally spaces out in the ”x-axis” (horizontal distance)
- The python tuple has 1 more measurement than the length of the mountain – this is due to the recording the “start height”.
- Many computations we may wish to see are based on the *segments* recorded here (the lines). The segments are not directly captured in the list, but can be computed using basic geometry/arithmetic.
- While our examples show integer values, obviously we could also use measurements like 12.45 miles. We’ve used integers for simplicity in our examples.

7 Formal Requirements

You will have several requirements for this lab:

1. a function to compute the steepest angle in the mountain.
2. a function to compute the total climb distance of the mountain.
3. a function to compute the longest “sequential climb” of the mountain.
4. a function to compute the number of peaks and valleys in the mountain range.
5. a function to compute a python list representing the changed mountain range if each valley lower than a certain point were to be filled in.
6. **You are also required to write at last one helper function of your own design.** (I recommend skipping ahead to read through this requirement, it’s better if you’re on the lookout for ways to do this as you attack the other problems, rather than saving this for the last thing you do.)

7.1 steepest_angle

The steepest_angle function is used to compute the angle (in degrees) of the steepest slope in the mountain range. As a note on geometry, we will not be differentiating between ”ascending” and ”descending” slopes for this function.

Since this class isn’t a course in applied geometry, we’re providing you with a basic outline for this function

- begin by finding the biggest change in heights (dy) between any two adjacent mountain height measurements. Note that this value should always be positive – whether the height is going up, or down, the change in height is a positive number.
- Python has a function `abs` which can take the absolute value, this may be of help here.

- Then, use the atan2 function from python's math module to compute the angle.
- The atan2 function takes two parameters, the change in y (dy) and the change in x (which would just be 1 in this case). If you need to know more about this function, it is reasonable to google search "python3 math atan2".
- You should look up the atan2 function to figure out if it returns it's angle in degrees or radians. Remember that the problem requires an answer in degrees. If a conversion is needed, don't forget to check if python has a function for that conversion, ad-hoc conversion factors may lead to incorrect answers if they are not precise to roughly 14 digits.

7.1.1 Requirements

- The steepest_angle will have one parameter: a tuple or list which contains numbers (representing heights)
- The list/tuple passed to this function should not be modified by the function.
- The return value of steepest_angle should be a number, representing the angle of the steepest segment of the mountain. This angle should be reported in degrees, not radians.
- If a list or tuple with fewer than 2 elements is provided the return value should be 0 (as you need at least 2 measurements to determine an angle)

7.2 total_distance

The length of a mountain range can be computed two different ways. The first is the horizontal distance. Think of this as the distance needed to "fly over" a mountain range. In our examples, this is essentially the length of the list minus one.

A mountain range, however, is not flat, and a mountain climber traveling the mountain range on the surface may have much greater distance to travel than the simple distance computation above. A mountain climber's travel distance will need to account for raises and falls in altitude, which can quite substantially increase how much a climber needs to travel. Moving one mile along the surface of the earth (as a bird might fly), but also moving one mile up, for example, would end up traveling 1.41 total feet (assuming they travel along a straight-line). This question asks about this second distance, which we'll generally refer to as the "travel distance" – the total distance a mountain climber would travel if they started at one end of the mountain range, and traveled to the other end taking no shortcuts.

Again, as this is not a class on applied geometry, we will give you an equation to get you started on computing this travel distance. For any two adjacent elements of the altitude list/tuple y_i and y_{i+1} the distance to travel from those two points is simply the euclidean distance:

$$distance_i = \sqrt{(y_i - y_{i+1})^2 + 1}$$

(Note – plus 1 to account for moving 1 mile along the "x-axis")

7.2.1 Requirements

- The total_distance function will have one parameter: a tuple or list which contains numbers (representing heights)
- The list/tuple passed to this function should not be modified by the function.
- The return value of the total_distance function should be an number, representing the sum-total travel distance of the surface of the mountain (as defined above)
- If a list with 1 or 0 elements is provided, a total distance of 0 should be returned

7.3 longest_sequential_climb

The longest_sequential_climb function computes the longest climb as you go over a mountain range. We say that a single segment of the mountain is a "climb" if the mountain gets higher as you go across that segment. (Note, this definition *is directional* unlike the steepest altitude question, which did not care about direction (slope up / slope down)) To phrase this more precisely (but also more mathematically) A segment from position i to $i + 1$ is a climb if the associated heights $y_i < y_{i+1}$ in the list/tuple.

We define a series of segments as a sequential climb if every segment in a row are all climbing. This can be as short as a single increasing segment, or as long as the entire list (should the whole list or tuple be increasing) We will define the *longest* sequential climb using our travel distance computation as seen in the previous question. In this way a climb that is *horizontally shorter* can sill be *longer* as defined by travel distance. For example, consider the list:
`[5, 4, 1, 2, 2.2, 2.3, 2.35, 2.37, 2.39, 2.4, 2.3, 2, 1, 100, 99, 98, 77]` This mountain has two sequential climbs (two runs of numbers that are uniformly increasing)

- `1, 2, 2.2, 2.3, 2.35, 2.37, 2.39, 2.4` (7.44 miles)
- `1, 100` (99.005 miles)

In this case, the function should return 99.005 miles.

7.3.1 Requirements

- The longest_sequential_climb function will have one parameter: a tuple or list which contains numbers (representing heights)
- The list/tuple passed to this function should not be modified by the function.
- The return value of the longest_sequential_climb function should be an number, representing the longest travel distance of any sequential climb in the input mountain range

- If a list with 1 or 0 elements is provided, a total distance of 0 should be returned

7.3.2 Hints

- This function is probably the hardest/longest of this assignment. Treat the task of writing an algorithm to solve it as a major part of the assignment all it's own.
- Like the problems before it, you should focus on first looping over *mountain range segments* rather than looping over *mountain position heights* (which is what a basic loop would do)
- Once you have a loop over mountain range segments. Focus on identifying: is this segment climbing or falling? You can build up from there.
- This problem *can* be solved using a single loop. This may not be the most straightforward solution. If you do try for this, be aware that you may need several if statements each loop, and several loop-variables.
- This is a big enough problem that you can attack it in one-complicated-pass, or in several steps. If you split the function into steps, make sure you're thinking about *how* to split it into steps. Certain splits make more sense in your head then in a computer.
- As an example **bad solution** – you might think 1) make a list of sequential climbs, 2) compute the travel distance of each climb, 3) take the maximum of the travel distances. This is a **hard to program** solution because it's quite hard to do step 1 using python lists.
- The hallmark of a *good solution* to this problem is how you use data and variables. Think carefully about what is easy to compute and what isn't.

7.4 num_of_peaks_and_valleys

We define a peak as a position on the mountain (a specific index into the list/tuple) that is larger than both of it's immediate neighbors. We define a valley as a position on the mountain (a specific index into the list/tuple) that is smaller than both of it's immediate neighbors. For both definitions, the first and last measurement in the mountain range (the first/last position in the list/tuple) cannot be a peak or mountain.

The num_of_peaks_and_valleys function should count how many peaks and valleys are in the mountain range and return both values.

7.4.1 Requirements

- The num_of_peaks_and_valleys function will have one parameter: a tuple or list which contains numbers (representing heights)
- The list/tuple passed to this function should not be modified by the function.

- The return value of the num_of_peaks_and_valleys function should be a tuple with two elements:
    1. the first position of the tuple should be the number of peaks in the mountain range
    2. the second position of the tuple should be the number of valleys in the mountain range
- If a list or tuple with 2 or fewer elements is provided the tuple `(0,0)` should be returned.

7.4.2 Hint

We'll see more over the week, but the python syntax for returning a tuple is EASY. It would look like this: `return apple, tree` You may find it easier to understand with parenthesis and proper variable names, but this is *enough*. When done well it *almost* looks like you're just returning 2 things.

7.5 fill_valleys

This last function is more hypothetical: What would happen if we were to fill in the low valleys of a mountain range with sand, so that each low part is brought up to a specific minimum height? To answer this question, the fill_valleys function should take a list or tuple representing a mountain range, and return a new list where all heights below a given minimum have been filled to the given minimum.

7.5.1 Requirements

- The function will have two parameters: a tuple or list which contains numbers (representing heights), and a number representing the new target minimum height.
    - Be careful here – this function can take a list OR a tuple. Some approaches to this problem will struggle if a tuple is input.
- The return value of the function should be a new list
    - The new list should have the same size as the input list
    - Any position in the old list that is lower than the input target minimum should be changed to the target minimum in the new list
    - Any other position in the old list should be copied as-is into the new list.
- Calling this function should not change a list provided to it.

7.6 Additional requirements

As an additional requirement. You are expected to create at least one "helper function". The term "helper function" refers to any function you write to make your job easier, rather than one that is immediately required. We will place very few specific requirements on this function, you can name it whatever you want, and it can have whatever role in the program you think would be most useful.

A good helper functions should:

- Do something you don't already have a function for
- Be used in a way to make other functions easier to write or read
- Be easy to describe the purpose of the function, without saying how it works.
- For example, we can explain the purpose of sqrt, or num_of_peaks_and_valleys without saying how to build those functions

A bad helper function would:

- Do something you already have a better way to do, such as functions: "plus", "times", "square_root", "squared", or "print_number"
- Not be used at all, or only used in one place, when it might be useful in more than one place.
- Not contribute at all to the ease of reading or writing the program (if this requirement feels like it's making the homework problem harder overall, you might be missing the point of this requirement)
- Be arbitrary, and hard to separate it's purpose from it's implementation. (As an extreme example of this, you could, arbitrarily cut a function in half and make the second half a helper function. You would likely end up with a helper function that doesn't make much sense on it's own)

We have not specifically planned one "perfect helper function" into this problem, so if you're looking for an obvious gap for this requirement to slot into, you might not find one. We have, however, identified several different options that strike us as particularly good. Some good choices create possible useful functions that would fit into the altitools library on their own, and also make another function easy to write. Other options eliminate a simple enough, but non-trivial, repeated computation (I.E. there's a mathematical computation that's needed in more than one required function which could be a great helper function.) Because the grading on this will be relatively generous, don't try to "guess" out exact ideas, just keep an eye out for ways you could make writing, or testing, your program easier with a helper function. (And don't forget to actually call your helper function. if you don't you'll lose points.)

8 Grading

Grading will be split between 50% automated tests, and 50% manual tests.

The automatic tests, if used correctly, can tell you about many, but not all, of the issues in your code Please make a point to read, and try to understand, any issues raised by the autograder on your own, before reaching out for help. The following listing of what the tests seek to do can be used to guide your efforts here, in particular, **the first three tests listed below** focus on simple issues that prevent all other testing (incorrectly named files, that sort of thing)

8.1 Automatic tests

If you are having trouble with the autograder – I recommend checking tests in the following order (unfortunately, we don't currently know how to force the tests to render with a specific order)

- 2 points: `test_submitted_files` tests that the files you submit have the expected name (altitools.py) Failing this means every other test fails
- 1 points: `test_import` tests that your file can be imported as a python module. Not being able to import your python file means there is something in your code actively preventing an import, such as a code error or input statement.
- 2 points: `test_expected_functions` tests that the expected function definitions are provided This will fair if any of the functions provided are misnamed.
- The remaining 45 points worth of tests are distributed evenly over the 5 required functions (9 points each) and are based on the tests in the tester

For manual grading, a rubric similar to this will be used:

- 5 points for your helper function.
  - For all 5 points, you should have a helper function that is a reasonably well-contained function (something with a small number of clear inputs and a clear output.)
  - A 4 point helper function is called by your code and clearly does *something.*
  - A 3 point helper function is called, but unnecessary (so it can either be removed from your code without consequence, or could be replaced by another existing function, or operator.)
  - A 2 point helper function exists, and does *something.*
  - A 1 point helper function exists, but does not clearly do anything, it may not even be valid python code.

  - A 0 point helper function does not exist. To get 0/5 you must not have turned in *any* non-required function

- 10 points code style. As with Lab01, we have a few specific requirements, but also generally require you to use best judgment on how to make sure your code is not just functional, but *readable*.
  - Code contains a header documenting who worked on the file
  - Each function has a documentation string in the appropriate place for python code (For examples, see Lab1's template file.)
  - The code is overall readable
- 7 points each for the remaining 5 functions.