Computer Laboratory 6

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

1 Essential information

- This assignment is due Tuesday October 25th 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.
- It will be VERY HELPFUL to find your Lab02 submission and have that ready to reference.

2 Introduction

In this lab we will get our first taste of Java. Rather than picking a simple task, we will be making a relatively large program. Fear not, however, because we will not be programming something new. Instead, we will be rebuilding our Altitools library as a java class, translating the design directly from the python version to Java. The end result will be a collection of static functions that will let us analyze a series of heights.

In particular, this lab will:

- Give you a chance to make sure you have a setup for Java programming.
- Practice translating a program you've already written from one language to another.
- Give you practical experience with Java coding.

I feel compelled to note that simply translating code from python to java like this is not necessarily a common task. While this will lead to working code – it will not be very *java style* code – that is, it will not make use of organizational techniques common in java code. That said, this will serve as GREAT practice for where our skills are NOW – there will be time in future labs to see the principals of a more object-oriented design. For now, focus only on gaining comfort with java's syntax through simple practice, and on how two different programming languages can be used to encode the same computation.

3 Software environment setup

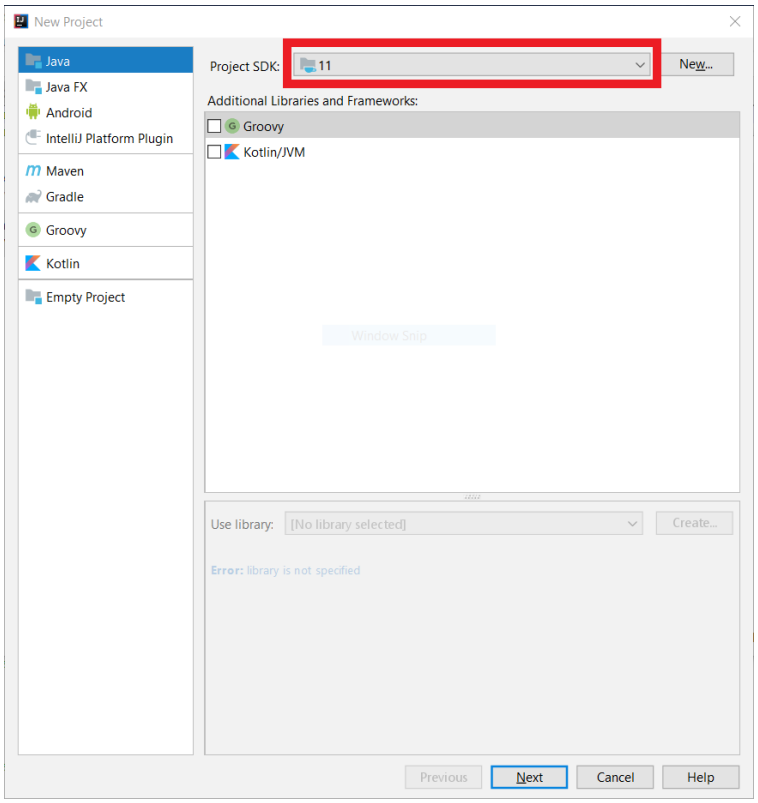
From this lab on we will be using the IntelliJ Idea IDE (integrated development environment). This section describes how to set up a new project for this lab and the Java files you are expected to download or create. (It should be noted that these figures may be slightly different from what you see depending on the exact version of IntelliJ you have – but the information should be the same)

3.1 Start IntelliJ and create a new Java Project

Begin by loading IntelliJ. Depending on your past use of IntelliJ you may need to enter some one-time settings such as color theme and default set of keyboard shortcuts for IntelliJ. You can set these options however you want, while I prefer default key-bindings and a light theme, many people may disagree, and it doesn't effect your coding output one bit. If you are not prompted for these settings, that too is fine. You can change these in the settings at your leisure.

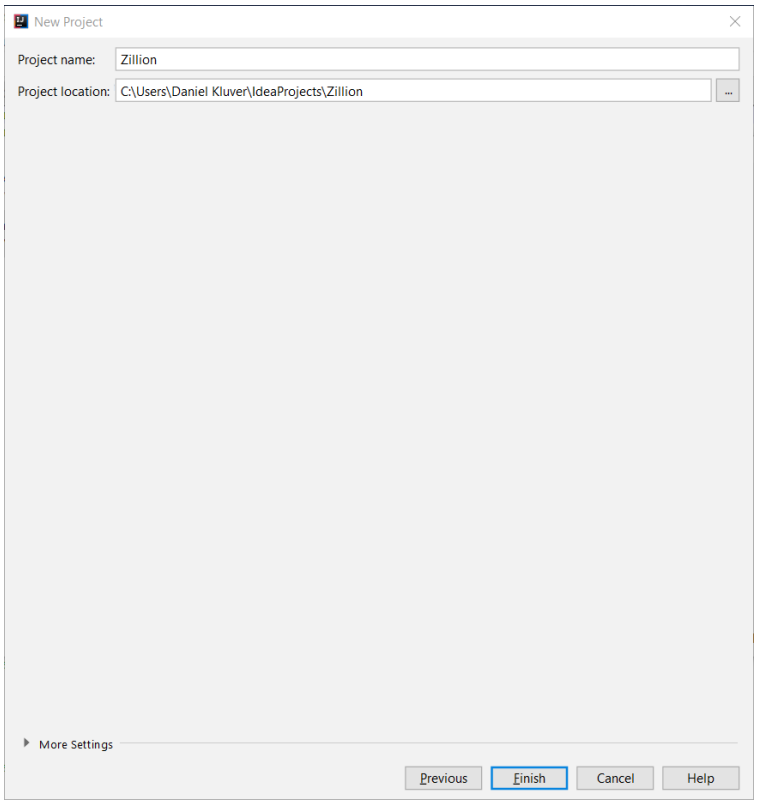
Once open, depending on settings and past use IntelliJ will either open to it's splash-screen or a previously opened project. Either way you want to **make a new project**. If you are in a previous project go to File → new → Project. Otherwise you can simply click the new project option.

The new project window should look something like this:



You want a Java project, with no additional library or framework support. If you are working on your own computer **don't forget to check the Project SDK** settings. You shouldn't need to be using an SDK version 11, like shown in the screenshot, so long as you are using 9 or above, you should be fine. If no SDK is listed, click new and setup your JDK. Typically IntelliJ will automatically open to your JDK (Java Development Kit) location.

Click next. This should bring you to the new project name and location seen:



Make sure your project has a name ("lab6" would be appropriate, but so would "altitools") **Make sure to check the project location** If necessary, update where the project is stored to keep your files well organized.

Click Finish and it should bring you to the main IntelliJ screen. If not shown, you will likely want to open the project view. The keyboard command for that is alt-1 on windows (and I believe this is true for Linux as well) This keyboard command is also shown in the background of the file viewer if no files are open.

3.2 Setup for Java files

Now that you have a project set up, click the tiny triangles in the project view to open the project itself and the source folder (src).

Download the template java files from canvas and move them to the src folder. You can do this by clicking and dragging the file from your file browser to IntelliJ's window. Make sure the file is in src, if it is not IntelliJ will not let you run this file.

Note Java, as a compiled language, has no patience for code it doesn't understand. Therefore, you may need to comment out parts of the testing code relating to functions you haven't written yet. So long as you have function calls to functions you haven't written Java wont run ANY of your code (even in unrelated files. Compiled languages tend to be all-or-nothing that way.) The template files provided should prevent this from being an issue this time but please remember this fact for future labs.

4 Files

This lab will involve the following files

- `Altitools.java` - A template of this file is provided. You will be completing this file to build the code for the altitools library
- `AltitoolsTester.java` – This file is provided. This file contains tests for the Altitools functions. This should be your primary debugging tool for this lab, as the java autograder is typically less clear, and **much more fussy** than the python autograder due to java's compiled nature (most of the autograder can't run if you don't provide it validish code)

5 Instructions

Before beginning you should:

1. Setup an IntelliJ package
2. Download the 2 provided files and place them in the src folder.
3. Ensure that AltitoolsTester can be run – it will not produce the correct outputs, but it should be runable without modifications.
4. Get a copy of your Lab02. If you do not have a copy of your original Lab02 – you can request a copy from another group, but if you do this you must add a comment to the top of the Altitools.java file to make this clear.
5. Update the header comment in the provided Altitools file with your name, the name of any lab partners, and if you are not working from your lab02, the name of who you got lab02 from. Remember the expectation is that the comment names everyone who contributed meaningfully to the code file it's in.
6. Review Lab02.pdf to make sure you remember the basic structure of the Altitools problem – this will not be restated here.
7. Skim the formal requirements section and the java syntax sections. You will need information from both sections to complete this task.
8. Only after skimming the rest of this document fully should you start programming.
9. **DO NOT SKIP THE GRADING AND TESTING INSTRUCTIONS AT THE VERY END** The java autograder is a bit of a different beast than the python autograder and there are critical details in these sections.

6 Formal Requirements

The five required functions are:

1. `public static double steepest_angle(double [] mountainRange)`
2. `public static double total_distance(double [] mountainRange)`
3. `public static double longest_sequential_climb(double [] mountainRange)`
4. `public static int[] num_of_peaks_and_valleys(double [] mountainRange)`
5. `public static double[] fill_valleys(double [] mountainRange, double target)`

6.1 steepest_angle

`public static double steepest_angle(double [] mountainRange)`

The steepest_angle function is used to compute the angle (in degrees) of the steepest slope in the mountain range. As before – we will not be differentiating between "ascending" and "descending" slopes for this function. The function takes an array of double values representing the mountain range and returns a double recording the steepest angle in degrees.

Notes

- As before – do not modify the mountainRange array itself in this function
- Java has similar mathematics functions to python (abs, atan2) in the "java.lang.Math" module. See the language notes for examples about using this.

6.2 total_distance

`public static double total_distance(double [] mountainRange)`

The total_distance function is used to get the total distance someone would cross if they traveled over the surface of a mountain range. As before – use the following equation for the distance between two height measurements.

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

The function takes an array of double values representing the mountain range and returns a double for the total travel distance.

Notes

- As before – do not modify the mountainRange array itself in this function

6.3 longest_sequential_climb

`public static double longest_sequential_climb(double [] mountainRange)`

The longest_sequential_climb function is used to indicate the longest consecutively increasing travel distance of any segment in the mountain range array. As before, be thoughtful here – this isn't the largest number of array positions increasing – it's the longest travel distance, which could occur over a shorter part of the array if the mountain is particularly

steep at those locations. The function takes a double array for the mountain range and returns it's value as a single double.

Notes

- As before – you are not allowed to change the mountainRange array in this problem.
- You may have used a very list-heavy solution to this problem in the past. This is not recommended here.
- I recommend a 1-loop solution. This takes a bit more work to come to, but is easier to do without a list-data structure. The key of this algorithm is a series of variables:
  - The current position in the array.
  - The length of the current sequential climb (if in fact we are climbing)
  - The longest sequential climb you've seen so far
- If you switch to the 1-loop solution, make sure you're thoughtful of the situation where the longest segment occurs at the end of the array. You may need to check if the "current sequential climb" variable is longer than the "longest sequential climb" variable after your loop depending on how you updated each variable in the loop.
- **to be clear** you don't have to use the "1-loop" solution (it's recommended, not required) you are free to solve this problem however you want. Just remember that you can't append to a list here, so an algorithm that *needs* that behavior will need to be updated.

6.4 num_of_peaks_and_valleys

```
public static int[] num_of_peaks_and_valleys(double [] mountainRange)
```

This function computes the number of peaks in the mountainRange array (positions that are higher than their immediate neighbor) and valleys (positions that are lower than their immediate neighbors). Since java does not have anything quite like python's tuples we will make this function return a new array with two values. If you compute peaks and valleys you can do this with the following:

```
return new int[]{peaks, valleys};
```

Notes

- As before – you are not allowed to change the mountainRange array in this problem.
- Java also does not have python's "comparison chaining" so if you used to have `x < y < z` you now need `x < y && y < z`

6.5 fill_valleys

```
public static double[] fill_valleys(double [] mountainRange, double target)
```

This function is used to understand what would happen if we were to fill in all low points in a given mountain range up to a fixed height. This function takes two parameters, a double array indicating the mountain range itself, and a "target" value. The function should return

a newly created array that is a copy of the input mountainRange where each value less than the target has been increased to the target.

Notes

- Because arrays cannot have their size changes after they are created and do not have an "append" function you likely will need a slightly different approach here.
- The common approach is to make a new array of the "correct" size. This new array will initially be full of 0, but you can fill it in with different values using a loop.

7 Java Syntax Guide

Some java has been provided in the provided Altitools template file. This mostly includes:

- Function declarations
- proper javadocs
- A mostly-complete header-comment
- A single provided function which may serve as a useful example of some common syntax.

The syntax for these topics will not be explored here.

7.1 Math differences and other minor notes

Most mathematical computation does not need to be translated. The only differences:

- Logical operators are back to their C-style versions: `||`(or) `&&`(and) `!`(not)
- We no longer have exponentiation operator `**` or int-division operator `\\`
- The standard division operator now has type-dependent behavior. If you give it two integers then it will perform an integer division. Otherwise we get float-division.
- Boolean values are now lower-case `true` and `false`
- All statements should end with a semicolon. (Not needed with flow-of-control, but needed for everything else)
- The return statement is unchanged.
- printing is done with `System.out.println()`,`System.out.println(something)`,`System.out.print(something)`
  - `System.out.println()` with 0 parameters, this prints the newline character only
  - `System.out.println(something)` only 1 parameter is allowed at max. This will print `something` then print the newline character
  - `System.out.print(something)` only 1 parameter is allowed at max. This print `something` but NOT print the newline character, similar to `print(something, end='')` in python
- Strings must have double-quotes and only double-quotes.

Finally, java's import statements and math library are slightly different:

Task	Python	Java

Import the math library	`# One of the following`
	`from math import atan2`
	`from math import *`
	`import math`
		`import java.lang.math;`
		`// this is in the template`
		`// file already`

absolute value	`abs(-2)`	`Math.abs(-2)`
sqrt	`sqrt(4)`	`Math.sqrt(4)`
atan2	`atan2(y,x)`	`Math.atan2(y,x)`
exponentiation	`a**b`	`Math.pow(a,b)`
to degrees	`degrees(3.14)`	`Math.toDegrees(3.14)`

7.2 Variables

Variables must have a type in java. For most variables in this lab the `int` type will be sufficient. Once created, variable assignments require no real modification.

Task	Python	Java
new int variables	`count = 0`	`int count = 0;`
update existing variable	`count = count + 1`	`count = count + 1;`

7.3 Basic flow of control

Task	Python	Java

if statement	`if condition:`
	`  tabbed body`
		`if (condition) {`
		`  tabbed body;`
		`}`

if/else statement	`if condition:`
	`  tabbed body`
	`else:`
	`  tabbed body`
		`if (condition) {`
		`  tabbed body;`
		`} else {`
		`  tabbed body;`
		`}`

if/else-if/else statement	`if condition:`
	`  tabbed body`
	`elif condition:`
	`  tabbed body`
	`else:`
	`  tabbed body`
		`if (condition) {`
		`  tabbed body;`
		`} else if (condition) {`
		`  tabbed body;`
		`} else {`
		`  tabbed body;`
		`}`

while loop	`while condition:`
	`  tabbed body`
		`while (condition) {`
		`  tabbed body;`
		`}`

for loop (repeat N times)	`for i in range(N):`
	`  tabbed body`
		`for (int i = 0; i < N; i++) {`
		`  tabbed body;`
		`}`

for loop (loop over a sequence)	`for i in range(len(lst)):`
	`  use lst[i]`
	`OR`
	`for elem in lst:`
	`  use elem`
		`for (int i=0; i<arr.length; i++) {`
		`  use arr[i]`
		`}`

7.4 arrays and lists

Arrays and lists are the biggest difference in this code. Python lists are created empty, and brought up to length by filling them with the append method. Java arrays are created at their final (and only) size, and cannot be resized once made. Therefore code for building and returning a list is quite different.

Task	Python	Java
create a new list/array	`lst = []`	`int[] arr = new int[10];`
update an existing position in the list/array	`lst[i] = newValue`	`arr[i] = newValue;`
reading from a list/array	`var = lst[i]`	`int var = arr[i];`
getting the size of a list/array	`size = len(lst);`	`int size = arr.length;`

"filling" a list/array	`squares = []`
	`for i in range(10):`
	`  squares.append(i*i);`
		`int[] squares = new int[10];`
		`for (int i = 0; i < 10; i++) {`
		`  square[i] = i*i;`
		`}`

for loop (loop over a list/array)	`for i in range(len(lst)):`
	`  use lst[i]`
	`OR`
	`for elem in lst:`
	`  use elem`
		`for (int i=0; i<arr.length; i++) {`
		`  use arr[i]`
		`}`

return a list/array from a function	`return lst`	`return arr;`

8 Java Autograder Notes

The java autograder is much fussier than the python autograder. This is mostly due to Java's nature as a compiled language. While we could often *partially* test python files that had errors in some functions – for your code to be testable your code must compile, have all the correct function definitions, and all the correct names. Even once you get the tests *running* you should also be aware that the autograder for java does not give as clear feedback as the python autograder when things go wrong. As such **you are expected to be testing you code on your own**. Do not use the autograder as a debugging tool.

Due to some strange things in the autograder, we have a few specific requirements for your code:

1. The code you submit must be valid and compileable java code. (We cannot test code with syntax errors. If we can't test it, we can't give you autograder credit for it)

2. The code you submit must have no package statement in it. (We have to do some weird things with package structures to get the code to compile correctly.) **YOU WILL GET 0 points if your code has a package statement!** Your IDE may have put one in without you noticing so you should check for this. If you see code like `package com.whatever;` at the top of your file you need to delete it.
3. The code you submit must match the provided function signature (name, parameter types, and modifiers) EXACTLY. Any mis-match, no matter how minor, may cause the test code to not compile, which would prevent testing.

As a general rule, if you started with the template file on canvas and didn't modify anything other than the function bodies you should be fine this time. Likewise, if your code can be run with the provided testing code (with no modifications to this testing code) you should be fine. If you do run into testing issues, do not hesitate to reach out to course staff to help with debugging.

9 Submission

For this lab you only need to turn in `Altitools.java`. You can submit other files if you wish, but we do not promise to look at them during grading. Your submission must be entered into gradescope before the beginning of your next lab, although it can be up to 24 hours late for a 10% deduction.

Grading on this assignment will follow the given general rubric:

- 50 points autograder – 10 points per function based on the test file.
- 10 points code style:
  - Many of the code-style elements have been provided for you in the template file, and all of them are just the java-versions of python code style requirements:
  - Your name (and the name of any collaborators on lab02 or lab06) should be in a comment at the top of the file. A comment for this has been provided, but IntelliJ might be "hiding" it from you.
  - Correctly formatted JavaDoc (this is the `/** ... */` comment blocks placed before each function that describes the function
  - Good variable names
  - Consistent tabbing (while java does not require this the same way python did – we **do** require it. Reading and grading poorly tabbed code is a painful experience)
  - Outside of these explicit requirements, as normal, we will generally be looking for (and giving feedback on) anything that makes the code harder/more painful to read, and will take away points if it's particularly bad.
- 8 points per function manual grading – broadly we're looking for things you did wrong which the autograder might not have caught.