Computer Science 1 — CSci 1100
Lab 10 — Closest Point Algorithm
Fall Semester 2017

Lab Overview

Algorithms that compute geometric properties of data are common in applications ranging from games, to computer vision, to robotics, to bioinformatics. This labs explores a simple geometric problem — finding the two closest points in a list. These will just be values rather than x, y or x, y, z coordinates. As an example, in the list

```
L1 = [ 15.1, -12.1, 5.4, 11.8, 17.4, 4.3, 6.9 ]
```

the values 5.4 and 4.3 are the closest together. Our problem is pretty simple and we do not expect that you will have any issues writing the code itself. Instead, what we want to focus on with this lab is two important aspects of writing good code: testing and ensuring that your code runs efficiently. We have discussed how to test your solution, and how different algorithms behave as the problem size gets larger in class. This lab is a practical example for you to explore some of these issues on your own

Before you start, unzip the files given to you in this lab. You will see two files: `example_program.py` and `test_driver.py`. The second file is a test module that imports the code from `example_program.py` and tests the output of functions in that file. The files contain a quick tutorial on using `doctest` Read through both files and play with the `doctest` module until you understand how to write tests, the syntax of test conditions, how to call the testing code, and how the files work together to test the `addone()` function.

In the first two checkpoints, you will write two different solutions, one that finds the two closest values without sorting, and another that finds the values by first sorting the list. You will need to test these solutions thoroughly using your new best friend, `doctest`. To be clear, you are expected to make a copy of the `example_program.py` file to a new file called `lab10.py`. In `lab10.py`, strip out all of the code and comments related to the `addone()` function and replace it with the code for this lab. Then modify `test_driver.py` to test your new module instead of `example_program`.

Once the first two checkpoints are completed you will begin to explore the efficiency of your solutions in Checkpoint 3 by developing code to call your checkpoint 1 and 2 functions for varying input sizes from 100 element lists up to 10,000 element lists.

Checkpoint 1: Solution Based on Double For Loops

Write a function called `closest1` that takes as its only argument a list of floats (or ints) and returns a tuple containing the two closest values. If there are less than two values in the list, you should return a tuple `(None, None)`. This function should not change the list at all, but instead it should use two `for` loops over the range of indices in the list to find the closest values. If you can, try to do this without testing the same pair of values more than once.

To illustrate its usage, assuming we've assigned `L1` from above, the code

```
(x,y) = closest1(L1)
```

```
print(x, y)
```

should output

```
5.4 4.3
```

Think hard about the cases you should use to test that your code is correct. Then generate examples of these cases in the comments for `closest1` and run the testing code on them. Fix any mistakes. Also, consider writing test cases to check that you do not change the list. These cases will become useful if you later forget that you are not supposed to change the list when you are modifying the code.

To complete Checkpoint 1 Show your TA or mentor (a) your code and (b) your test cases, and (c) the results of running the test code on your module. Be prepared to explain why your tests are reasonably complete, and be prepared to demonstrate what happens when your tests fail. You might be asked to generate new test cases.

Checkpoint 2: Solution Based on Sorting

Write a function called `closest2` that takes as its only argument a list of floats (or ints) and returns a tuple containing the two closest values. You may assume there are at least two values in the list. This function should not change the list. It should, however, make a copy of the list, sort the copy, and then, in a single pass through the sorted list, decide which are the two closest values. You might have to think a bit about why this idea should work.

Once again, think through the test cases you will need. Then, generate test cases and use them to ensure your function is correct.

To complete Checkpoint 2 Show your TA or mentor (a) your code and (b) your test cases, and (c) the results of running the test code on your module. Be prepared to explain why your tests are reasonably complete, and be prepared to demonstrate what happens when your tests fail. You might be asked to generate new test cases.

Checkpoint 3: Comparative Evaluation

In this checkpoint you will evaluate the result in two ways. First, following through what we did in Lectures 20 and many other previous lectures, try to think about how many comparisons first function made? If the length of the list is N, what is (roughly) the number of comparisons as a function of N?

Assuming (correctly), that sorting makes $O(N \log N)$ comparisons, how many additional comparisons does the code you wrote for the second version make? Based on this, which function do you think is faster?

The second evaluation is to run timing experiments on lists of random values. Unlike the code in `lec20_search_smallesttwo.py` from the searching lectures which uses `random.shuffle` to randomly shuffle a sequence of integers, you will need to use the function `random.uniform` to generate your experimental sequence. For example,

```
random.uniform(0.0, 1000.0)
```

generates a single random float between 0 and 1000. Other than this, feel free to adapt and modify as much of the code we developed in class as you like.

For simplicity, you can include all code from Checkpoints 1-3 in a single py file, putting the timing code after the `if __name__ == '__main__':` line. For this checkpoint, you will not use any additional `doctest` tests. We are now assuming your code is correct and we are focusing on comparing the performance of the two variants.

Run timing experiments to compare the performance of your two solutions on random lists of length 100, 1,000, 10,000. What do you conclude about the two solutions?

To complete Checkpoint 3 Explain your analysis from the first part of this checkpoint, and then show your code and your experimental results from the second part. Explain your final conclusion.