CSE 241 Algorithms and Data Structures Fall Semester 2015

Lab 1: Finding Closest Pairs, and Why Algorithms are Cool

Assigned: 8/26/2015 *Due Date: 9/14/2015*

1 Overview

This lab has several goals:

- Ensure that you understand the closest-pair algorithms studied in class.
- Provide first-hand experience with the practical benefits of nontrivial algorithm design, in particular the divide-and-conquer methodology.
- Raise some practical issues in measuring algorithm performance.

The project URL for this lab is
`https://svn.seas.wustl.edu/repositories/yourid/cse241_fl15/close`
where "**yourid**" is the name of your 241 SVN repository.

In this lab, you will implement both the naive and divide-and-conquer closest-pair algorithms and do some performance comparisons between them. The following sections are marked to indicate roughly what percentage of the total credit for the lab may be gained by completing each section.

Please start early so that you can get help if you need it!

2 Part One: Implement the Algorithms (75%)

To complete this section, you must implement the two closest-pair algorithms discussed in class: the naive algorithm, and the divide-and-conquer algorithm. To help you get started, we have provided Java and C++ code that implements everything you will need to create a functional program, except for the algorithms themselves.

Your implementation will take as its input a set of n points. For the naive algorithm, you will receive an array of (not necessarily sorted) points. For the divide-and-conquer algorithm, the input points are provided as *two* sorted arrays. One array, `pointsByX`, contains the points sorted in nondecreasing order by X-coordinate; the other, `pointsByY`, contains the same points sorted by Y-coordinate. (The C++ implementation passes in arrays of pointers to the points to match Java's implicit reference semantics.)

Your algorithms should find the closest pair of points in the input and print their coordinates, along with the distance between them. The result should be preceded by an *algorithm specifier string* that is either `NAIVE` for the naive algorithm or `DC` for the divide-and-conquer algorithm. For example, your naive output might look like this:

```
NAIVE (2769, 3214) (3721, 5587) 2556.8404
```

If there is more than one closest pair in the input, print only one (I don't care which). Your code should print the above output *only* if the `print` argument to the function is true; otherwise, you should still run the algorithm to find a closest pair but should print nothing.

To complete this section, you must ensure that your SVN repository contains all of the following:

- Your implementations of the two closest-pair algorithms in the bodies of the appropriate provided functions.
- A brief README document describing your implementation (you need not recapitulate the divide-and-conquer algorithm) and anything else interesting you noticed while implementing the algorithms. If your implementation is buggy and you were unable to fix the bugs, please tell us what you think is wrong and give us a test case that shows the error.

  Put this document in your project repository in the same directory as your code. The file should be in PDF format (to accommodate the graph in Part 2) and should be named `README.pdf`.

We will test your implementation's correctness after it is turned in by running a suite of test cases on the code in your repository. You may not have access to all the test cases that we use for validation.

2.1 Notes and Advice on the Implementation

Very Important: the provided code assigns a globally unique sequential index to each point at the time it is created. Even points with identical coordinates (which can occur and would make a good test case) will have different indices. The Point class includes a method `isLeftOf()` that implements the following predicate:

> Given points p and q, p `isLeftOf` q if p's X-coordinate is less than that of q, or if their X-coordinates are equal *and* p has a lower index than q.

The X-ordered input array is in fact sorted so that if p `isLeftOf` q, p occurs before q (even if $p.x = q.x$). You will find this property and the `isLeftOf` predicate useful in your implementation: if you split the `pointsByX` array at a given point p^*, you can rapidly identify all points q in `pointsByY` such that q `isLeftOf` p^*, regardless of how many points share the same X-coordinate as p^*. If you don't see why `isLeftOf` matters even after trying to implement the algorithm, please ask.

We expect you to use good coding style in writing your implementation. It should be easy to read and well-commented and should not commit egregious abuses of memory, pointers, type safety, and so forth. Correct but poorly-styled implementations will lose points.

We recommend that you immediately implement and test the naive algorithm, which shouldn't take very long, then get started ASAP on the divide-and-conquer algorithm so that you have time to debug it. Don't rely on the auto-grader's test cases alone to determine whether your code is correct. Start by testing and, if necessary, debugging with your own small examples. At each step of execution, think about what should be happening, and check whether the code behaves as you expect. For example, make sure that the X-ordered and Y-ordered arrays passed to a given recursive call contain the same sets of points.

Finally, it may be easier to first produce an implementation that just prints the distance between the closest pair, then modify it to print the points as well.

2.2 Notes on the Provided Code

The driver program takes a single argument. If you specify an argument of the form '@n', where n is a number, then the program generates a set of n randomly chosen points as its input. For example, specifying '@4000' generates a set of 4000 points.

Any other argument is assumed to be the name of a point file, which should have the following format. The first line of the file contains the number of points to read. Each subsequent line contains a single point's X- and Y-coordinates. For example, a valid input file would be

```
3
100 200
57 69
33 999
```

You can use this file-reading facility to debug with small examples of your own choosing. In Java, the point-reading function is `PointReader.readXYPoints`, which takes a filename and returns an array of `XYPoints`. In C++, the function is `readXYPoints`, which takes a filename and an `int *`, returns an array of pointers to Points, and sets the pointed-to int to the length of the returned array.

3 Part Two: Do the Comparison (20%)

The following two studies look at some aspects of the performance of the naive and divide-and-conquer closest pair algorithms.

Once you have working implementations of the naive and divide-and-conquer closest-pair algorithms, you should first compare their running times on inputs of the following sizes: 10000, 20000, 40000, 60000, 80000, 100000. You should produce and turn in a *single graph* plotting the running times of both algorithm versus input size. If your naive implementation takes more than about ten minutes for a given size n, you need not plot its running time for larger input sizes. Java users should run your experiments on a relatively unloaded machine to get consistent results, since the Java timer class measures wall-clock rather than CPU time.

The portion of the code to time for each algorithm is the block between the calls to start and stop the timer, as shown in the `Main` source file. When doing timings, be sure to **turn off any printing of output** in your algorithms by setting the `print` argument to "false" in the calls in `Main`. Printing is expensive and is not part of the actual algorithmic cost.

To motivate our second study, we note that "the running time of algorithm A on inputs of size n" is not well-defined, since not all inputs will take an identical amount of time to process in the divide-and-conquer algorithm. One could ask, how much variation in time is there for some distribution of random inputs to A? To investigate this question for our random input generator, modify the code in the driver to time the divide-and-conquer algorithm on at least 100 randomly generated inputs of size 50000 and report the average, minimum, and maximum running times you see with this algorithm. If you know how to compute a 95% confidence interval, you should report that as well (if not, don't worry about it). Now time 100 runs with the *same* input of size 50000 and again report the average, minimum, and maximum running times. Variations in time to run on the same input can be attributed to external factors. How large is the variation from one input to the next, compared to that on the same input?

Coding advice: if you need to do multiple timings on the same set of points, simply wrap this block and associated timing functions in a loop. If you need to do multiple timings on *different* sets of points, call `genPointsAtRandom()` once at the beginning of each loop iteration to generate a new set of points. (If using C++, be sure to delete these points and the array that holds them at the end of each loop iteration!) In each case, you should do all the timings within one call to your program and either save a list of results or compute a running min/max/average/variance at the bottom of the loop.

Any conclusions from your experiments, including your graph(s), should be reported as part of your README document.

4 Part Three: Find the Crossover Point (5%)

There's a good chance that, for sufficiently small input sizes, the naive algorithm may actually run *faster* than the divide-and-conquer algorithm. Why might this occur?

Using your implementations from Part One, estimate as best you can the "crossover" input size where the divide-and-conquer method first starts to consistently outperform the naive algorithm. Look for the size at which the divide-and-conquer algorithm starts to win *consistently* and continues to do so for all larger sizes. Don't just look at very small sizes, as small variations in the input could cause either algorithm to win at these sizes.

Because the running times for small sizes are so quick, you will need to modify the driver to run each algorithm in a loop for k trials, where k is a big number (at least 1000, maybe more) and divide the total time to complete the loop by k to estimate the average time per trial. To improve your accuracy in estimating the running time, choose a big enough k that the total running time of all k iterations is at least 5 seconds.

At each input size you test, you should average the running time (esimated as above) of at least 1000 random inputs to reduce the uncertainty in running time due to system and input variation. For this section, turn in a brief description of the tests you did and a summary of your running time results and conclusions. Graphs of running times showing the crossover would be nice but are not required. Add this information to your README document.