CSE 241 Algorithms and Data Structures **Lab 1**

Finding Closest Pairs, and Why Algorithms are Cool

Assigned: 9/5/2012 *Due Date: 10/1/2012*

1 Overview

This lab has several goals:

- Ensure that you understand the closest-pair algorithms studied in class.
- Give a first-hand understanding of the practical benefits of nontrivial algorithm design, in particular the divide-and-conquer methodology, and raise some practical issues in measuring algorithm performance.
- Get you to set up and familiarize yourself with the working environment in which you'll be doing the rest of your 241 labs.

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!

Detailed instructions on how to obtain the provided code for your labs and how to submit them for grading may be found on the Labs page of the course website.

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. You may write either a C++ or a Java implementation. To help you get started, we have provided C++ and Java code that implements everything you will need to create a functional program, except for the algorithms themselves.

Your implementation will take as its argument 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.)

Given the input points, your algorithms should find the closest pair of points in the input and print their coordinates, along with the distance between them, like this:

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

If there is more than one closest pair in the input, print only one. **To make your TAs' lives easier, please print the point with lowest X-coordinate first; if the two points have equal X-coordinates, print the point with lowest Y-coordinate first.**

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. Please note that we may *replace* the provided `Lab1.java` / `lab1.cc` file to test your code, so any changes you make to this file will be ignored. All other files will be built and run as you submit them.

- A correctly edited *control file* that specifies the implementation language you used and indicates that the lab is ready for grading. The file `lab1/control.txt` is the control file for this lab; see that file and the Labs page of the course website for editing instructions.

- At least three *test cases* (input files of your own design) designed to illustrate the correctness of your implementation. For these test files, think about how a closest-pair implementation can go wrong; for example, what happens when two input points have the same coordinates? Your test cases should show that you've thought about possible pitfalls and avoided them.

  Put your test cases in the `lab1/test-cases` subdirectory of the repository and name them `case1.txt`, `case2.txt`, and `case3.txt`. Please explain the design of these test cases in your `README` file (see below).

- A brief document describing your implementation (you need not recapitulate the divide-and-conquer algorithm), how you chose your test cases, 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. You'll lose fewer points for a wrong answer if you indicate that you know it's wrong (and, if possible, why it's wrong).

  Put this document in the `lab1` subdirectory of your repository. It should be named `README`, with a file extension matching its format. The file may be plain text (`README.txt`), Word (`README.doc` or `README.docx`), or PDF (`README.pdf`).

We will test your implementation's correctness after it is turned in by running a suite of test cases of our own design 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 come talk to me or to your TAs.

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.

As the requirement for test inputs indicates, we expect you to test your implementation's correctness. It is *not* sufficient just to produce code that runs without crashing! 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. Start by testing and, if necessary, debugging with 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. If you're implementing the algorithm in Java, there is a Plotter class in the provided code that can graphically display the points in an array to help you debug.

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 Java wrapper is known to work with OpenJDK 6, which is the Java version on the CEC Linux machines. The C++ wrapper was tested with GNU C++ 4.1.2 and 4.6.0 but should be portable to many other compilers. Although we have tried not to introduce any nonportable code in the wrappers, *the official C++ compiler for this course is GNU C++.* Any C++ code you turn in must work with this compiler (but feel free to write standard-conforming code, even if it only works properly with gcc 4.1 or later). The CEC compute server, "shell," has g++ 4.6.0, which is in your path by default.

Both wrapper programs are meant to be controlled from the command line. They take a single argument, which is either a file name or a number of points prefixed with the '@' character.

If a point file is specified, it 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 read your test cases, and to debug with other small examples of your own choosing. In Java, the point reading function is `PointReader.readXYPoints`, which takes a filename and returns an array of `XYPoint`s. 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.

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.

The C++ wrapper expects exactly one argument. The Java wrapper can optionally take *two* arguments, in which case the first argument is the name of a *transcript file* that will contain a copy of everything written to the terminal during program execution. The second argument provides the input specification, which is treated as described above. If you're using Java from within Eclipse or some other development environment and can't easily provide command-line arguments to your program, feel free to modify and recompile `Lab1.java` to hard-code input arguments for your own testing (but remember that any such changes will be lost when you turn in the lab for grading).

If you are using the C++ wrapper and want a transcript file similar to that produced by the Java version, you can either cut and paste the output into a file or use the UNIX `script` facility to capture the session (type `man script` at the prompt for details).

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: 1000, 2000, 4000, 8000, 16000, 32000, and 64000, and 128000. 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. Be sure to **turn off any printing of output** in your algorithm before doing timings, as 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 answer this question for our random input generator, time at least 100 randomly generated inputs of size 128000 and report the average, minimum, and maximum running times you see with the divide-and-conquer 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 do 100 runs with the *same* input of size 128000 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?

Any conclusions from your experiments should be reported in your README document. If you are submitting a plain text README, you may include the graph in a separate file in PDF or Excel format, as long as you indicate the filename of the graph in your README. Otherwise, please include the graph in 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 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. You will have to edit the main driver code to add these loops. 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 10 seconds.

At each input size you test, you should average times over 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.