

Lab 1

*January 14, 2003**Due Date: January 28 (early date January 21)*

1 Overview

This lab has several goals:

- Ensure that you understand the closest-pair algorithms presented in class.
- Get a first-hand understanding of the practical benefits of nontrivial algorithm design, in particular the divide-and-conquer methodology, and think about some practical issues in algorithm performance.
- Set up and familiarize yourself with the CEC 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! The recommendation is that you complete and submit at least the naive algorithm and the dividing part of the algorithm by Tuesday January 21.

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 functional programs except for the algorithms themselves. The code is available from CS 241 course web page (<http://classes.cec.wustl.edu/~cs241>). You may change it in any way you like so long as you can produce the output described below. As a further breakdown, 5% is given to implement the naive algorithm, 35% is given to implement the dividing portion of the divide-and-conquer algorithm, and 35% is given for the rest of the divide-and-conquer algorithm.

Your implementation will take as arguments a set of n points, 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 these lists, you should find the closest pair of points and print their coordinates, along with the distance between them, like this:

(2769, 3214) (3721, 5587) 2556.8404

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.

You must turn in the following items to complete this section:

- Your implementation of two closest-pair algorithms, including any files you wrote or modified. Please *do NOT* submit any provided code that you didn't modify.
- A transcript of a shell session showing the output of your program on random inputs (generated using the seed "2003" specified by the provided code) for the following sizes: $n = 10, n = 20, n = 50, n = 100, n = 250, n = 750, n = 1500, n = 3000$, and $n = 6000$. If you just implement the splitting portion of the divide-and-conquer algorithm then for $n = 10$ and $n = 20$ output the input points (sorted by x -coordinate) and then the points in both left and right halves both sorted by x -coordinate and sorted by y -coordinate.
- 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).

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 `leftof()` that implements the following predicate:

Given points p and q , p `leftof` 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 `leftof` q , p occurs before q (even if $p.x = q.x$). You will find this property and the `leftof` 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 `leftof` p^* , regardless of how many points share the same X-coordinate as p^* . If you don't see why `leftof` 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

To access GNU C++ from your CEC UNIX account, you must install the `gnu` package (from the prompt, say `pkgadd gnu`) if it is not already installed. Both wrapper programs include the ability to read a list of points from a file with 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 `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.

The Java wrapper program takes zero, one, or two arguments. 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, if present, is assumed to be the name of a file with a list of points as described above. If no second argument is given, the user must enter the number of points to generate, and those points are generated at random. If you're using Java from within Emacs or some other development environment and can't provide command-line arguments to your program, feel free to modify and recompile the wrapper code to hard-code a particular point file name.

The C++ wrapper program takes zero or one arguments. The first argument, if present, corresponds to the second argument of the Java wrapper (a file of points). There is no built-in transcript facility in the C++ wrapper, so you must 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 (10%)

Experiment with your implementation and answer the following questions as part of the document you submit along with your code. Please clearly identify where you're answering these questions:

Once you have working implementations of the naive and divide-and-conquer closest-pair algorithms, compare their running times on inputs of sizes between 25 and 6525 (at intervals of 250). 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.

Note that the naive algorithm doesn't need a sorted input, so don't spend the time to sort before calling it – doing so needlessly makes it look slower compared to the divide-and-conquer algorithm.

Note also 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. For this lab, we're mostly going to ignore this fact – just use one randomly generated problem for each of the input sizes above. However, I'd also like you to time at least 100 randomly generated inputs of size $n = 1000$ 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). Be sure to time only the closest-pair algorithm, not the random generation of points. How imprecise are we being by not averaging over many inputs for every size tested?

4 Part Three: Explore the Crossover (5%)

Experiment with your implementation and answer the following questions as part of the document you submit along with your code. Please clearly identify where you're answering these questions:

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 Two, identify the crossover point where the divide-and-conquer method starts to outperform the naive algorithm. If the crossover point is nontrivial ($n > 3$), describe how to modify your divide-and-conquer implementation so that it calls the naive algorithm internally when doing so would be faster than making another recursive call. Does the resulting “hybrid” implementation run faster than a pure divide-and-conquer approach?

Hint: for small inputs, you will have a problem getting accurate running time estimates to find the crossover, since the running time for each input is so small. To get better estimates, time a large number N of randomly generated inputs – enough so that the total running time for all of them is at least 5 seconds – and then divide by N to get an average running time.

5 Extra Credit: Change the Crossover (10%)

Try to implement the best heuristics you can think of to speed up the naive implementation. For example, is it really necessary to compute $n(n - 1)/2$ pairwise distances for every input? Do your heuristics change the crossover point where the naive algorithm starts to lose to the divide-and-conquer algorithm for most inputs?

For this section, turn in your modified naive implementation, a brief description of the changes you made, and an indication of whether/how much the crossover point moved.