

Lab 1

*September 6, 2005**Due Date: September 20 (early date: September 13)*

1 Overview

This lab has several goals:

- Ensure that you understand the divide-and-conquer closest-pair algorithm presented in class.
- Get a hands-on understanding of the practical benefits of designing more sophisticated algorithms, in particular, the divide-and-conquer design technique.
- Design or modify one of the algorithms we discussed and compare its performance to the brute force and divide-and-conquer algorithms.

The following sections describe what you are to do, how much each part contributes to your grade for Lab 1, and finally what you should submit.

Please start early so that you can get help if you need it! The recommendation is that you complete and submit the brute force algorithm and the dividing part of the divide-and-conquer algorithm by September 13.

2 Part One: Implement the Algorithms (70 points)

To complete this section, you must implement the two closest-pair algorithms discussed in class: the brute force 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 CSE 241 course web page (<http://classes.cec.wustl.edu/~cse241>). You may change it in any way you like so long as you can produce the output described below. As a further breakdown, 10 points are for implementing the brute force (and getting familiar with the provided code), 25 points are for implementing the dividing portion of the divide-and-conquer algorithm, and 35 points are for successfully completing the rest of the divide-and-conquer algorithm. Your implementation will take as arguments a set of n points. (The C++ implementation passes in an array 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:

For $n = 10$, the elapsed time is 10 milliseconds.

A closest pair of points is (13,4) and (11,5) with distance 2.23606797749979

Although the data sets that will be generated for you are already sorted by x -coordinate, the provided start for the divide-and-conquer algorithm does not assume the data is already sorted and thus is set up to sort the points by both x and y -coordinates. Please leave this in so the running time includes that for the divide-and-conquer algorithm. For the brute force algorithm you should not be sorting the points.

Finally, you will be asked to show results for both the brute force and divide-and-conquer algorithms for inputs of size 4, 10, 25, 275, and 1000. The point sets that you should obtain (if using Java) are provided on the course web page. If the random data sets you get don't match, then please instead use the provided data sets. (See Section 2.2 for how you can easily do this.)

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.

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 brute force algorithm, which shouldn't take very long, then get started 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 they are wrong then your program won't work and it will be a lot easier to debug by first checking you have correctly created the point arrays for the subproblems and only then making the recursive calls. If you are using Java, there is a `Plotter` class in the provided code that can graphically display the points in an array to help you debug.

2.2 Notes on the Provided Code

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 needed lines are included within the comments of the provided code.

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: Compare the Algorithms' Performances (10 points)

Once you have working implementations of the brute force and divide-and-conquer closest-pair algorithms, compare their running times on inputs of different sizes. Start with $n = 25$ and increase n in increments of 500 until you reach an input size where the brute force algorithm takes around 10 minutes. I'm expecting that once you determine how big to make n that you'll modify the driver to include a loop to run the algorithm for the desired values of n . There is no need to submit your modified driver. Run both algorithms on each of the inputs separately timing each one. Note that the brute force 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. 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. All you will need to save from these runs is the value of n and the running time of each algorithm for all input sizes used.

Finally, determine how many points you can include in your input before the divide-and-conquer algorithm runs for more than a minute.

4 Part Three: Try Something New (20 points)

Here is your chance to implement an algorithm of your choice and see how it compares to the brute force and divide-and-conquer algorithm. What you try need not be faster than the divide-and-conquer algorithm. Perhaps you had an idea for an algorithm that you'd like to try or heard a suggestion in class that you want to use. Ideally you'd pick something that you think might run fast in practice and try it out. If you want to modify the method used to generate the random data you may, but also include test results using the provided method for generating the data. We will associate points with how interesting your new algorithm is and how much additional coding is needed. For example, modifying the divide-and-conquer algorithm to call the sort algorithm each call to sort the points by y -coordinate is not very interesting and also a very simple change. So if this is what you did then you would probably only get about 5 points here. However, you don't need to do anything particularly complex. Just find something that is interesting to try (and tell us why you think it's interesting) and you should be fine. You can always ask Dr. Goldman if you aren't certain about what is expected.

5 What to Submit

You must submit your code electronically (see the "Labs" portion of the CSE 241 homepage). Your electronic submission should include all files needed to compile your program (including any provided files). You will also submit a hard copy. For the hardcopy, please include the following

items, *in the listed order*. If you submitted portions on the early date staple that to the back of what you submit on the due date.

- A signed cover sheet with your name legibly written on it.
- Any code that you wrote or modified. Please *do NOT* submit any provided code that you didn't modify. Also, if you modified a small portion of some provided code, just submit the portion you modified with the changes you made highlighted or clearly marked in some way. If you just modify the driver to include a loop to call your `closestPair` algorithm for various number of points or change a variable name, there is no need to submit the modified driver. Finally, please try to print your code in two column format (landscape mode) to reduce paper usage.
- If your implementation is buggy and you were unable to fix the bugs, please let us know what you think is wrong and give us output from a test case that shows the error. You'll lose fewer points for indicating there is a problem than you would if we discovered it ourselves.
- A transcript of a session showing the output of your program on separate runs of the original provided Lab1 driver for both the brute force algorithm for inputs of size 4, 10, 25, 275 and 1000. Please combine all outputs into a single file. If the points you obtain from the random point generator do not match those shown on the course web page, then please show the output using the data sets on the course web page (using the `PointReader` to input them). If you decide to create a loop in the driver to try those particular 5 values that is fine but you will need to reseed the random generator using the below line before each data set is generated.

```
java.util.Random randseq = new java.util.Random((long) 3.14 * n);
```

- A transcript of a session showing the output of your program on separate runs of the original provided Lab1 driver for both the divide-and-conquer algorithm for inputs of size 4, 10, 25, 275 and 1000. Please combine all outputs into a single file. IF you just implemented the dividing portion of the divide-and-conquer algorithm (for submitting on January 27) then instead of doing the above, for $n = 10$ output the points in both the left and right halves of the top-level recursive calls sorted by x-coordinate and sorted by y-coordinate. You should check if your output is correct and if not indicate what it should be.
- For Part 2 you should include a single graph created using Excel or some other plotting program of your choice that shows the performance of both the brute force and divide-and-conquer algorithm. Your graph should label the axes, include a key, and so on. There is no need for you to submit the answers output – just the graph is needed. Briefly indicate if your graph looked as expected and if not, try to explain why. Also, tell us how large you were able to make n before the divide-and-conquer algorithm took over a minute.
- For Part 3 you should include a clear and concise description of the algorithm you implemented. If it is not obvious that your algorithm will always output the right answer convince us why it will. You can just replace the graph discussed above for Part 2 by a graph that has three plots (one for each algorithm) instead of one with just two plots. Finally, briefly discuss how fast your algorithm was as compared with the brute force and divide-and-conquer algorithms and why you think this performance occurred. If you did some additional experiments to answer this question feel free to show those (but don't get carried away – one or two sheets of paper for Part 3 should suffice).