

Lab 1

January 17, 2006

Due Date: January 31 (early date: January 24)

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 your own algorithm to solve the problem and compare its performance to the divide-and-conquer algorithms.

We have provided Java and C++ code that implement everything needed except for the algorithms themselves including the pre-processing for the divide-and-conquer algorithm. The code is available on the course home page, <http://classes.cec.wustl.edu/~cse241>, under Labs.

The following sections describe what you are to do, how much each part contributes to your grade for Lab 1, and what you should submit. **Please start early so that you can get help if you need it!** The recommendation is that you complete Parts I and II by the early submission deadline of January 24. We expect you to use good coding style in writing your implementation. Your code should be easy to read and appropriately commented. You 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!

2 The Parts of the Lab

Part One: Implement the Naive Algorithm (10 points)

You are to implement the naive algorithm that computes the distance between each pair of points (just once) and then returns a closest pair of points. You are welcome to modify the `PointPair` class implementation to provide any further functionality that you would like.

Part Two: Implement the Dividing Portion (25 points)

You are to implementing the portion of the divide-and-conquer algorithm prior to when the recursive calls are made. This must run in time proportional to the number of points. To check that your implementation is correct, for $n = 10$ you should check that the four arrays of points that will be sent to the recursive call (the left and right halves sorted by both x - and y -coordinates) are correct.

Very Important: the provided code assigns a globally unique sequential index to each point at the time it is allocated. The `XYPoint` class includes a method `leftOf` where for 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 `pointsByX` array is sorted so that `pointsByX[i].leftOf(pointsByX[i+1])` even if these two points have the same x -coordinate. If you are not using `leftOf` to do this part, then come talk to me or one of the TAs and discuss your implementation since there is most likely a problem when points share the same x -coordinate.

Part Three: Implement the Divide-and-Conquer Algorithm (35 points)

You are to complete the implementation of the divide-and-conquer algorithm. You should compare the results with that of the naive algorithm. Note that there is a unique distance between the closest pair of points but there could be more than one set of pairs with that distance. Determine roughly how many points you can include in your input before the divide-and-conquer algorithm runs for more than a minute. For those using Java you can increase the memory available using the command line arguments `-Xms150m -Xmx150m`. This should work up until about 2,500,000 points. If you want you can use a larger value (such as 200m, 300m, ...) up until the memory you have on your computer. If you are using Eclipse, under the Run menu pick "run..." and then put this under the VM arguments under the arguments tab.

Part Four: Compare the Algorithms' Performances (10 points)

Once you have working implementations of the naive and divide-and-conquer closest-pair algorithms, compare their running times on inputs of different sizes. You should include at least 10 different choices for n (the number of points) divided evenly between $n = 25$ and a value for n for which the brute force algorithm takes at least a minute. I'd expect you to 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. 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** from within your closest pair algorithms before doing timings, as printing is expensive. You should create a single graph that plots the running time of both algorithms as a function of the number of points.

Part Five: Try Something New (20 points)

Here is your chance to implement an algorithm of your choice and see how it compares to the divide-and-conquer algorithm. Pick something you think might run fast in practice and try it out. If you want to modify the method used to generate the random points you may, but also include test results using the provided methods for generating the points. We will associate 10 points with the creativity in designing your algorithm and how much additional coding and/or testing you performed. You can always ask Dr. Goldman if you aren't certain about what is expected.

3 What to Submit

Please submit the following items, *in the listed order*. If you submitted portions on the early date staple those grade portions behind 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 (excluding the driver which you will most likely modify to produce the requested output). 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. 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 using the original provided Lab1 driver* for inputs of size 10, 25, 275 and 2500. If you are not submitting Part 3 then for creating this transcript you should comment out the portion of the code in the driver that calls the divide-and-conquer algorithm. Please combine all outputs into a single file.

For Part 3 along with including the output as discussed above, report how large you were able to make n before the divide-and-conquer algorithm took over a minute. If you are running out of memory, you can use the virtual machine arguments as discussed in the assignment.

- If completed Part 2 (but not Part 3), then 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 4 you should include a single graph created using Excel or some graphing program that shows the performance of the naive and divide-and-conquer algorithms as a function of n . You 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.
- For Part 5 you should include a clear and concise description of the algorithm you implemented, and why you thought it would work well. If it is not obvious that your algorithm will always output the right answer convince us why it will. Create a graph similar to that discussed in Part 4 that compares your algorithm with the divide and conquer algorithm. Your plot should include n large enough that the slower of the two algorithms takes at least a minute. Briefly discuss how fast your algorithm was as compared with divide-and-conquer algorithm and why you think this performance occurred.