Data Structures and Algorithms (CS 130b) **Prof. Suri**

Programming Assignment 1

Handed Out: Jan 6 *Due: Jan 22, midnight*

1 Overview

Your first programming assignment is to implement the divide-and-conquer algorithm for determining the **closest pair** of points among a set of n points in the plane. (In addition to my notes for Lecture 1, this algorithm is also described in the textbook, in Section 10.2.2.)

The divide-and-conquer algorithm works by dividing the point set into two nearly-equal halves, by splitting the set at the median x-coordinate. Let x_{mid} be the x-coordinate of the dividing line separating the two point sets. We recursively compute the closest pair in each half, and then use them to solve the merge step. More specifically, suppose d_1 and d_2, respectively, are the closest pair distances for the left and right subproblems, and let $d = \min\{d_1, d_2\}$.

Determine the set of points whose x-coordinates lie in the range $[x_{mid} - d, x_{mid} + d]$. Call this set M, for the middle (or, strip, as we called it in class). We order the points of M in ascending order of their y-coordinates. We find the closest pair of points in M by comparing each point p to only those whose y-coordinate differ from p by at most d. We return smaller of d and the closest pair distance in M.

In order to receive full credit for the assignment, you must do the following.

The $O(n^2)$ algorithm (10 pts)

First program the naive $O(n^2)$ time algorithm, which computes the closest pair by comparing the distances among all $\binom{n}{2}$ pairs. This algorithm will also be useful for checking the correctness of your divide-and-conquer algorithm.

The basic divide and conquer (40 pts)

Implement the divide-and-conquer scheme for computing the closest pair. In the basic version, you may resort to doing a **sort each time** you have to either divide the point set, or order the points in M by y-coordinate. The running time of this program obeys the following recurrence

$$T(n) \;=\; 2T(n/2) + O(n \log n),$$

which solves to $T(n) = O(n(\log n)^2)$. While not optimal, this should still run significantly faster than the $O(n^2)$ scheme, even for moderate values of n.

The optimal divide-and-conquer (30 pts)

Finally, eliminate the repeated sorting step by pre-sorting the points **just once by x-coordinates and once by y-coordinates** in the beginning. All other reordering operations during the algorithm can be performed by copying from these master sorted lists. With sorting occurring only twice in the beginning, this algorithm will have the optimal asymptotic complexity of $O(n \log n)$.

Input-Output Formats

- Each point is a pair of real numbers (x, y).
- You should make sure that the input does not contain duplicate points (which may give closest pair distance of 0). Remove all duplicates, keeping only one copy of each point. (You can assume that two points are the same if their x and y-coordinates match up to 7 decimal places.)
- Points with the same x-coordinate will be sorted by their y-coordinate in ascending order i.e. smaller y values first. Likewise, points with the same y-coordinate are sorted by their x-coordinate.
- The program output should list the "closest pair distance" on the first line, followed by the pair of points "(x1, y1) (x2, y2)" with the closest pair distance on the second line. The pair must be sorted (see earlier bullet), and decimal valued numbers should be rounded to the 7th decimal place.
- If more than one pair of points achieves the closest pair distance, then **output all such pairs**. Allow a tolerance of 7 decimal places for distances to be equal. For example, pair distances of 10.123456711 and 10.123456733 will be treated as equal. The pairs must be sorted according to the first point in the pair, then the second point. Again, the points in a pair are sorted by x-coordinates first, then y-coordinates.

2 Testing and Empirical Analysis

[80 pts] Testing by TAs

You are required to include a Makefile with your source code such that entering `make` produces an executable named `closestPair`. There should only be one executable for all 3 algorithms, and it will accept 1 command line argument to determine which algorithm to execute. It must work on the CSIL machines and shall have the following usage:

```
closestPair [brute|basic|optimal]
```

where the options "brute", "basic", and "optimal" refer to your $O(n^2)$, $O(n \log^2 n)$ and $O(n \log n)$ implementations, respectively.

Your program shall accept input one line at a time through `stdin` until it encounters EOF (Ctrl-D), and parse each line as two numbers separated by whitespace, with the first

number treated as the x-coordinate. If bad input is supplied, your program shall ignore it. For example,

```
csil$ ./closestPair brute
1.0232 102.12309
0.232 -92.293
asdf
-100      0
10.7832 20.452
0 0
1 2 3
```

will discard the 3rd and last lines, then run your $O(n^2)$ implementation on 5 points. If fewer than two points are given, your program will produce an error message and exit.

The main thing the TA will test is **correctness;** that is, do all three algorithms, namely, the $O(n^2)$, the $O(n(\log n)^2)$ and the $O(n \log n)$ time algorithms, give the same and the correct output? The TA will also check how your program handles the **degeneracy** of input (duplicate points).

In addition to testing by TAs, I want you to do an extensive testing and scalability analysis using random inputs, as described below.

Random Input Generation.

Write a program `randData` that accepts a command line argument n and returns a list of n random points. To do this, you may use the *rand()* function in the following formula

```
min + (max - min)*(double)rand() / RAND_MAX
```

to help generate a random number between `min` $= -10,000$ and `max` $= 10,000$. It is recommended you include the statement *srand(time(NULL))* before using *rand()* to generate different sequences of numbers on each invocation of `randData`.

Lastly, the output format of `randData` should match the input format of `closestPair`.

```
Example:
csil$ ./randData 5
2067.36 -9846.11
3537.35 -9972.65
-7114.22 4474.7
4227.94 7224.47
-3844.4 2419.46
```

[20 pts] Scalability.

Show how well these algorithms scale with the problem size by plotting the runtimes of your algorithms vs. increasing values of n. For each n, generate the data set using the random generation method described earlier. I suggest using $n = 10^2, 10^3, 10^4, 10^5$, and perhaps even 10^6 and 10^7 on your divide-and-conquer solutions to do the scalability analysis.

If you've done the above, you can enter

```
csil$ time ./randData [n] | ./closestPair [algorithm]
```

for different values of n to gather your runtime data.

Discuss the growth curves of the 3 algorithms. Do they match the theoretical bounds of the algorithms? If not, comment on what could be causing the anomaly.

This is the only non-code portion of the assignment, and you must turn this in as a brief document (PDF).

Submission.

Submit via `turnin` to `prog1@cs130b` your Makefile, source code, and writeup.

Example:

```
csil$ turnin prog1@cs130b *.cpp *.h scalability.pdf
```

Examples.

```
csil$ ./closestPair brute
1.0232 102.12309
0.232 -92.293
asdf
-100      0
10.7832 20.452
0 0
1 2 3
closest pair distance: 23.120591
(0, 0) (10.7832, 20.452)

csil$ ./closestPair basic < testInput01.txt
closest pair distance: 4.1231056
(9810, 748) (9811, 752)

csil$ ./randData 10000 | ./closestPair optimal
closest pair distance: 3.9492404
(6629.49, 9401.85) (6633.43, 9402.12)

csil$ time ./closestPair optimal < testInput02.txt
closest pair distance: 0.01
(4146.43, 4555.89) (4146.44, 4555.89)
(4676.11, 4149.56) (4676.11, 4149.57)
(6054.55, 8227.33) (6054.56, 8227.33)

real 0m0.286s
user 0m0.280s
sys 0m0.004s
```