CS325 programming assignment 1

Electronic submission of Code and Report to TEACH
at 11:59PM Feb 2nd

January 20, 2017

TAs in charge Juneki Hong (junekihong@gmail.com) Juan Liu (liujua@oregonstate.edu)

Hand in Instructions Code should be submitted through the TEACH website. Students are encouraged to work in groups of up to 3 people. Each group only need to submit one submission, which should include the names of all group members. You can choose any of the following programming languages: `java, python, c/c++/c#`. If you wish to use a language that is outside of this list, you will need to obtain advance approval from at least one of the TAs in charge.

Closest pair of points In this assignment, you will solve the problem of finding the closest pair of points among a set of n points in the plane, which is an important computational geometry problem that has many applications in graphics, computer vision, GIS and air traffic control etc. In particular the main idea you will explore for this problem is divide and conquer, which I briefly describe below.

The divide-and-conquer algorithm works by dividing the point set into two nearly-equal halves, achieved by splitting the set at the median x-coordinate. Let x_m be the x-coordinate of the dividing line separating the two point sets.

We recursively compute the closest pair in each half, and then solve the merge step based on their results. 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 $[x_m - d, x_m + d]$. Call this set M, for the middle. 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 differs from p by at most d. Let their distance be d_m, we return $min(d, d_m)$.

For this assignment, you will implement three different algorithms for this problem.

1. Brute-force. This algorithm works simply by computing the distance between all $\binom{n}{2}$ pairs of points and finding the closest pair. This algorithm will have $O(n^2)$ run time.

2. Naive Divide and Conquer. Implement the above outlined divide-and-conquer algorithm for computing the closest pair. In this naive version, you will sort the points within M based on y-coordinates in each recursive call from scratch.

3. Enhanced divide and conquer. In this version, you will eliminate the repeated sorting by pre-sorting all the points just once based on x-coordinates and once based on y-coordinates. All other ordering operations can then be performed by copying from these master sorted lists.

Empirically testing the correctness of your algorithm. Your program should take an input text file of points and output its results to an output text file. Your program will be run by the TAs, so please leave instructions on the steps to compile/run your program in a "README.txt" file.

Suppose you are given a file "example.input", and it contains the following points:

```
0 0
5 5
9 8
8 9
1 8
1 7
9 5
4 0
9 6
9 7
```

Your program should take these and output a file "output.txt" in the following format:

```
1.0
1 7 1 8
9 5 9 6
9 6 9 7
9 7 9 8
```

The minimum distance is printed at the top, followed by all of the corresponding minimum pairs of points. In the case of ties (like in this example), you should sort the matching points in order[1]. Here, you see 4 pairs of points that happened to achieve the minimum distance of 1.0. [2]

[1]The sorting should be done by X value, then by Y value. For example, if you have 2 sorted points $(x_1, y_1), (x_2, y_2)$, then $x_1 < x_2$ with ties broken by $y_1 < y_2$. Don't worry, this should be the default sorting behavior if you call some library's array.sort() function on an array of points.

[2]For simplicity, you can assume that all data points will have distinct x and y values. This avoids complications that might arise in the median calculation.

Your output file should look like the above, and you will be given "example.input" to help test your algorithms. When it comes time to grade, the TAs will test your code using a separate file.

Make a separate runnable command for each algorithm (Brute Force, Divide and Conquer, Enhanced DnC). This could be different command-line flags, or even different programs. Mention how to run your programs in a text file called "README.txt".

So for example, we might need to run:

```
> ./bruteforce example.input
> ./divideandconquer example.input
> ./enhanceddnc example.input
```

To run the each algorithm. And then these programs could output "output_bruteforce.txt", "output_divideandconquer.txt", "output_enhanceddnc.txt" respectively.

Empirical analysis of run time. For this part, you need to generate your own random inputs (for your input, please make sure that no points have the same x value or y value) of different sizes using a random number generator and run your algorithms on these inputs to measure their empirical run time. The suggested range for input size n is $10^2, 10^3, 10^4, 10^5$. If time allows, you are encouraged to try even larger input sizes for the divide and conquer algorithms. For each size, you should generate 10 random inputs and run each algorithm on them and report the average run time over the ten runs.

Write up In addition to submitting the source code (with clear instructions for running them), you will also need to submit a brief write up, which should include the following:

- **Pseudo-code.** Please provide the Pseudo-code for each of the three algorithms.
- **Asymptotic Analysis of run time.** Please analyze the runtime for the three algorithms. In particular, please provide the recursive relation of the runtime for algorithm 2 and 3 and solve them.
- **Plotting the runtime.** Plot the empirically measured runtime of the three algorithms as a function of the input size. Your plot should have clearly labeled axes and legends.
- **Interpretation and discussion** Discuss the runtime plot. Do the growth curves match your expectation based on their theoretical bounds? Discuss and provide possible explanations for any discrepancy between the experimental runtime and the asymptotic runtime.