CSE 417
Algorithms & Computational Complexity
Assignment #4
Due: Wednesday, 2/6/19

Turnin Instructions TBD.

This assignment is part written, part programming. It all focuses on the "closest pair of points" problem from section 5.4 and lecture.

1. [10 points] The 1-dimensional closest pair of points problem is easily solved by sorting the list of points by coordinate; after sorting, the closest pair will be adjacent in this sorted list. Near the bottom of pg 226, the book outlines why this algorithm does not generalize to the 2-dimensional case. Make this more concrete by providing a family of examples (parameterized by n) of problem instances having n distinct points where the closest pair of points (all closest pairs, if there are ties) are widely separated from each other in the list of all points when sorted either by x-coordinate or by y-coordinate.

2. [10 points] In the discussion of the closest points algorithm, for simplicity, my slides assumed that distinct points never had the same x coordinate. One implicit use of this simplifying assumption occurs on slide 23, which contains a claim and its proof. The *claim* stated there is true in general, but the given *proof* relies on this additional assumption. Specifically: (a) The statement "No two points lie in the same $\delta/2$ by $\delta/2$ square" is true (as shown on slide 23) if all points have distinct x-coordinates, but may be false in the general case where that simplifying assumption is violated. Explain why it is false in general. (b) However, the *claim* stated on that slide remains true even if multiple points have the same x-coordinate. Prove this.

3. [70 points] Implement three algorithms for the "2-D closest pair of points" problem and compare their running times.

   (a) Version 1 is the naive $\Theta(n^2)$ that calculates all pairwise distances.

   (b) Version 2 is the $\Theta(n \log^2 n)$ algorithm described in lecture (approximately slide 24) that includes a "sort by y" step in the recursion.

   (c) Version 3 is either the $\Theta(n \log n)$ algorithm given in section 5.4 (which globally sorts by y), or the $\Theta(n \log n)$ version sketched in lecture (approximately slide 26, wherein each recursive call returns a list sorted by y as well as the minimum distance seen in the subproblem).

   For testing/debugging, your program should read a sequence of x-y pairs from standard input, and run each of the three algorithms on these points, printing the coordinates of, and distance between the closest pairs of points. (If several pairs tie for closest, it doesn't matter which you select.) I will later provide some specific test cases for you to run; as part of your turn-in, include outputs from your three algorithms on these specific tests. The desired input file format is simply a white-space separated list containing an even number of decimal numbers. E.g.

   ```
   -1.0 0.0   1.0 2.0   -1.0 2.0
   ```

   specifies three points: the first on the x-axis one unit left of the origin, the second at $x = 1, y = 2$, and the third at $x = -1, y = 2$. As shown, (and unlike the simplified form discussed in lecture) distinct points may share x and/or y coordinates.

   For your timing experiments, compare the runtimes of the three methods on identical problem instances generated in two ways: First, problems containing n points placed uniformly at random in the unit square. Second, n points placed uniformly along the vertical line segment between $(0, 0)$ and $(0, 1)$. (Of course, there is a better way to solve the 1-D problem, but I do *not* want you to code it. The reason for choosing this class of problems is that they represents bad cases, perhaps even worst cases, for our 2-D divide-and-conquer algorithms.) Repeat both problem types for various values of n, including ones large enough to clearly separate the three methods' run times. (To separate the $\Theta(n \log n)$ and $\Theta(n \log^2 n)$ methods, you may need large values of n; it is fine if you choose not to run the quadratic method on these large instances; it will be quite slow.) Include a scatter plot of your runtimes, with interpolated trend lines for the three asymptotic growth rates. Roughly how large must n be before the divide and conquer algorithms are faster than the naive method? Does our simple asymptotic theory adequately explain the general growth rate of your measured run times?

You may use C, C++, C#, Haskell, Java, Lisp, ML, Perl, Python, or R, but use the same language for all three versions. (Ask if you want to use some other language.) You may use built-in or publicly available libraries for sorting, random numbers, or other utility functions.

Turn-in: TBD

Extra Credit: It is quite likely that randomly placed points are *not* a worst-case input for this algorithm. What is a (nearly-) worst-case input (more precisely, a family of inputs parameterized by n)? Does it really force the algorithm to take time $\Omega(n \log n)$, or $\Omega(n \log^2 n)$, or are our upper-bound analyses actually too pessimistic?