CSE 417
Algorithms & Computational Complexity
Assignment #5 (rev. a)
Due: Friday, 2/11/22

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

1. 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) 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. In the discussion of the closest points algorithm, for simplicity, my slides assume that (*) "no two points have 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 the 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 the assumption (*) is violated. Give a counterexample illustrating why it is false in general. (Your example doesn't need more than 4–8 points. The edges of the squares sketched on slide 23 are especially important.) (b) However, the *claim* stated on that slide remains true even if multiple points have the same x-coordinate. Prove this.

3. Implement two algorithms for the "2-D closest pair of points" problem (and compare their running times; see next problem).

   (a) Version 1 is the naive $\Theta(n^2)$ algorithm 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.

   Inputs: For testing/debugging, your program should read a sequence of x-y pairs from standard input, and run each algorithm on these points, printing to standard output (aka the console) the coordinates of, and distance between the closest pair of points, among other things. (Output detail specified below. If several pairs tie for closest, it doesn't matter which you select.)

   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    2.99  0.0   0.0
    1.0  0.0 -1.0
   ```

   specifies four points: the first on the x-axis one unit left of the origin, the second at $x = 2.99, y = 0$, and the third/fourth at $x = 0, y = \pm 1$. As shown, (and unlike the simplified form discussed in lecture) distinct points may share x coordinates. "White-space" means any combination of spaces, tabs and/or newlines. Most programming languages have input routines that will make it easy to read data in this format. "Standard input" usually comes from whatever you type on the keyboard; end-of-file is usually control-D (unix or Mac) or control-Z-return (Windows). Alternatively, from the command line, you can redirect stdin to read from a file, e.g.:

   ```
   python hw5.py < points-test1.txt
   # or
   javac hw5.java && java hw5 < points-test1.txt
   ```

   ~~I will later provide~~ Here are some specific test cases; "test0" is the example input above; the other three are part of our autograder tests.

   - points-tests.zip

Outputs: Print a single line of output for each version of the algorithm, containing

(a) The algorithm version, e.g. "Version 2,"

(b) The number of points in the input, "n,"

(c) The coordinates of the closest pair of points "x1, y1, x2, y2," in lexicographic order (i.e., x1 $\leq$ x2, and if x1==x2, then in addition y1 $\leq$ y2).

(d) The distance "delta" between them, to 3 decimal places, and

(e) The time taken, in milliseconds.

Format these as comma separated values, with your version 1 line before version 2. E.g.,:

```
Version 1, 3,  -1.0, 0.0,    0.0, 1.0, 1.414, 0.010
Version 2, 3,  -1.0, 0.0,    0.0, 1.0, 1.414, 0.011
```

We will run your code on the given test cases, and others, using Gradescope's autograder setup, running commands like the examples shown above.

Again, you may use Java or Python. You may use built-in or publicly available libraries for sorting, random numbers, or other utility functions.

For this question, just turn in your *code*. (Don't turn in the test case files; we have them. Don't turn in output files; there shouldn't be any—all output goes to standard out (console).)

4. By default, your program should perform as described above. But you should also arrange it so that it can compare the runtimes of the two 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 two methods' run times.

   The "trigger" for running the timing tests instead of processing data from standard input should simply be that standard input is empty – $n == 0$ points provided via standard in. When you detect that condition, generate random instances of the kind described above for various values of n, and run both versions of the algorithm on each random instance. You may choose what values of n to try. I recommend you repeat each test for each value of n at least 3 times to get some idea of the variability in timing due to the random nature of the data. Doing all of the 2-D random tests before all of the 1-D tests will allow you to easily separate them for analysis. (Simply print a pair of output lines as described above for all instances to the console; capturing this as a .csv file should simplify the reporting requested below.)

   Include a scatter plot of your runtimes, with interpolated trend lines for the two asymptotic growth rates. Roughly how large must n be before the divide and conquer algorithm is faster than the naive (quadratic) method? Does our simple asymptotic theory adequately explain the general growth rate of your measured run times? Do the two classes of random instances look systematically different in terms of running time?

   As your answer to this question, I am expecting a brief write-up, say 1-2 pages, summarizing and discussing these findings, with the requested graph(s). Do *not* turn in your code again; that goes with problem 3.

5. **Extra Credit:** Implement a third version of the closest points algorithm and compare it to the other two as outlined above. 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). (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.) Discuss your results in the same format requested in the previous problem. In addition to this write-up, you may turn in a modifed code file here. Alternatively, if the code you submit for Q3 includes this functionality, have it turned off by default (autograder is not expecting "Version 3" lines), but make sure your write-up tells how to turn it on, e.g. via a command-line parameter, so we can run it.

Revision History:

Rev a: Link to Test Cases. — 2/10/22.