CSE 417 HW4

Zachary McNulty (zmcnulty, 1636402)

February 2019

1 Problem 1

Q: 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

Consider the set of n points:

$$\{(0,0),(1,1),(0.5,2i{+}4) \quad i \in \{1,2,...,\lfloor\frac{n-2}{2}\rfloor\},(2i{+}4,0.5) \quad i \in \{1,2,...,\lfloor\frac{n-1}{2}\rfloor\}\}$$

The distance between $(0,0)$ and any of the other points is at least 4 as the other points have y or x coordinate respectively at least 4. The distance between $(1,1)$ and any of the other points is at least 3 as the other points have y or x coordinate respectively at least 3. The distance between two points with x coordinate 0.5 is at least 2 and the same holds for those with y coordinate 0.5. $(2i+4, 0.5)$ and $(0.5, 2i+4)$ are distance at least 3.5 away simply comparing x components. Thus, clearly $(0,0)$ and $(1,1)$ are the closest pair of points. ~~However, when sorted by either x or y coordinate, $(0,0)$ is at the beginning of the list and $(1,1)$ is at the end.~~

However, when sorted by either x or y coordinate, (0,0) is at the beginning of the list and there are ~ n/2 points before (1,1) shows up.

2 Problem 2

Q: 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

a)

If all points have distinct x values, then it is possible for the line L to be found in such a way that it can divide the points approximately in half without intersecting any of the points. However, when some points have the same x coordinate, if these points happen to be near the median x-coordinate they could be placed on opposite sides of L when dividing the points in two. Since they are on opposite sides, they are not used to calculate the initial δ value. Thus, it is possible they are $< \delta/2$ apart and lie in the same square, both lying on the border of a $\delta/2 \times \delta/2$ square adjacent to L. For example, consider the set of points:

$$\{(0,0),(1,1)(1,1.01),(2,4)\}$$

Ordering these points by their x coordinate and splitting them in half, we will get $(1,1)$ and $(1,1.01)$ in separate halves. This will give a δ value of $\sqrt{2}$ for the distance between $(0,0)$ and $(1,1)$ and then $(1,1)$ and $(1,1.01)$ will clearly be less than $\delta/2$ apart, causing them to lie on the edge of the same $\delta/2 \times \delta/2$ box.

b)

The solution to the question above only works because the two points are in different parts. If two points are chosen to be in the same part, however, they must be at least δ apart, as this is the minimum distance between two points on the same side. Suppose you start at point i in the list sorted by y values. If we were considering the 8 points $\{i, i+1, i+7\}$ next in the sorted list, each point must either be to the left or to the right of L. There are two cases: either 4 points are on each side of L, or one side has at least 5 points.

case 1: at least 5 points on one side.
Since i was first in the list, all these points have y values $\geq y_i$. For each of the five points on the same side, we can make the same argument as in the slides that each one must be in its own $\delta/2 \times \delta/2$ square. This will require us to make at least 3 rows of $\delta/2 \times \delta/2$ boxes to fit the 5 points. Since each box has height

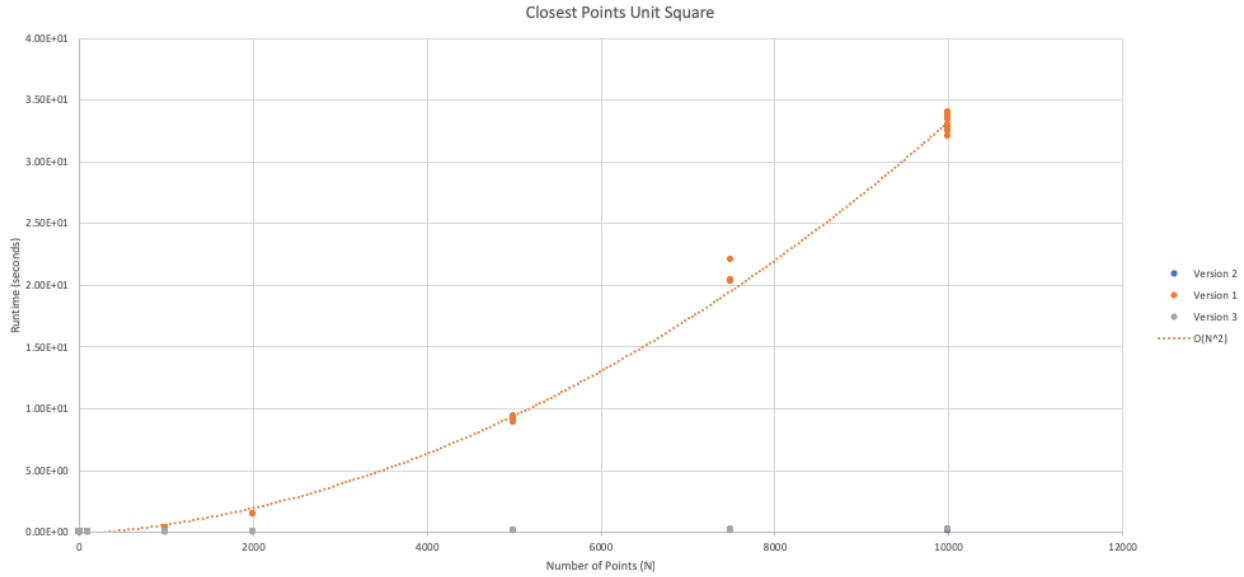
$\delta/2$, this means one of our 5 points has y value greater than $y_i + \delta$. Thus, the distance to i from this point is greater than δ. Since all points farther than this one in the list have a greater y value, there is no reason to observe them and the claim follows.

case 2: 4 points on each side
Since i was first in the list, all these points have y values $\geq y_i$. As there are 4 points on the side of i, we know each must be in its own $\delta/2 \times \delta/2$ box. The same holds for each of the four points on the other side of L. Following the same argument as in the slides, since all 8 boxes have been filled, the next point must go in a box at least 2 levels above the box of $point_i$ and thus must be greater than δ away. Thus, we do not have to compare more than the next 7 points to $point_i$.

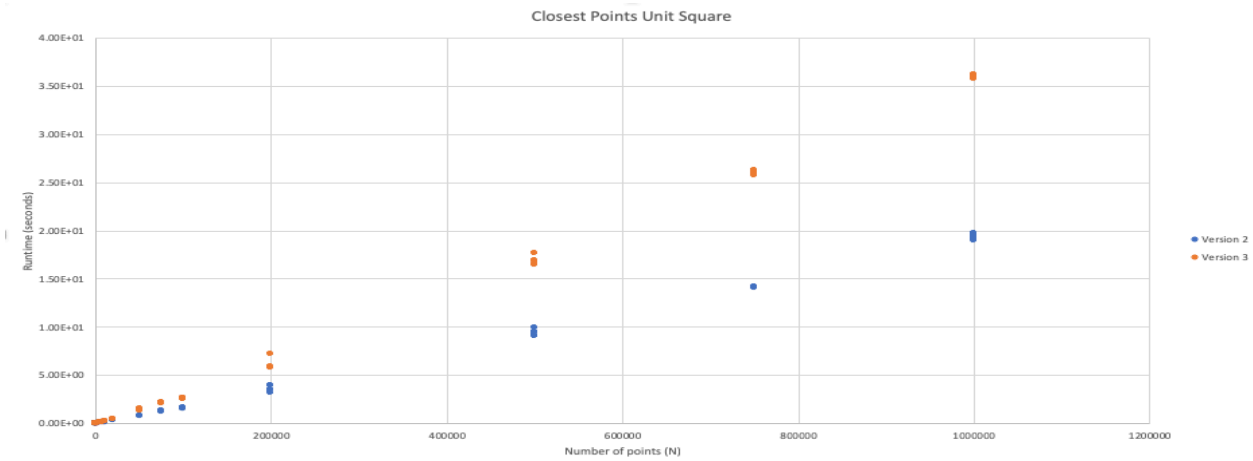
Zachary Mcnulty
zmcnulty, 1636402
CSE 417: HW4

Problem 3 Report – Algorithm Timing Analysis

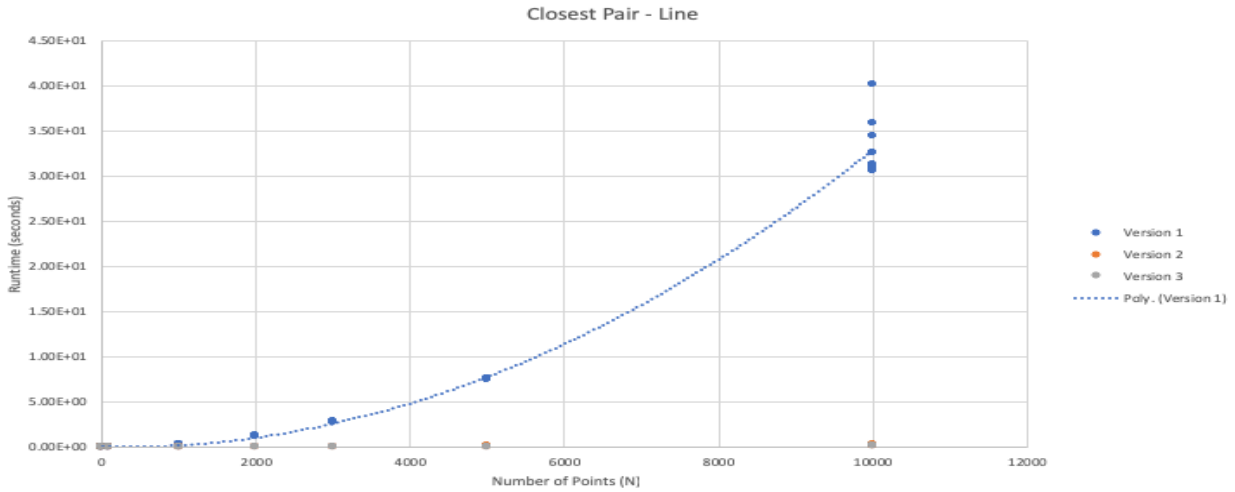
My code takes an input text file so to test the performance of these algorithms I simply generated a bunch of these test files randomly. To generate these random collections of points, I used the Python library numpy which has a convenient function for generating points uniformly at random within a given range. Using a bash script, I could easily generate and run many of these test files across a range of input sizes. For these timing trials, I considered two main cases: a set of n points distributed uniformly throughout the unit square and a set of n points distributed uniformly along the line between (0,0) and (0,1). Below, Version 1 refers to our naïve $O(N^2)$ algorithm, Version 2 to our $O(N \log (N)^2)$ algorithm, and Version 3 to our O(N log N) algorithm. We will start with analyzing the case where points lie in the unit square.



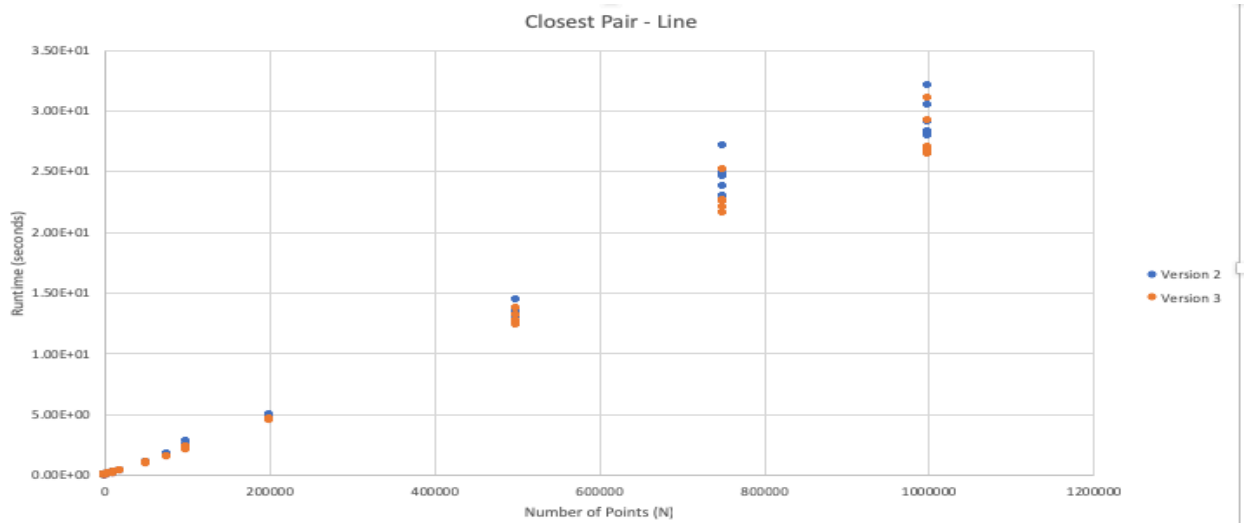
Above plots the performance of all three algorithms. We can see that the naïve "check all possible distances" algorithm is quickly outperformed by the other two. After there are more than ~2000 points, there is a clear separation between the naïve algorithm and the divide/conquer algorithms. Plotted above is also an $O(N^2)$ trend line to verify the complexity of our naïve algorithm. As expected, it clearly belongs in this complexity class and the theoretical analysis is pretty easy: there are n choose 2 = (n)(n-1)/2 distances to compare and all these comparisons are constant time, leading to an $O(N^2)$ complexity. Now to compare our O(N log(N)) and $O(N \log (N)^2)$ algorithms.



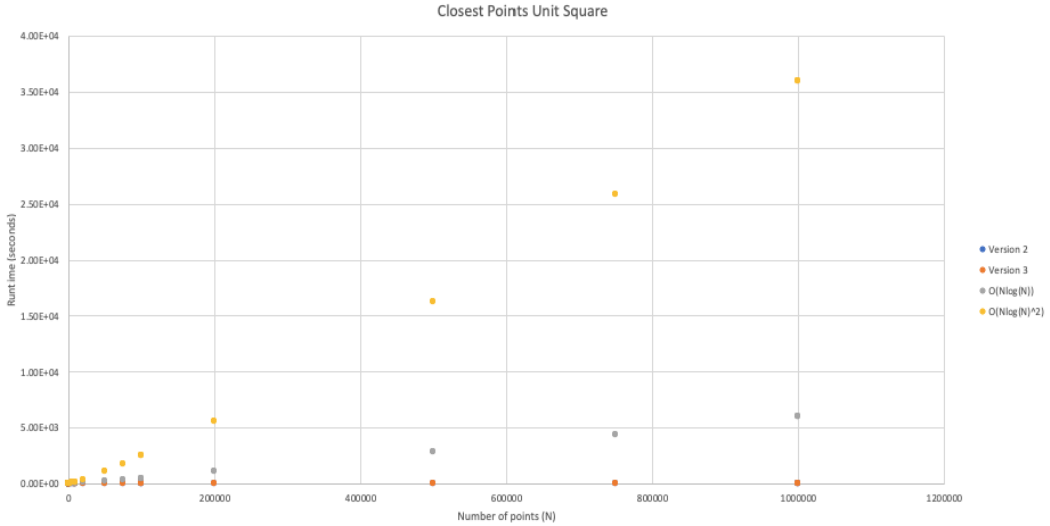
Above we can see that, despite being in a greater complexity class, the Version 2 algorithm outperforms Version 3 when the points are uniformly placed within the unit square. One reason this might be the case is that as we have it implemented the Version 3 algorithm requires sorting the entire list of points by their y coordinates. Because we are maintaining this list of points sorted by y to pass up the recursive chain (as discussed in lecture), we are forced to sort the entire list. Even though we can do this in O(N) time by merging the sorted subset of points from the two recursive cases, it is still a significant time expenditure. On the other hand, Version 2 is not maintaining these lists of points sorted by their y coordinate, so it can filter out points that are greater than delta away from L BEFORE having to sort by y. If this filtering takes out a large fraction of the points, the O(N log N) sorting may not be as costly as expected. In fact, if the filtering is significant this sorting may perform faster than the O(N) merge sorting of the entire list as we do in Version 3. If this is the case, we would expect Version 2 to slow down significantly as there are more points closer to L. Below, in the case where all points lie on a line, we will see this is the case.



With all the points on a vertical line, we can still see that the naïve algorithm performs very poorly relative to the other two. Again, by around ~2000 points, the algorithm is clearly worse off than the others.



Interestingly enough, Version 2 now performs slightly worse than Version 3 when all the points lie on the same vertical line. As we suggested earlier, the filtering factor in Version 2 may be the cause: when the points lie on a vertical line, all of them lie within delta of L assuming now two points with the same coordinates exist.



Lastly, we will consider the asymptotic analysis of the final two algorithms. As we see above, both upperbounds (NlogN and $Nlog(N)^2$) are both far above the actual runtimes. As we discussed earlier, this might be because they do not take into account the fact that often the lists of points are partially sorted (often due to previous steps in the recursion) which can be taken advantage of during some sorting algorithms like quicksort of Timsort (implemented in Python) to get faster, sometimes even linear sorting times. For example, these theoretical complexities fail to capture the filtering factor mentioned earlier in this report. Thus, our theoretical complexities might be a bit too pessimistic.

3 Problem 4: Extra Credit

Q: 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 $\Omega(nlogn)$, or $\Omega(nlog^2n)$, or are our upper-bound analyses actually too pessimistic?

For algorithm version 2, one thing that speeds up this algorithm a lot is that we can filter out points whose x coordinate is greater than δ away from L before we sort by y coordinate, greatly speeding up the $O(nlogn)$ sorting time if there is few points within this range. In the worst case, all points are within this range (i.e. all points lie within δ of L for a given round of recursion). This also slows down the subsequent part of the algorithm that searches for the minimum distance between points in separate parts as now each round of recursion has a bigger set of points to search between. This latter issue also effects version 3 of the algorithm. This makes me believe that the worst case scenario for this algorithm would be a set of points arranged on a vertical line with no points with the same coordinates. In this case, at every round of recursion every point would be within δ of L, leading to the greatest amount of work at each possible level of recursion. This will not effect the base case or the number of levels of recursion as we still divide the points in half each time, its just that there is more work to do at each level. In version 2, if the points have no natural ordering in the y coordinates, this will force a $O(NlogN)$ runtime at each recursive level as the list of points is sorted by y from scratch at each level. For version 3, this arrangement of points does not change the fact that merging the sorted (by y) subsets of points is an $O(N)$ operation nor does it change the number of levels involved, so the algorithm is still clearly $O(NlogN)$.