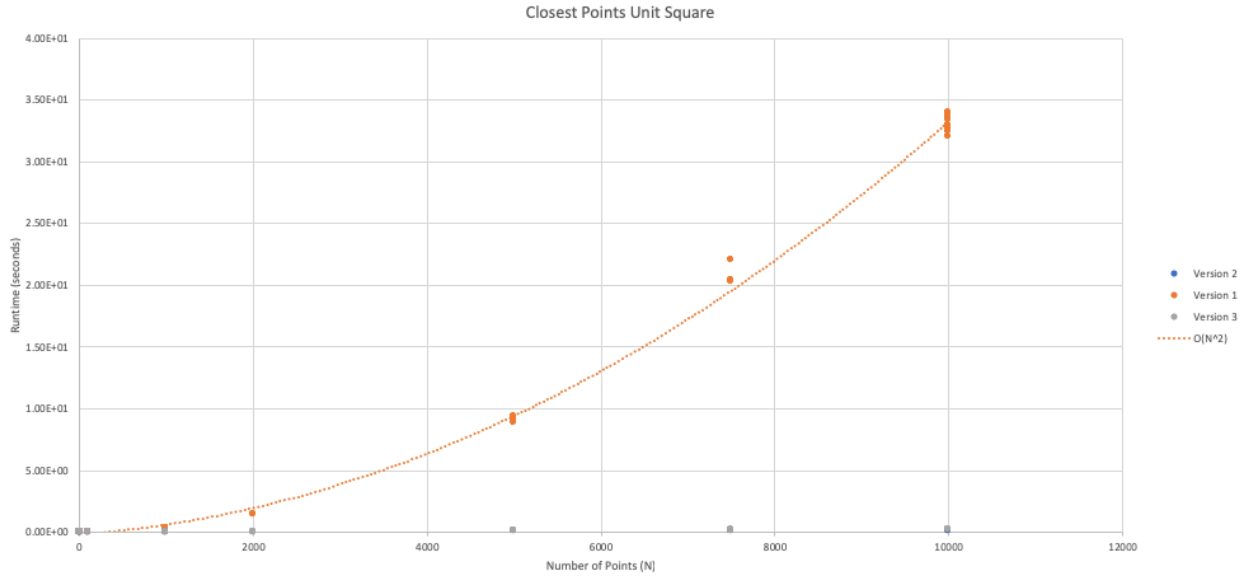
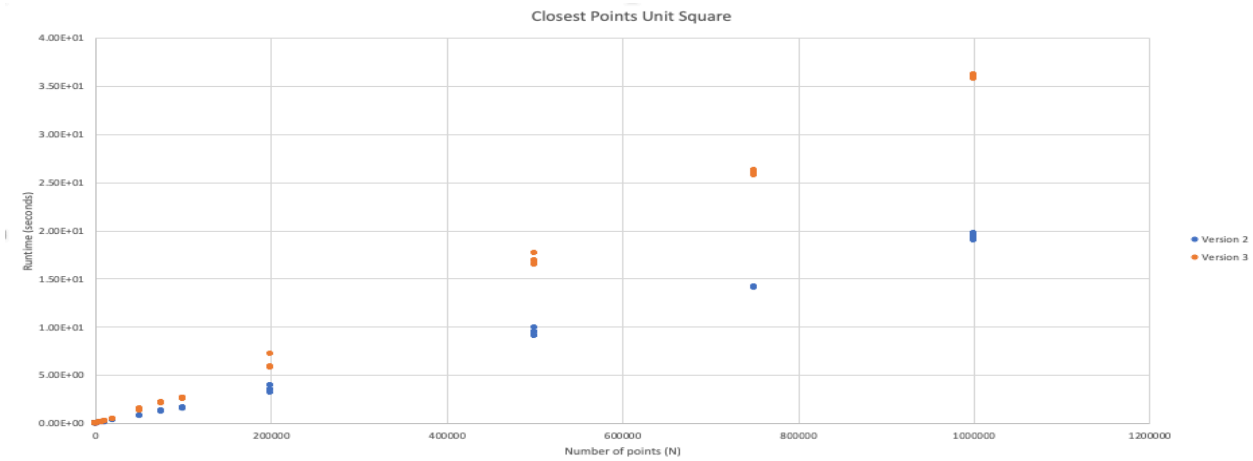
Zachary Mcnulty
zmcnulty, 1636402
CSE 417: HW4

Problem 3 Report – Algorithm Timing Analysis

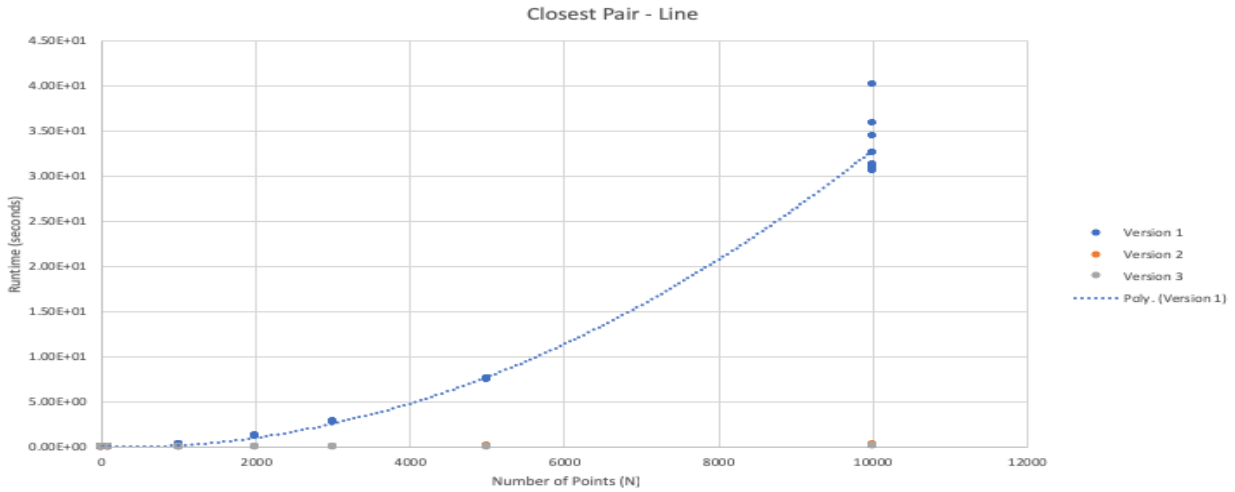
My code takes an input text file so to test the performance of these algorithms I simply generated a bunch of these test files randomly. To generate these random collections of points, I used the Python library numpy which has a convenient function for generating points uniformly at random within a given 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.



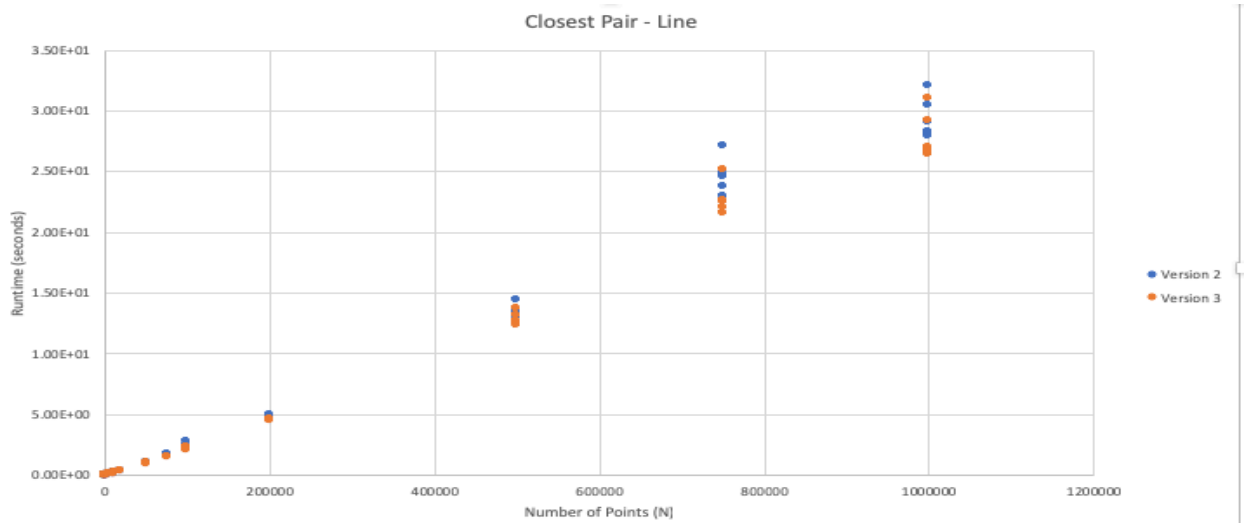
Above plots the performance of all three algorithms. We can see that the naïve "check all possible distances" algorithm is quickly outperformed by the other two. After there are more than ~2000 points, there is a clear separation between the naïve algorithm and the divide/conquer algorithms. Plotted 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.



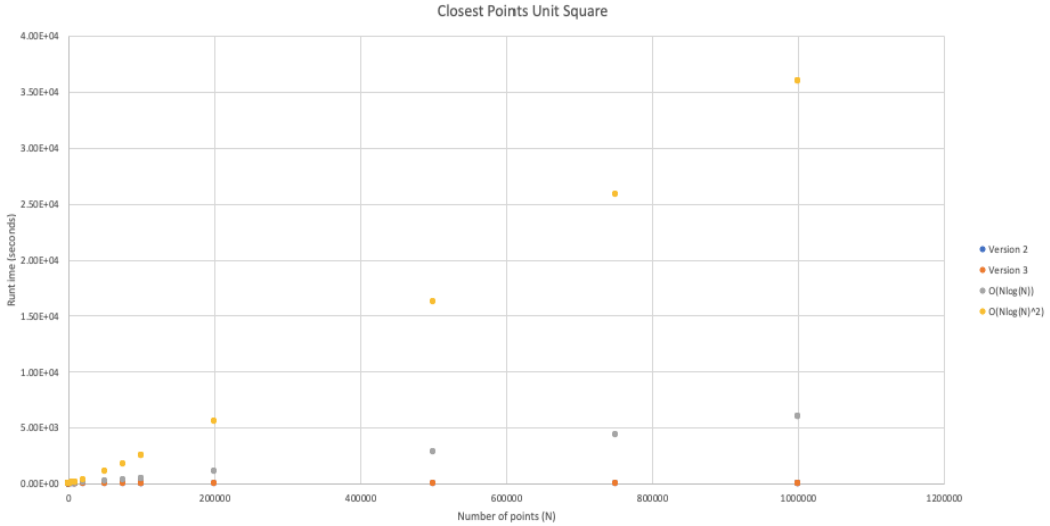
Above we can see that, despite being in a greater complexity class, the Version 2 algorithm outperforms Version 3 when the points are uniformly placed within the unit square. One reason this might be the case is that as we have it implemented the Version 3 algorithm requires sorting the entire list of 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.