COSC201 Assignment 1

7287441

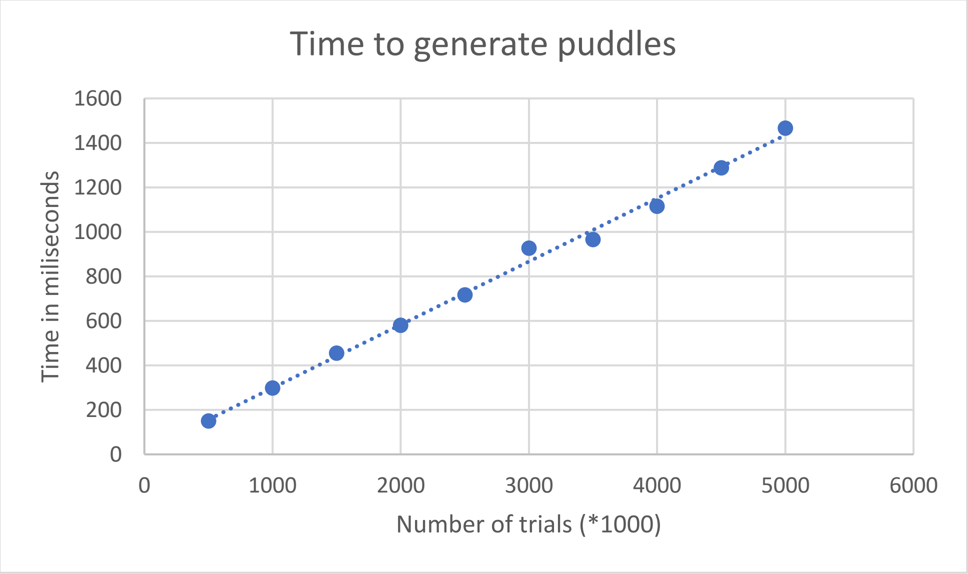
Chele691

Leon Chen

Part 1,

Generating the points

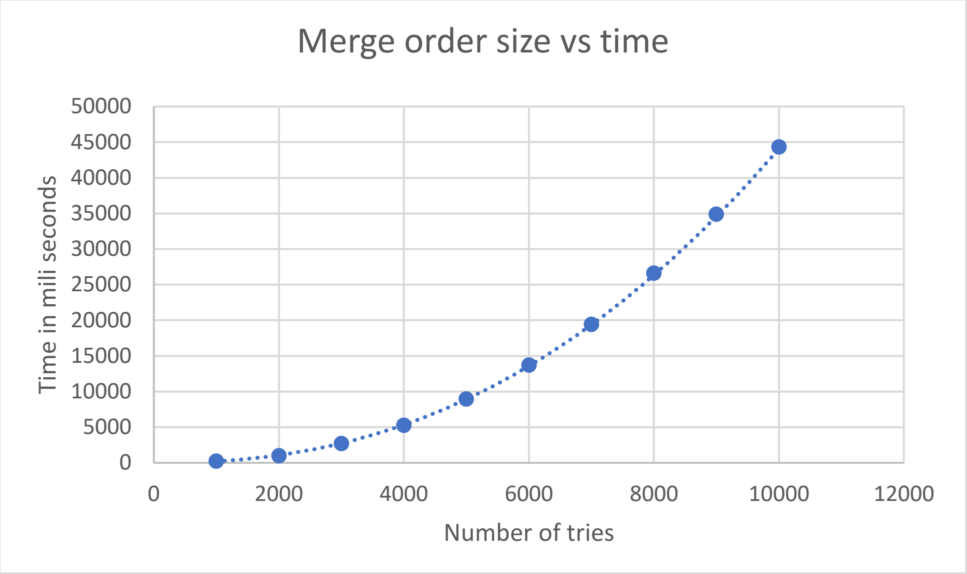
When generating the data, the big o should be bigO(n) and the line should be linear. This is as when generating the puddles, you go through the number of times and create a puddle making it a big o of n. As seen below, line is linear, and graph represents a big O of N relationship. To get these points I took the time of getting from 500 000 to 5 000 000 in increments of 500 000. I did this 10 times and averaged out the time then plotted it onto a scatter graph.



Part 2

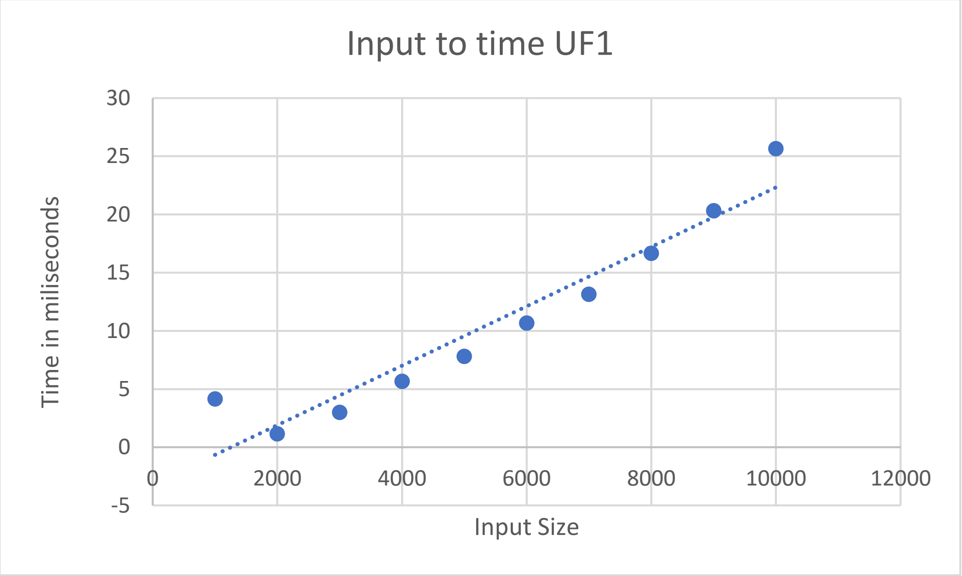
Generating the merge-ordering list

I tested the time of the merge-order list by timing the time it takes from n = 1000 to n = 100000 in increments of 1000. I did this 10 times and then average out the data and plotted. Resulting in the graph below. When analysing the code, we can see a nested loop meaning that the big O notation is n^2. When looking at the graph, the graph seems to be following the trend of exponential growth and therefore the results replicate the theory for the big O of merge order.

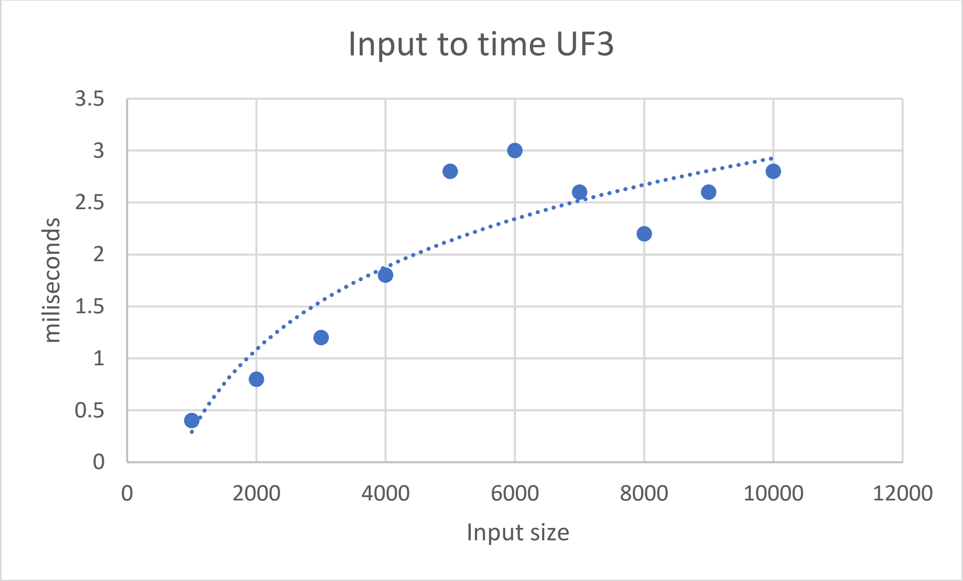


Part 3

In theory, the union for UF1 should be big O of n^2, but due to memory constraints and issues with bottle necking from merge order we cannot get proper timings to have an accurate assessment of UF1's time complexity. UF2 is also n^2 but has the same issue as UF1 with bottle necking before we collect enough data.

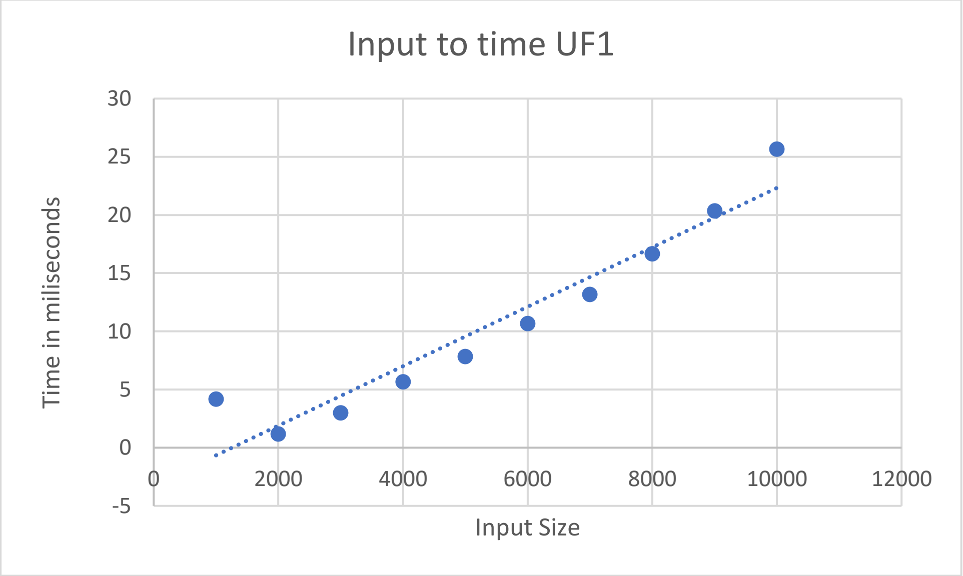


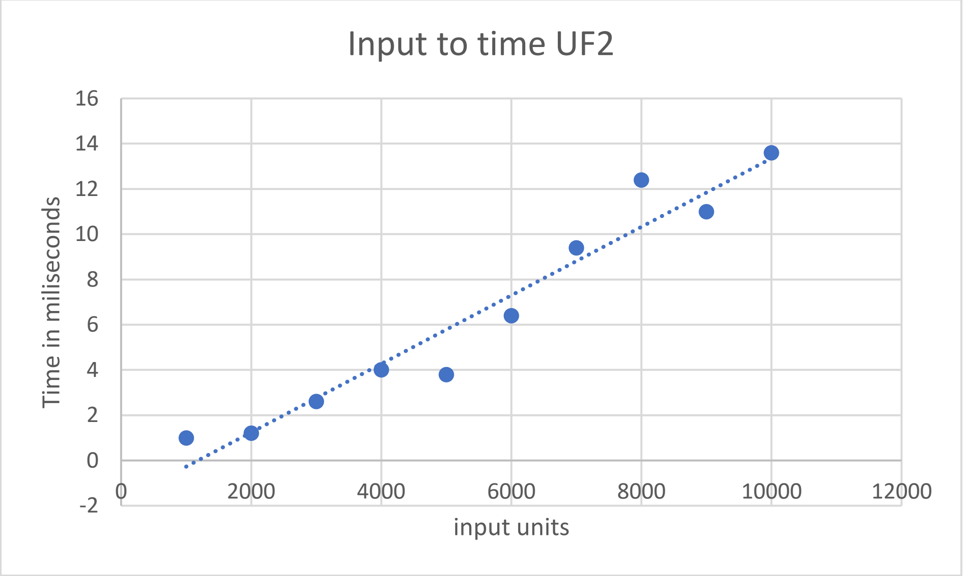
In theory, the Union for UF 3 and UF4 should be the big O of n log n, we can kind of see the relationship below, but due to bottle necking and the time it takes to do merge order before the union we cannot get an accurate reading as the time taken to do the union is very small and a lot of variabilities can occur for the time to be slightly higher or lower due to how small the time is. We can take away that UF 3,4 are faster than UF1 and 2 due to the increase in time taken, though with the memory constraints are still quite noticeable. The bottle necking happens because merge order is slow but union is fast. Meaning that we cannot an accurate recording of union without merge order bottle necking and causing a java heap issue.

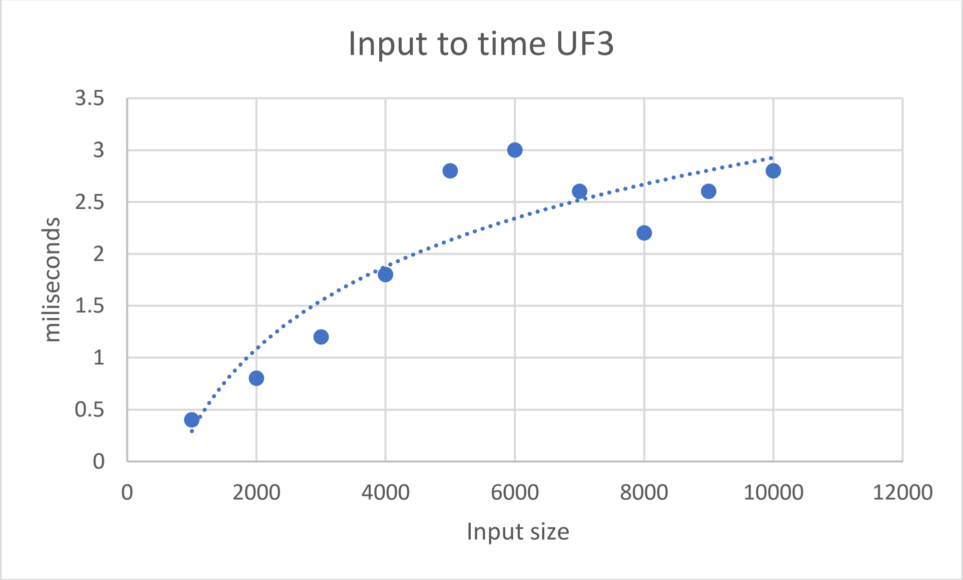


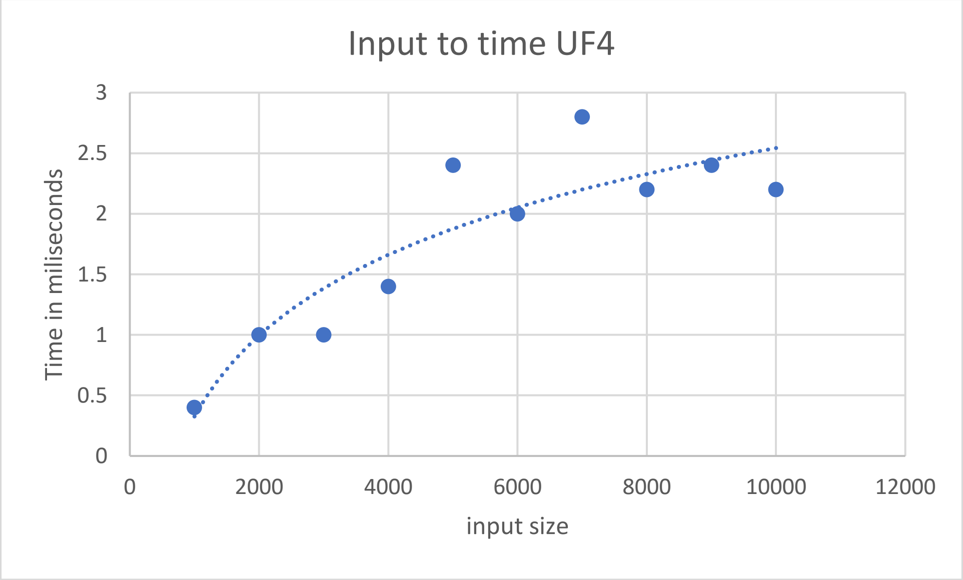
Question 2,

To get the time of the union find, I timed the union for each union find with inputs from 1000 to 10000 in increments of 1000. I did this 10 times for each union find and averaged the data out to plot. The data produced the following graphs for each of the Union finds.

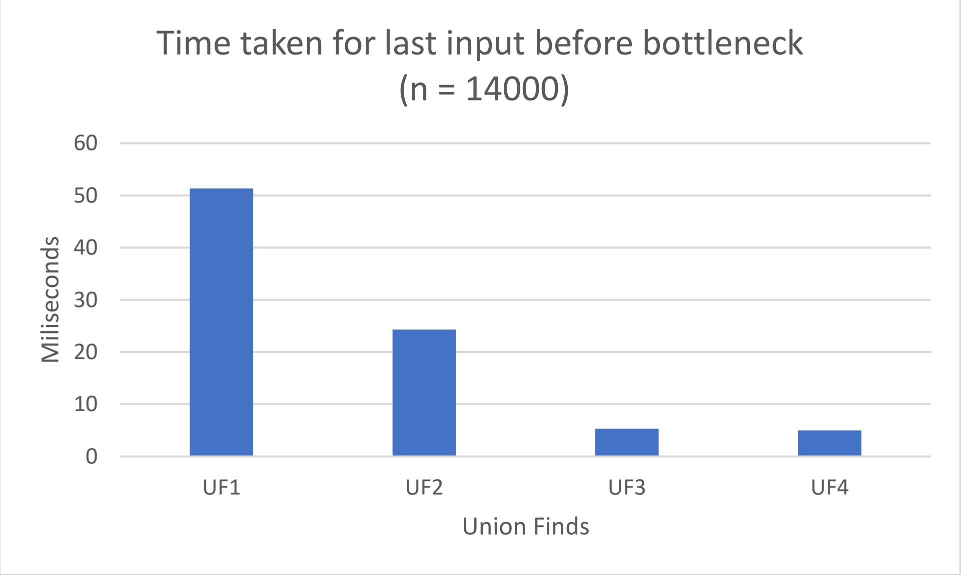



Input to time UF2
Time in miliseconds
16
14
12
10
8
6
4
2
0
-2
0
2000
4000
6000
8000
10000
12000
input units


Input to time UF3
miliseconds
3.5
3
2.5
2
1.5
1
0.5
0
0
2000
4000
6000
8000
10000
12000
Input size

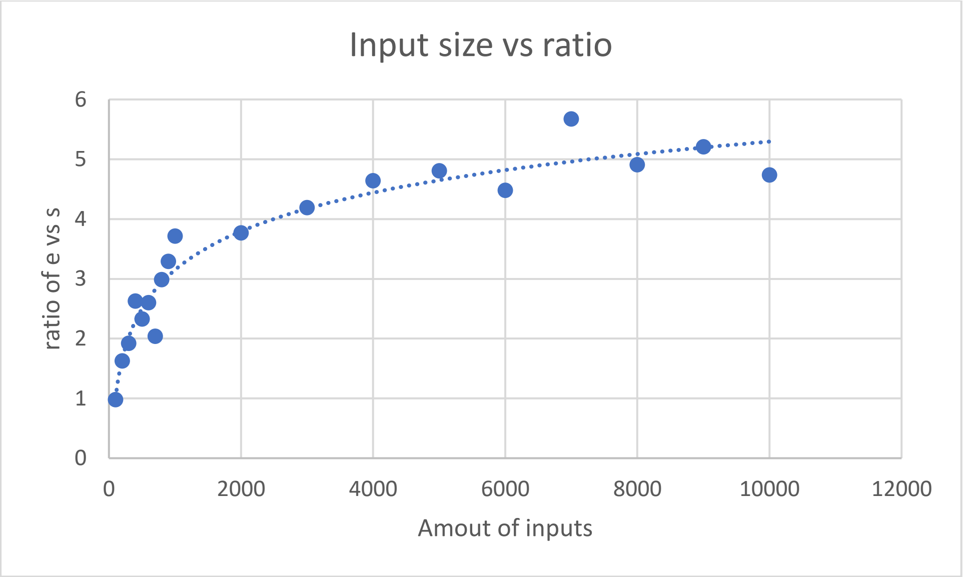

Input to time UF4
Time in miliseconds
3
2.5
2
1.5
1
0.5
0
0
2000
4000
6000
8000
10000
12000
input size

Even though, the bottle necking occurs at such a small time, we can still see some differences between the union finds. For example, UF1 is the slowest as it takes close to 25 milliseconds for a input size of 10000, and UF2 coming in 2nd with 14 milliseconds and UF3 and UF4 very similar around 3 milliseconds and difference could be due to the very small time and other variabilities for UF3 and UF 4. We cannot accurately test which one is faster due to memory constraint, but in the test it seems that UF4 is slightly faster. These all bottlenecked around 14000 units of input with UF1 taking 51.3333 milliseconds, for UF2 24.3333 milliseconds, and 5.33 milliseconds for UF3 and 5 milliseconds for UF4, shown in the graph below. As the merge order causes the union to bottle neck before we can collect enough data (time does not exceed 0.01 of a second), we can conclude that UF4 is the fastest due to the theory that UF4 keeps track of the representative when UF3 does not, giving it a slight edge over UF3 for complexity. Along with the theory UF1 being n^2, UF2 being n^2 as well m UF3 and UF4 are n log n. Therefore, UF4 is the fastest UF for both test and theory even though testing has several constraints.

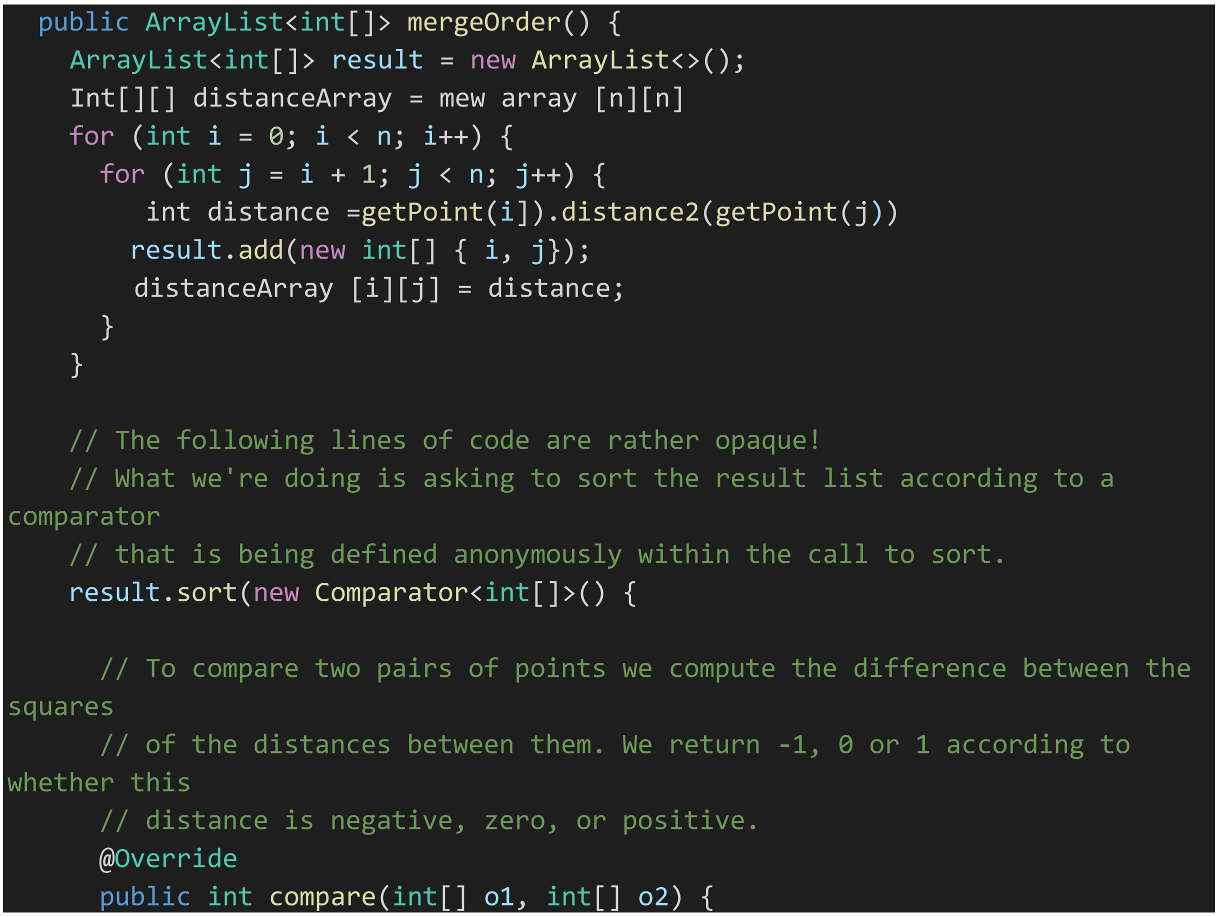


Question 3

To do this, I used UF4 and started with lower numbers from 100 to 1000 in 100 increments then from 1000 to 10000 in 1000 increments. This counts the ratio of superfluous merges and essential merges until 1 group remains. The graph seems to look logarithmic as it starts off with a lower merge ratio for lower units of inputs and then grows to around 5 and however around 5. The reason for this being is that with lower units there is a lot of space between puddles and essential merges can happen without a lot of superfluous merges. But once the input size becomes large enough, the superfluous merges happen more often and the ratio levels out. This is due to the area not growing bigger but more puddles being added, making a more compact area. With a more compact the ratio of essential merges and superfluous merges levels out as the variability of the location of the puddles becomes lower and therefore the ratio grows out and then levels out and hovers around 5. Having a logarithmic shape.



4.	I added some pseudo code to the already existing methods of merge order. What I added was, storing the distance between points I and j into an array named DistanceArray. Then comparing the points of DistanceArray and sorting result array by the distance in distance array (accessing the distance via the points in results array and sorting the results array by the values stored in the distance array). Then returning the results array sorted by the distance of the points



```
 public ArrayList<int[]> mergeOrder() {
  ArrayList<int[]> result = new ArrayList<>();
  Int[][] distanceArray = mew array [n][n]
  for (int i = 0; i < n; i++) {
   for (int j = i + 1; j < n; j++) {
      int distance =getPoint(i]).distance2(getPoint(j))
    result.add(new int[] { i, j});
    distanceArray [i][j] = distance;
   }
  }

  // The following lines of code are rather opaque!
  // What we're doing is asking to sort the result list according to a
comparator
  // that is being defined anonymously within the call to sort.
  result.sort(new Comparator<int[]>() {

   // To compare two pairs of points we compute the difference between the
squares
   // of the distances between them. We return -1, 0 or 1 according to
whether this
   // distance is negative, zero, or positive.
   @Override
   public int compare(int[] o1, int[] o2) {
         double dd = distanceArray [o1[0]][o2[1]] – distanceArray
[o2[0]][o2[1]]

        if (dd == 0.0)
          return 0;
        return (dd < 0 ? -1 : 1);
      }

    });
    return result;
  }
```