COSC201 Assignment 1 Forming a pool

Yuki Yoshiyasu
5861229

Introduction

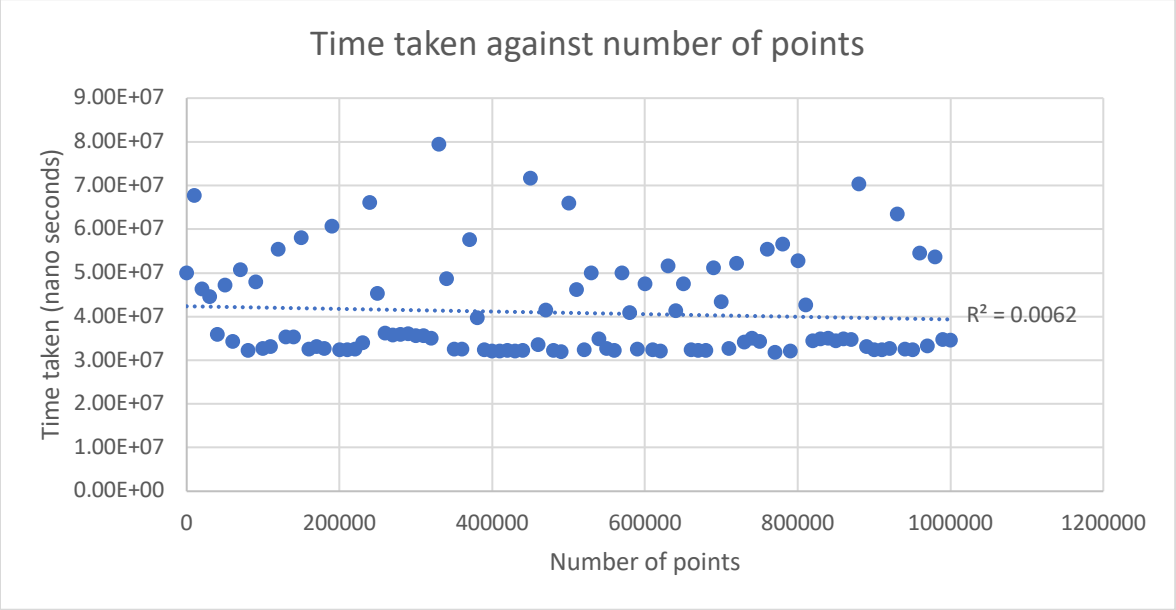
The purpose of this assignment is to investigate the efficiency of the various union-find implementations we have considered, and to learn just how large a union-find instance can be handled smoothly on the hardware you have available.

Question 1

Throughout all the phases, I have used UF 4 for all experiments to ensure reliable/valid results.

Phase 1

The data of generating points with the time taken turned out unexpected as the plots didn't show a linear increase. To calculate the time for each point generation, I have modified the A1Demo.java class then started the timer before the condition in the for loop where the point gets created and stopped the timer after the point is created. This ensures obtaining the correct time for each point generated. From the chart below, we can observe that 62% of the values fit the model.



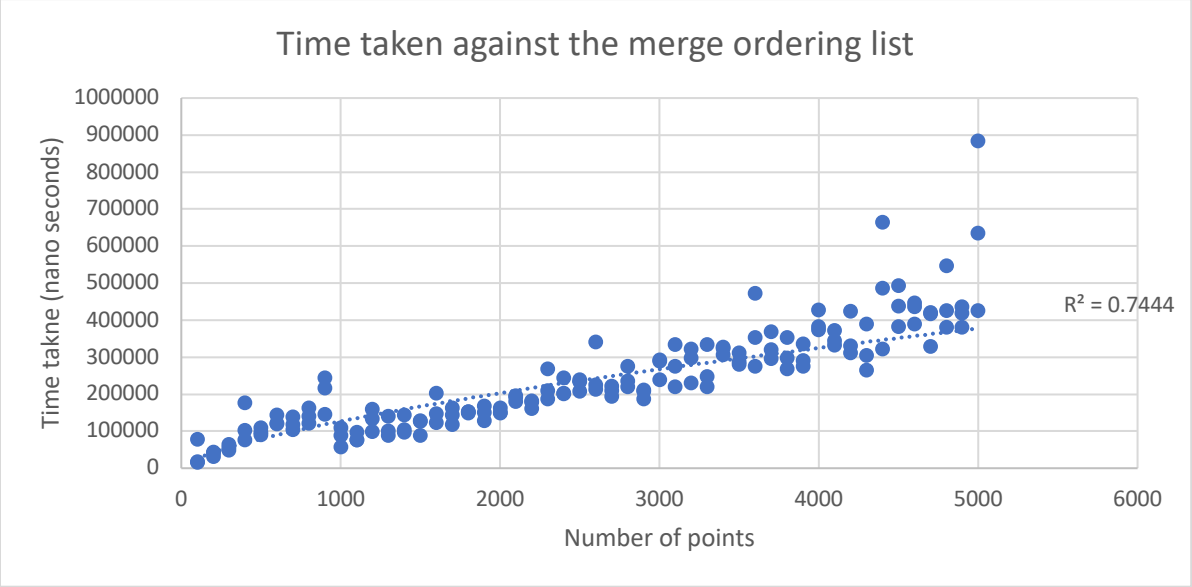
In theory, generating the points should be linearly generated O(n) for its worst-case scenario. The allocation and initialisation of n places in the memory take constant time per place. Since getting the point is just the worst case of O(1), the number of points proportionally increases against the scale of the operations.

In conclusion, my chart does not prove my theory since we cannot observe a linear increase from the blue trendline. However, this does not write off my theory as the theory was calculated by figuring out each operation's worst-case scenarios. Since I reached the memory constraint before all the data was generated to prove the theory. Thus, the data

generated doesn't show the expected results because my hardware handled the point generations rapidly.

Phase 2

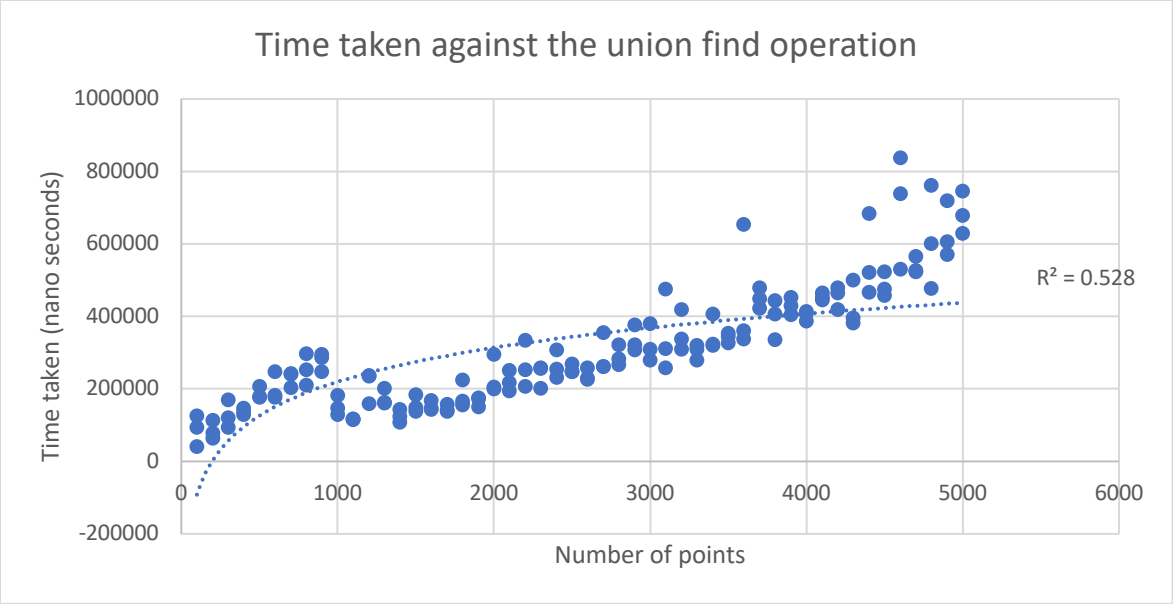
In theory, the time complexity of generating the merge order list should be quadratic (O(N^2)) operation, since the number of points doubles with each increase in order. To measure the times taken for the merge ordering list, I have modified the code from A1Demo.java. Where the timer starts in the loop for each pair in the merge order and then stopped after getting the points from the pairs. This is to ensure that we obtain accurate times. Also, the experiment ran for 100 trials to reduce random errors.



Upon the observation of the trend line of the chart generated above, we can see the time taken is approximately doubling each increase in the number of points. By interpreting the nested for loops in the merge ordering list, the worst-case scenario should be O(N^2). Which is evident from the model obtained by the experiments. We can also observe that 74% of the data fit the model.

Phase 3

Generating the union find times was concluded by starting the timer before the union operation and stopping the timer straight after it in the altered A1Demo.java class. The experiment was trialled 100 times to eliminate random errors.



In theory, the union find operation for UF4 should be a worst case of O(n log n). Since the union operation in UF4 recursively calls the find method and is bounded by the time for two find operations. Because UF3 has a worst case of O(log n) without the recursive find method. This can be supported by the plots and the trendline in the chart above. As the number of points increases, the time taken to increase logarithmically. At the start of the chart, we can observe a longer time taken, this is due to JIT compilations. We can also observe that only 52% of the data is represented by the model. However, this is most likely because the first operations take a longer time than the subsequent operations altering the trend line.

Question 2

Initial findings

The timings were standardized by starting the timer before the for loop where the union operation occurred. After running each trail 100 times starting at n = 1800 and incrementing each time by n+=25, I have concluded with the data below.

Size of UF1:
Approximately a size of 1880 ~ 1900 resulted in a time of 1 second to execute.

Size of UF2:
Approximately a size of 1860~1900 resulted in a time of 1 second to execute.

Size of UF3:
Approximately a size of 5900~6100 resulted in a time of 1 second to execute.

Size of UF4:
Approximately a size of 8300~8600 resulted in a time of 1 second to execute.

Upon observing these values of sizes, the difference isn't clear between UF1, UF2 and UF3, UF4. Therefore I will need to measure the sizes in a precise manner, such as, by decreasing the incrementing value.

The first two union find methods, (UF1 and UF2) had an indistinguishable difference in terms of efficiency per second. This is as expected, since in theory, UF1 has a worst-case time complexity of O(n). Since The loop performs a constant number of operations O(1) for each element of the representative array, which results in a time complexity of O(n) for the loop. Whereas UF2 in theory has a worst-case time complexity of theta(find()) and the find() operation itself has a time complexity of theta(n), therefore the execution is slightly faster than UF1.

UF3 and UF4 had a greater difference of 5900 to 8300. In theory, UF3and UF4 has a worst case of O(log n complexity), even though the time complexities are similar, UF4 has an improved find function with the worst case complexity of O(n log n) since it recursively calls the find method.

These experiments were possible due to my hardware. My hardware consists of a 3.2 GHz M1 Pro CPU with 8 cores, a 14-core GPU, a 16-core Neural Engine, 16 GB of onboard RAM, and a 512 GB onboard SSD. Therefore was able to run the union find instances to 1 second without any bottleneck or memory constraint effects.

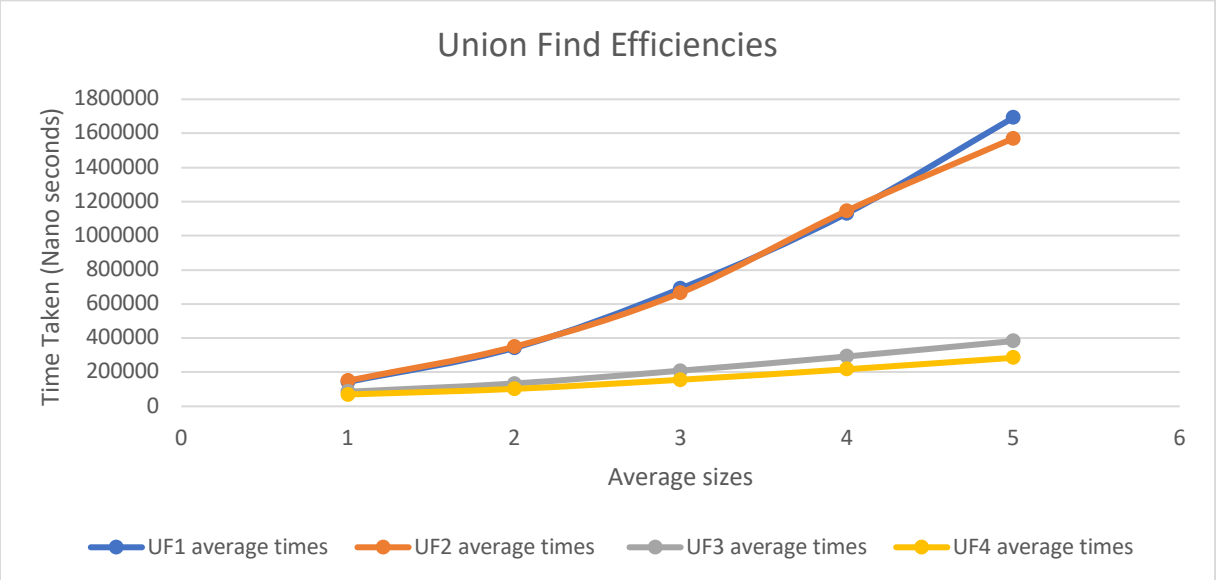
Results/models

The table below shows the findings of all the Union Find instances with their mean time. Each trial has been executed 100 times to reduce random errors and improve the precision of experimental measurements via statistical averaging. The timer times the execution of the loop process, which only includes the time to union.

The mean time taken (seconds) for the union find against different sizes.

	N=500	N=1000	N=1500	N=2000	N=2500
UF1	0.1426s	0.3420s	0.6901s	1.1314s	1.6921s
UF2	0.1502s	0.3492s	0.6655s	1.1467s	1.5708s
UF3	0.0854s	0.1338s	0.2090s	0.2922s	0.3826s
UF4	0.0692s	0.1021s	0.1556s	0.2171s	0.2845s

*UF efficiencies charts average size is supposed to be incremented by 500 each time.



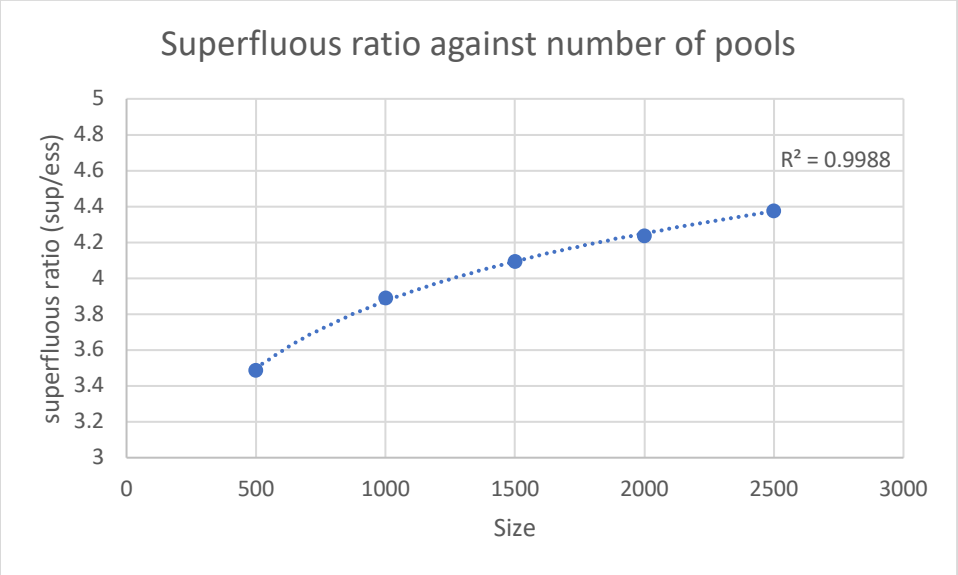
We can confidently say that it is evident from observing the table and the trend lines of the UF efficiencies, that UF4 is the most efficient union-find implementation.

Question 3

To hypothesize the ratios between the two union operations, I utilised the most efficient method (UF4) to run the experiments. As this is apparent from the questions previously answered. The results below are generated using the firstSuperfluousMerge and superfluousRatio methods in the poolAnalyser class. Each method had been trialled 100 times to eliminate outliers which may reduce the precision of the results. As well as converting the types of merges to a double from an int has also greatly improved the precision of results.

Size vs ratio between superfluous merges and essential merges.

	N = 500	N = 1000	N = 1500	N = 2000	N = 2500
firstSuperfluous merge (average)	34.29	46.15	55.53	60.29	69.34
superfluousRatio(sup/ess) merge (average)	3.487	3.892	4.094	4.238	4.375



As we can observe from the table above, the ratio between superfluous unions and essential unions increases logarithmically to the increase in pool sizes. We can also interpret the trend line from the graph above, in which the model fits perfectly with 99.8% of the results with a logarithmic increase. This result is understandable because the larger the number of puddles, the more likely superfluous merges will occur. Therefore, there should be more opportunities for superfluous merges to occur. Theoretically, there are N^2/2 of distinct pairs of puddles, meaning pool size -1 number of essential merges are happening. We can observe from the table above that the results obtained roughly follow the theory stated above, as the ratio of the two merges increase as pool size becomes greater. However, as we increase the pool size to N=2000 onwards, the ratio tends to decrease or hit a plateau.

The reason why the data hits a plateau is that the total ratio is calculated by (n-1)/(n^2-n) – (n-1). Therefore, the increase in size also increases the possibility of essential merges and superfluous merges equaling each other. Which is why we see the merges ratio flattening out.

Question 4

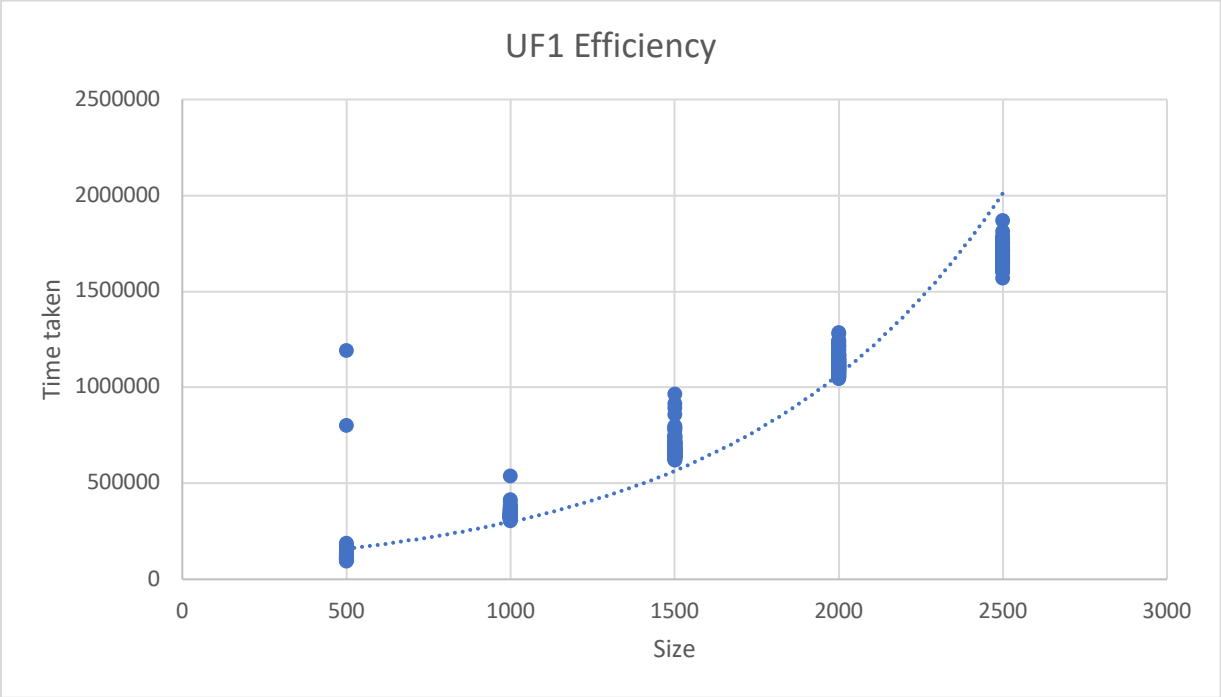
To store the distance between points, I plan to use a 2D array of type double. The array's dimensions will be n by n, where n represents the total number of points. Using this array, I will compare each pair of points (pair[x], pair[y]) by looping through each value and determining if it's positive, negative, or zero, and then storing the results in the 2D array. This will allow for easy access to precomputed distances for comparison. To sort the array of precomputed distances, the Arrays.sort() method can be used, or merge sort concepts such as divide and conquer can be utilized to sort the points.

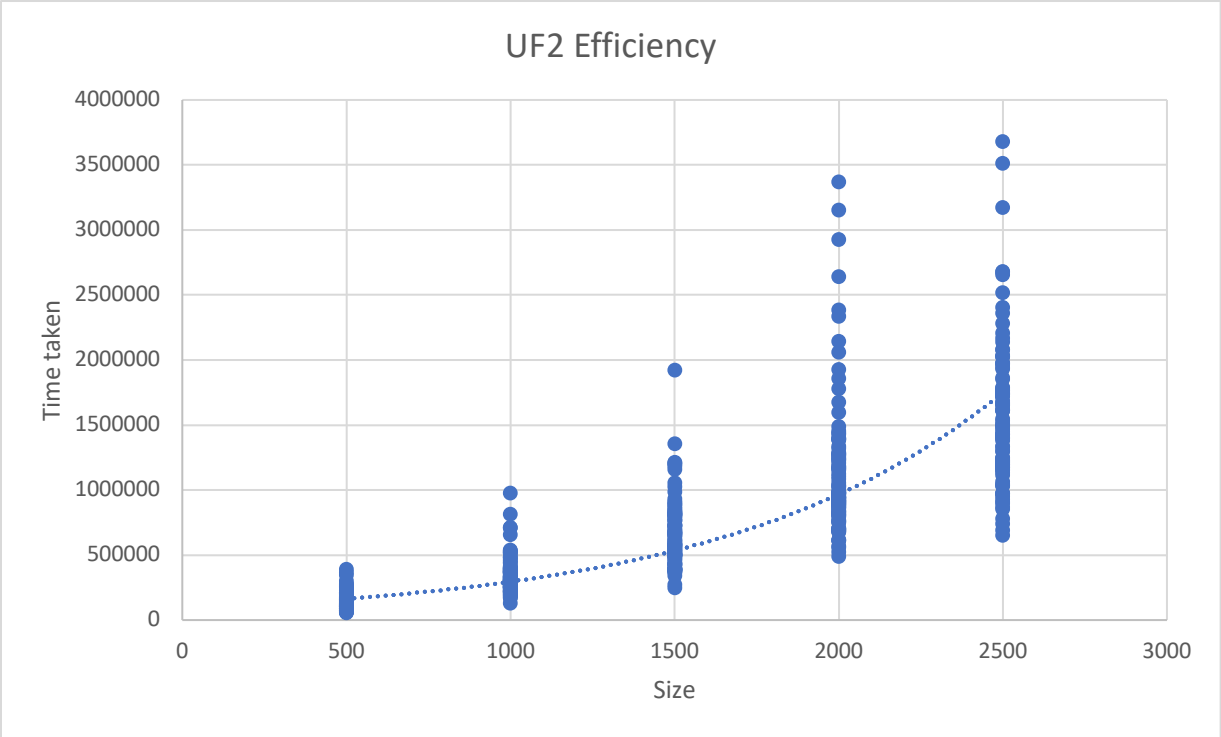
Pseudo code below:

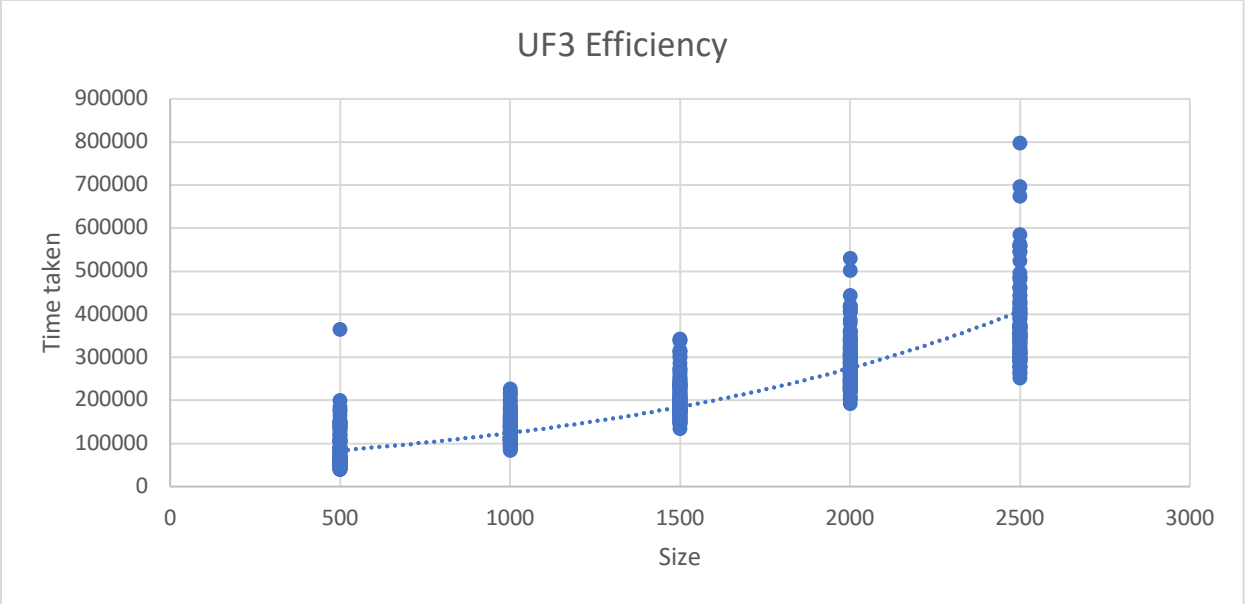
```
// Initialize the array to store the distances
double[][] distances = new double[n][n];

// Go through each pair of points using a nested for loop
for (int x = 0; x < n; x++) {
  for (int y = x + 1; y < n; y++) {

// Compute the distance between the points and store it, I will just be modifying the merge order methods.
    double distance = computeDistance(pair[x], pair[y]);
    distances[x][y] = distance;
    distances[y][x] = distance;
  }
}

// Sort the array of precomputed distances using Arrays.sort()
Arrays.sort(distances);

//The method returns a double, as distances won't always be a whole number

Public double computeDistance(point a, point b){

//I will be using the compare method in the mergeOrder class

double distance =  getPoint(o1[0]).distance2(getPoint(o1[1])) -
getPoint(o2[0]).distance2(getPoint(o2[1]));

return distance;
}
```

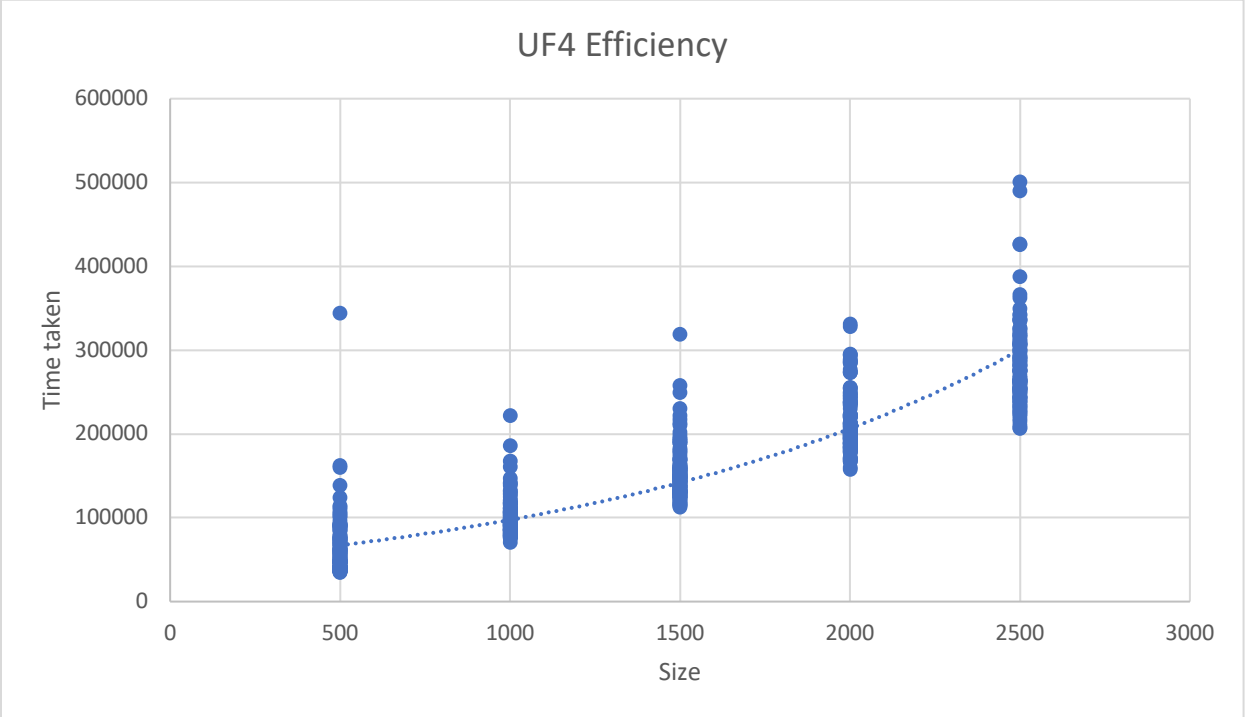
Extra resources

Additional graphs showing each UF efficiency for question 2.
*All UF efficiency charts' time taken are measured in nanoseconds.






UF3 Efficiency
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
Time taken
0
500
1000
1500
2000
2500
3000
Size


UF4 Efficiency
600000
500000
400000
300000
200000
100000
0
Time taken
0
500
1000
1500
2000
2500
3000
Size