Programming assignment 1 – Emil Collén

Setup

For this assignment, I have been using VSCode on my two windows machines (a laptop and a desktop).

Problem 4

I used the array sizes and number of unions shown in the tables 1-3 below. The time taken is described in these tables and displayed in scatterplots in figures 1-3. The number of unions depends on the array size, in order to be in proportion to the array size. In these tables and figures are both simple Union find (UF) and Quick Union find (QUF) presented. In each case, the results are taken the average of several repetitions to make the result more representative of the time taken.

In all cases, we can both visually determine that the regular UF has a linear growth. That is, it is linearly dependent, and for example, doubling the number of unions will always double the time taken. We can also compute the slope, by taking some examples from time measurements from table 2. The slope always remains at around 0.00007, during these measurements. The same principle goes for the other array sizes as well, but the slope may vary. Computations:

25-50k:

3.745-1.905 / 50 000-25 000 = 1.84 / 25 000 = 0.0000736

100-125k:

9.132-7.385 / 125 000-100 000 = 1.747 / 25 000 = 0.00006988

175-225k:

16.784-13.117 / 225 000-175 000 = 3.667 / 50 000 = 0.00007334

The QUF however, is more difficult to determine its growth, as it is basically taking no time whatsoever in the beginning, but once the number of unions grow in relation to the array size, the time taken rapidly grows. This is due to the height of the "trees" that are formed, when one considers the unions as a tree-like structure. When these trees grow in height, every union becomes much more costly as it must climb upwards in the trees. This can be avoided by path compression, where the height of the tree is shortened in each root method call. Path compression is not implemented here, as I chose the use the QUF instead. Some example computations from table 2:

25-125k:

0.005-0.001 / 125 000-25 000 = 0.004 / 100 000 = 0.00000004

200-225k:

8.071-3.503 / 225 000-200 000 = 4.568 / 25 000 = 0.0018272

In the case above, the slope between 200-225k unions is roughly 4500 times steeper than between 25-125k unions.

I would say that the results for both implementations match my expectations, as UF loops through the entire array each time, which makes it linear ~N. The time taken for QUF is based on the distance to the root, which is very small for quite some time. Once the number of unions reaches around 70% of the array size, the distance quickly grows and thereby the time taken grows exponentially. This is presented in figure 4.

Table 1 - 100k objects, time taken

100k objects		
Unions	UF (s)	QUF (s)
10 000	0.323	0
20 000	0.681	0
30 000	0.971	0.001
40 000	1.283	0.001
50 000	1.555	0.002
60 000	1.883	0.011
70 000	2.203	0.102
80 000	2.494	0.373
90 000	2.899	0.861

Table 2 - 250k objects, time taken

250k objects		
Unions	UF (s)	QUF (s)
25 000	1.905	0.001
50 000	3.745	0.001
75 000	5.573	0.003
100 000	7.385	0.004
125 000	9.132	0.005
150 000	11.21	0.08
175 000	13.117	0.951
200 000	15.249	3.503
225 000	16.784	8.071

Table 3 - 500k objects, time taken

500k objects		
Unions	UF (s)	QUF (s)
50 000	7.35	0.002
100 000	14.591	0.003
150 000	22.029	0.005
200 000	29.467	0.007
250 000	36.818	0.011
300 000	44.617	0.373
350 000	51.47	4.52
400 000	59.035	16.027
450 000	65.379	35.766

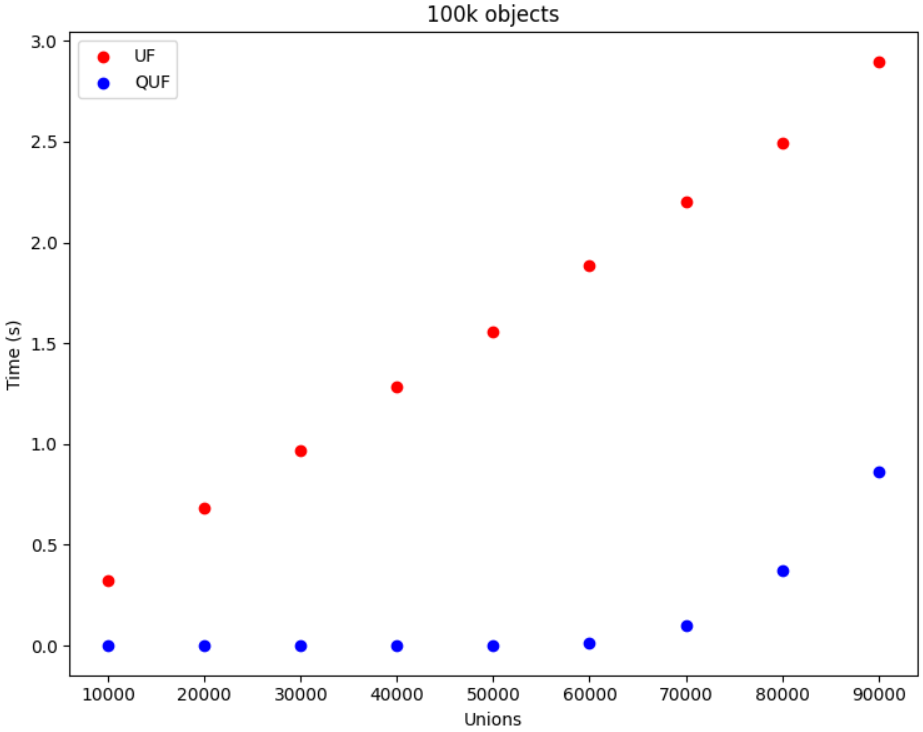

Figure 1 - 100k objects, time taken

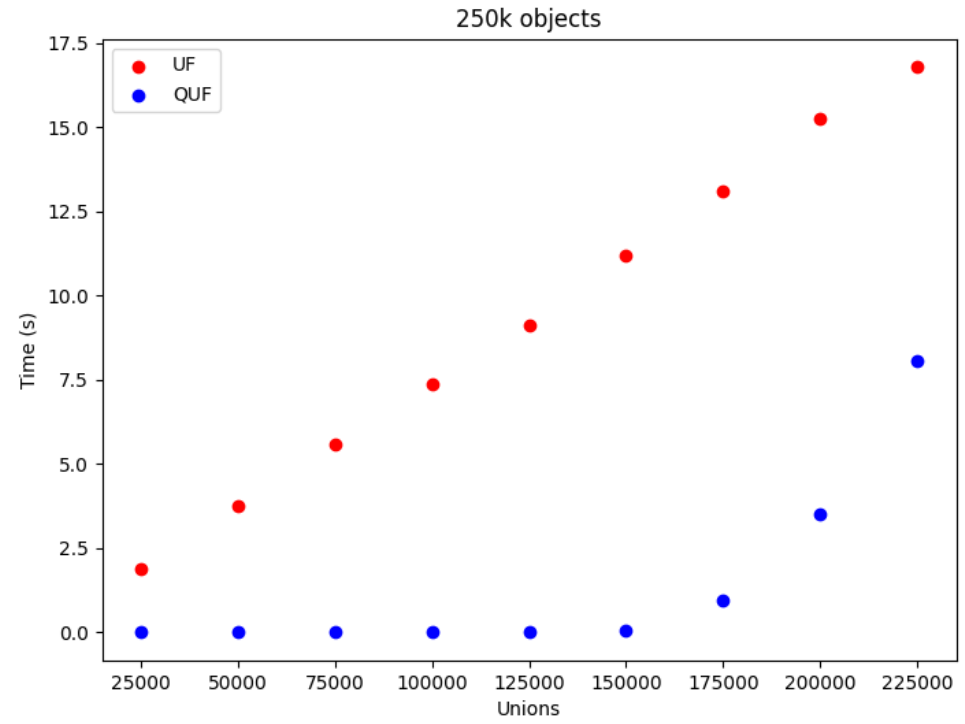

Figure 2 - 250k objects, time taken

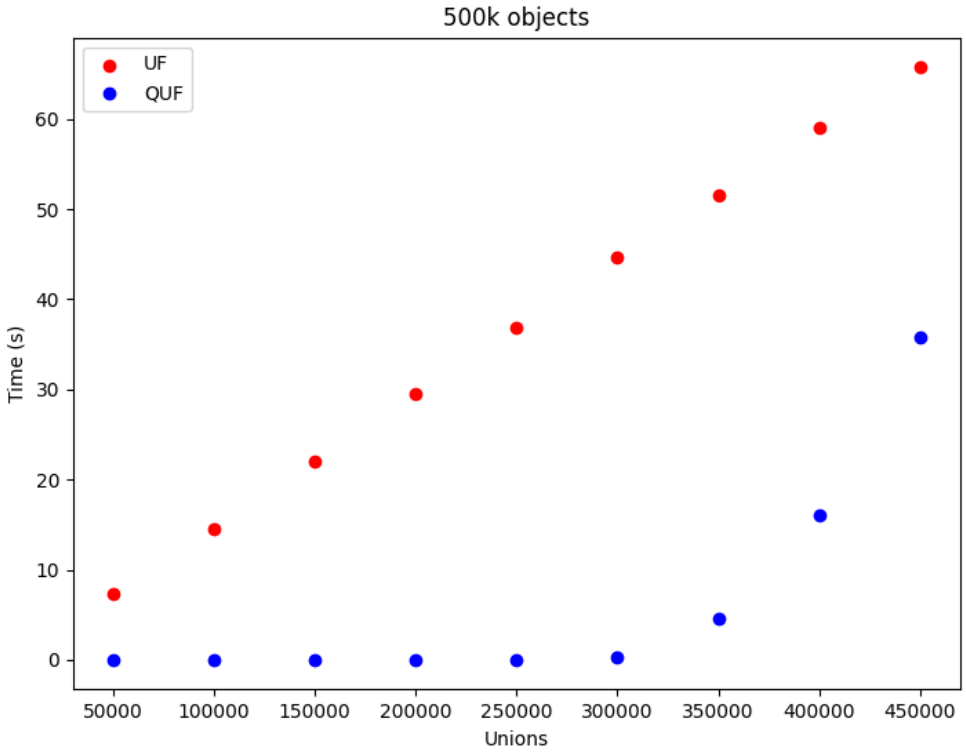

Figure 3 - 500k objects, time taken

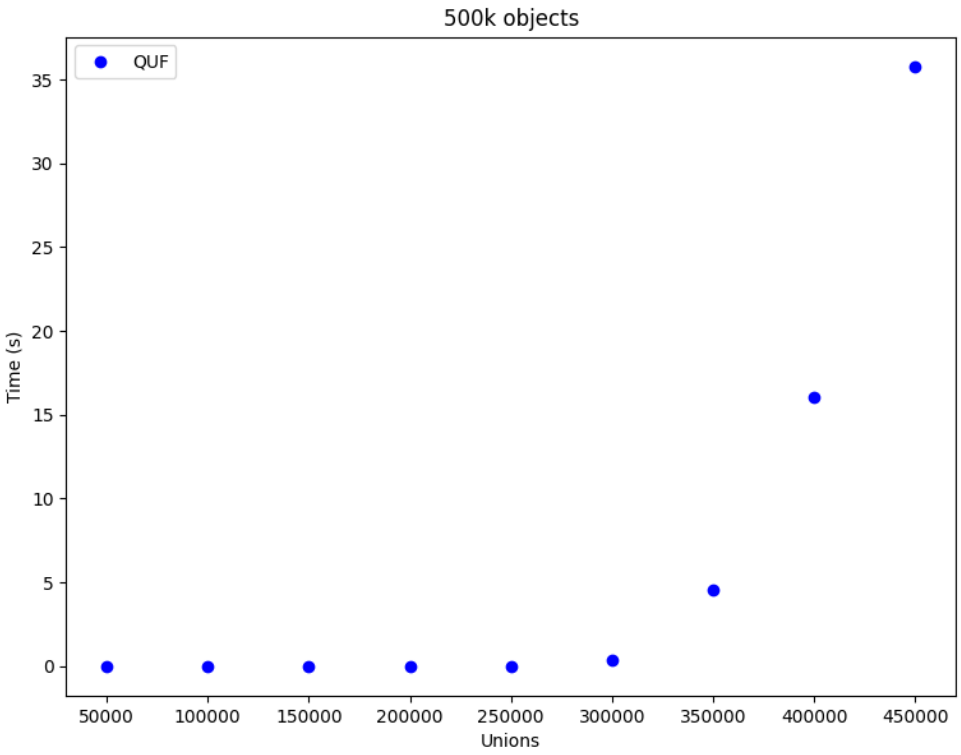

Figure 4 - 500k objects, time taken – QUF

Problem 7

The two 3sum implementations were ran on the array sizes shown in the tables 4 and 5 below. These tables also present the time taken and the log2 ratio between each array size. These timings were run multiple times to get an average, and the number of repetitions depends on the array size. The faster they ran, the more times went into calculating the average. As an example, the longest ones, such as 6400 on the brute force technique, were run only a few times while the shorter (using both techniques) were ran hundreds of times.

Table 4 - Brute force, problem 5

Array size	Time (s)	Log2 ratio
200	0.007	-inf
400	0.053	2.9205
800	0.423	2.9965
1 600	3.192	2.9157
3 200	27.167	3.0893
6 400	223.612	3.0410

Table 5 – Caching, problem 6

Array size	Time (s)	Log2 ratio
200	0.0017	-inf
400	0.007	2.0418
800	0.029	2.0506
1 600	0.123	2.0845
3 200	0.520	2.0798
6 400	2.212	2.0887

Analyzing the result, we can determine that the brute force technique takes $\sim N^3$ while the caching method takes $\sim N^2$. This means that smaller arrays basically doesn´t matter what technique we use, but as soon as the size starts to grow, we get a tremendous difference in the time takes as brute force outgrows caching extremely fast.

This result is also to be expected, as we for the brute force method are using a loop in a loop in a loop, which results in Big Omega (cannot get better than $\sim N^3$). When we use caching, we avoid the third loop, since the third loop is basically replaced by a linear operation when we are using a hash map to cache values we have already checked. This means, that we get $\sim N^2$ in our best case.

To compute the models using powerlaw, we have that:

Model : $y = a * x^b$

$a = 2^c$

For brute force:

$c = \log2(27.167) – 3.0893 * \log2(3200) = -31.20758157$

$y = 2^{-31.20758157} * x^{3.0893} = 4.03256945E\text{-}10 * x^{3.0893}$

For caching:

$c = \log2(0.52) – 2.0798 * \log2(3200) = -25.16030858$

$y = 2^{-25.16030858} * x^{2.0798} = 2.666812102E\text{-}8 * x^{2.0798}$