Assignment 1 report

Jesper Wingren
Jw223rn@student.lnu.se
1dt907

List of content

1. Union find

1.1 Setup

The union find problems were solved by making two classes for the different solutions and a problem4 class that solves the fourth problem using a for loop and the timeit method. This is where I get the time results for running my algorithms.

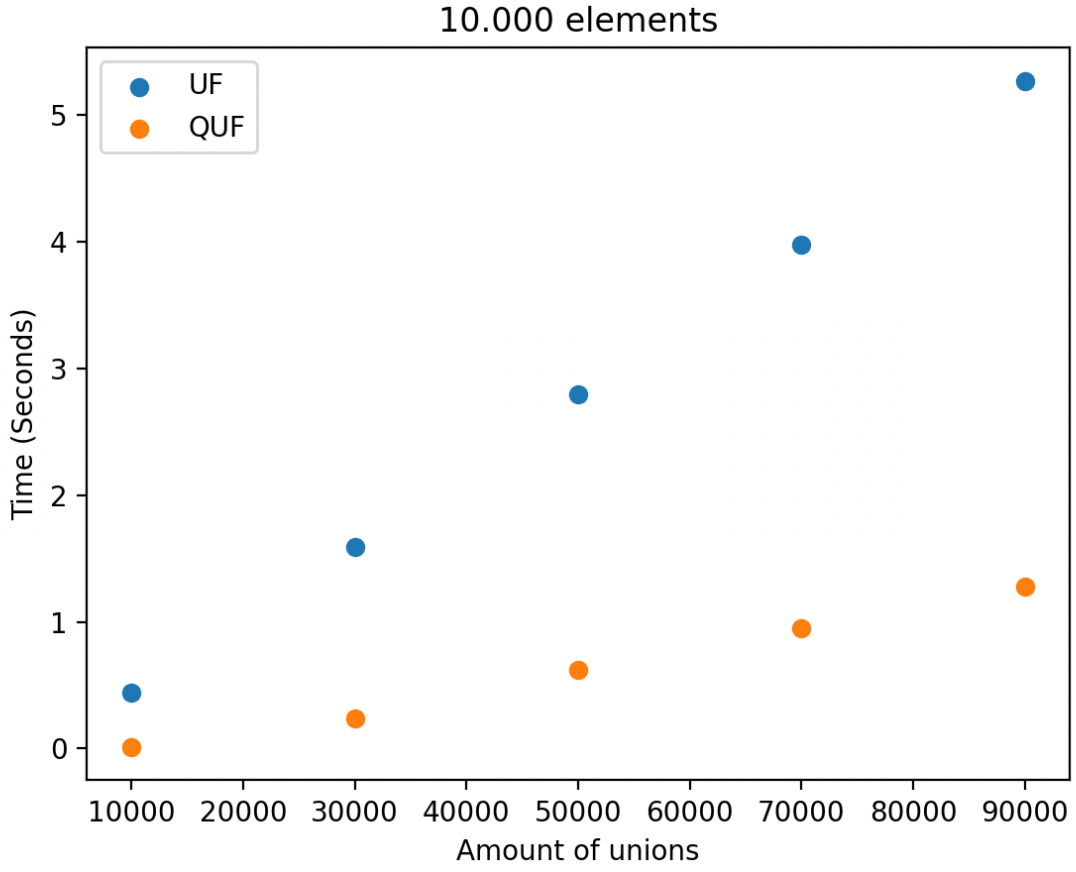
1.1.1 Union find

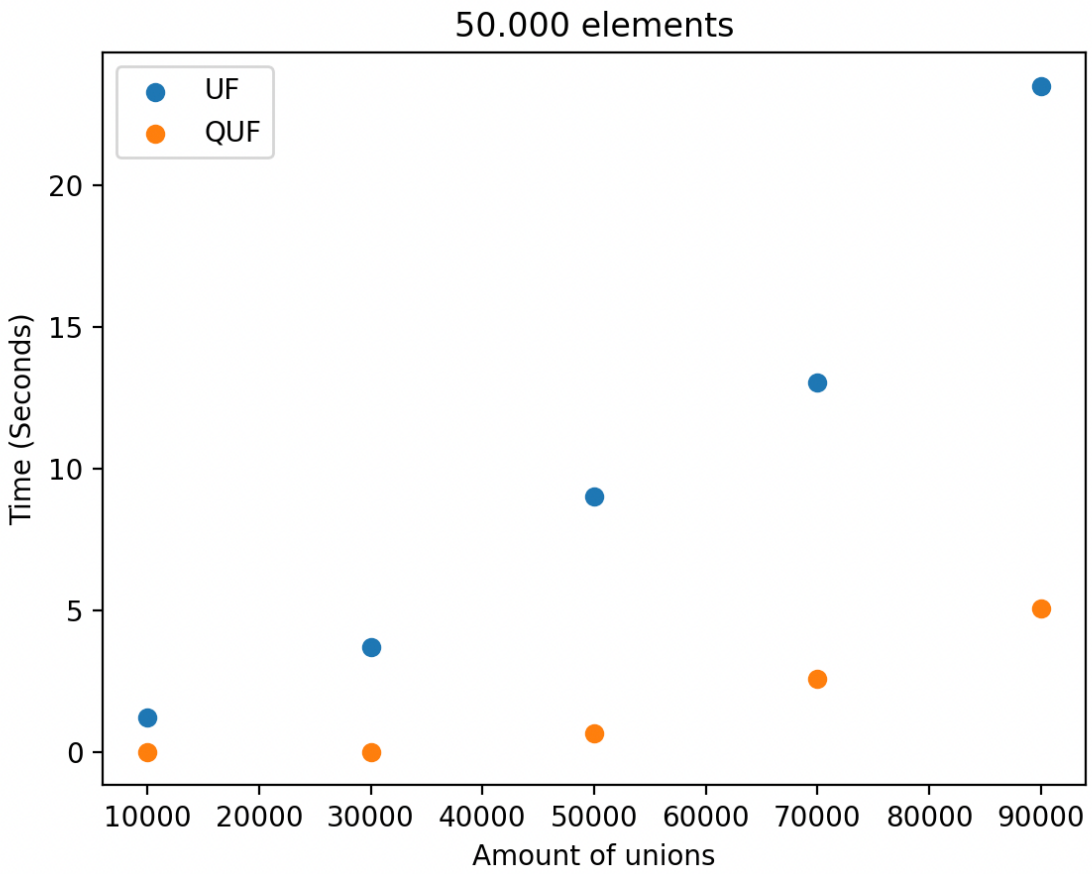
Union find was setup in a class called UF and with a property of mylist which is the list used to solve the problem. UF contains 2 methods, connected and union. The connected class checks if two numbers are connected by checking each index and their value while the union changes the value at the indexes to be the same thereby "connecting" them.

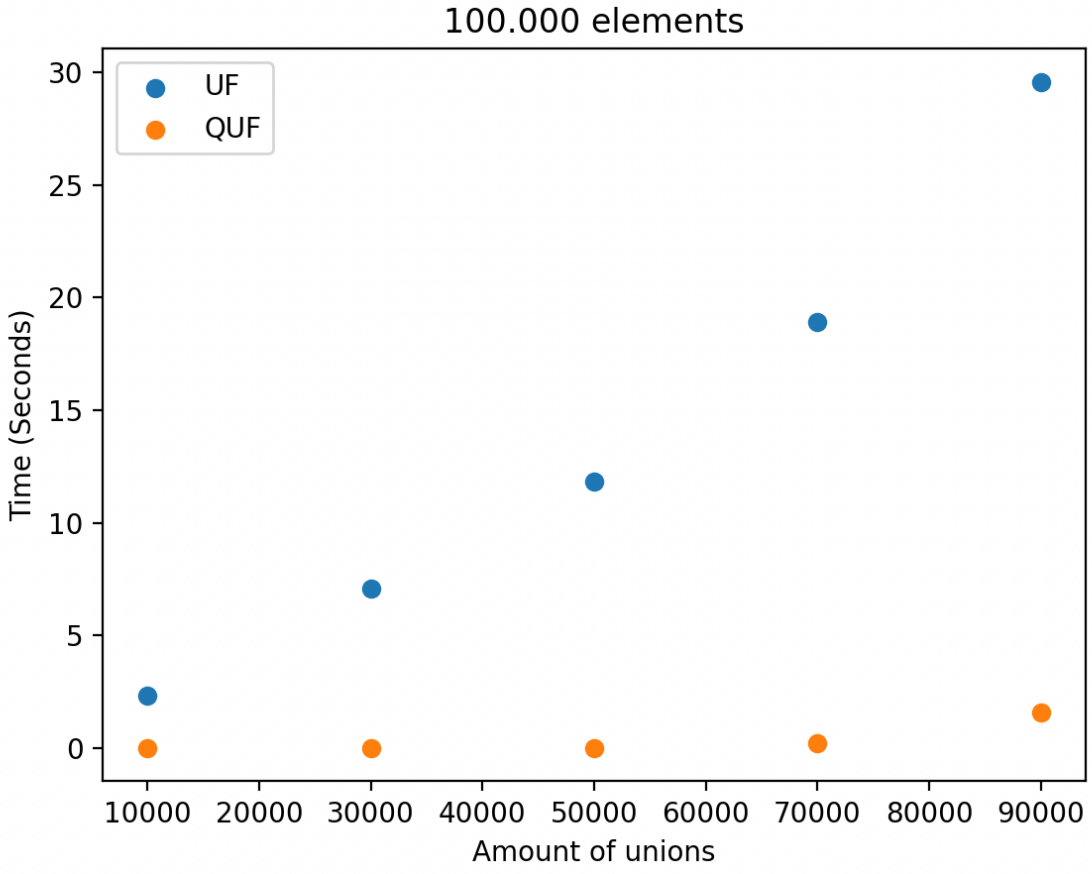
1.1.2 Quick union find

Union find quick sort was setup in a class called QUF with the property, mylist. This list is what is used to solve the problem. This class contains three methods, root, connected and union. The root checks the root of a given value by going to its index value until it matches the value thereby finding the root. That method isn't used by itself instead it is used as a helper method in connected and union. The connected simply checks if the roots are the same for two values and union matches the roots of two set values.

1.2 Results




50.000 elements
UF
QUF
Time (Seconds)
0
5
10
15
20
10000
20000
30000
40000
50000
60000
70000
80000
90000
Amount of unions


100.000 elements
UF
QUF
Time (Seconds)
0
5
10
15
20
25
30
10000
20000
30000
40000
50000
60000
70000
80000
90000
Amount of unions

1.2.1 Results for union find

The union find should grow linear and looking at the results it almost grows linear. Why it should grow linear is because we have to go throughout the whole list for every union thereby it completely relies on the number of unions and the length. Considering the time won't be fully accurate we can draw the conclusion that it is linear as expected.

1.2.2 Results for quick union find

The quick union find should grow faster the more unions it does and as you can see the time increases mor and more slowly.

1.2.3 Result discussion

If you compare the results the QUF is a lot faster than the UF but we can also see that QUF will increase and probably go slower than UF eventually.

2. 3Sum

2.1 Setup

The 3sum problems were solved by making two classes for the different solutions and a problem7 class that solves the seventh problem using a for loop and the timeit method. This is where I get the time results for running my algorithms.

2.1.1 3sum-Brute force

It works by checking through all the elements with three different for loops and checking if the sum computes to 0.

2.1.2 3Sum with caching

The 3sum with caching works by checking for complements for the number. This means it must not check through everything each time. Instead, it stores it in two different caches making the next 3sum go faster.

2.1.3 Calculations setup

I have used to different methods for calculating the fitted and the computed curve. The fitted curve uses “a * x^b“ while the computed curve uses “(x * 2^(aComp / bComp))^bComp”

For getting the a and b values for the fitted curve I use the built-in method in python called curve_fit.

For getting the a and b values for the computed curve I used:

```
aCompBrute = np.log2(time3sum[6]) - bCompBrute * np.log2(sizeAmount[6])
aCompCache = np.log2(time3sumcache[6]) - bCompCache * np.log2(sizeAmount[6])
```

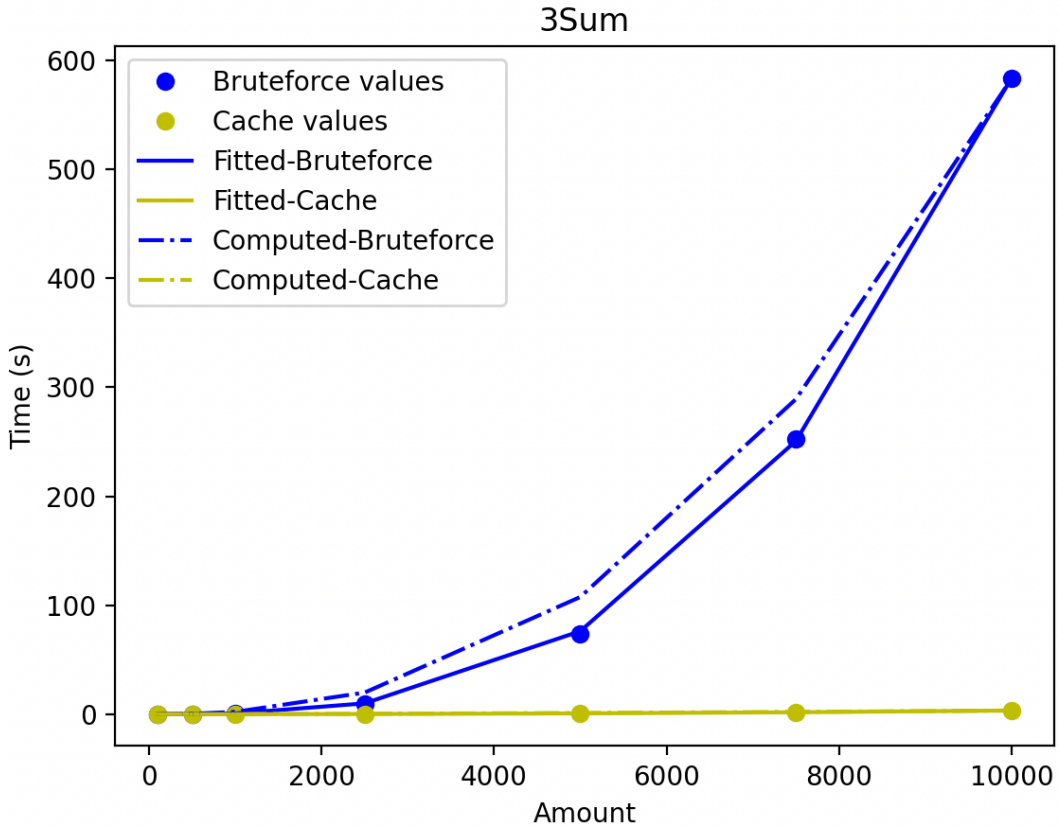
```
def computedBValue(time, size):
    logTime = np.log2(time)
    logSize = np.log2(size)

    b_val = ((logTime[6] - logTime[0]) / (logSize[6] - logSize[0]))
    return b_val
```

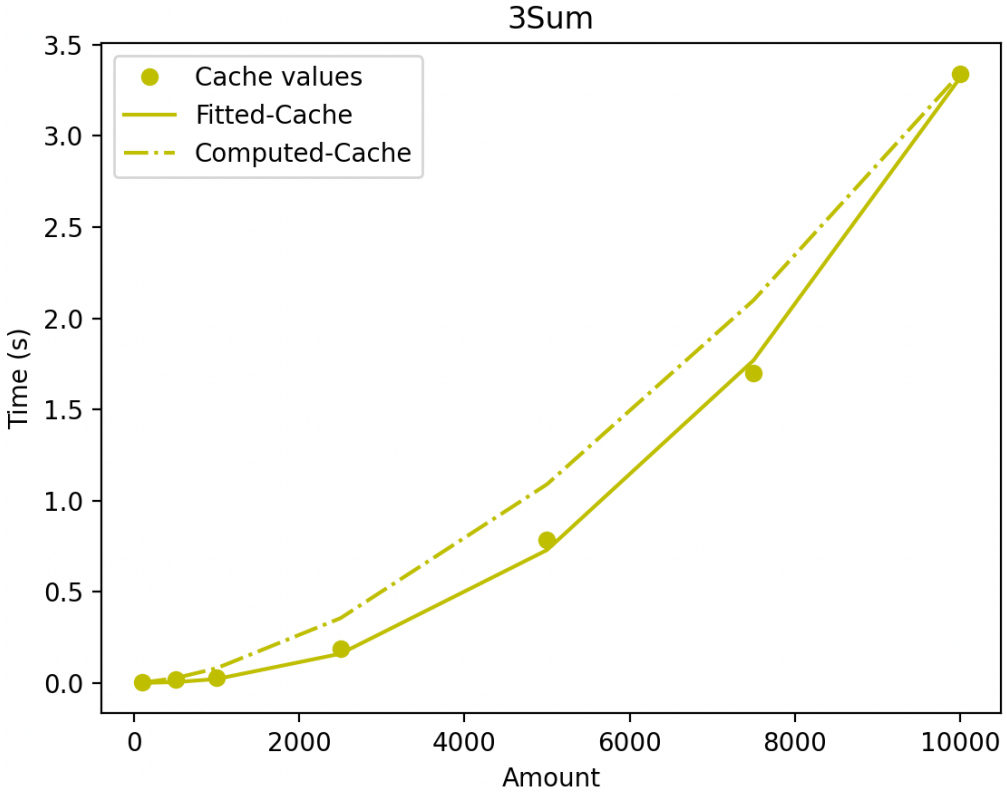
2.2 Results

```
sizeAmount = np.array([100, 500, 1000, 2500, 5000, 7500, 10000])
time3sum = np.array([0.007715625, 0.108891459, 0.602767166, 9.167325667, 73.820756459, 252.517078708, 583.159775042])
time3sumcache = np.array([0.00195325, 0.017687792, 0.029005958, 0.189759417, 0.783339083, 1.701242208, 3.341058083])
```

```
A values:
Fitted: Brute; 1.033306390167447e-09 | Cache; 5.96547470584726e-09
Computed: Brute; -23.223749916052554 | Cache; -19.740120404528884
B values:
Fitted: Brute; 2.9379790995475683 | Cache; 2.186317216603219
Computed: Brute; 2.4392082240713555 | Cache; 1.616563096381705
```



Here is a zoomed in picture of the first graph displaying that there is a growth in the 3sum with cache even if it is small compared to brute force variant.



2.2.1 Result discussion for 3 sum-Brute force

The brute force variant as the linear fit suggests grows by approximately $O(N^3)$ as seen in the growth value in the power linear fit. But checking the computed value it is closer to a $O(N^2)$ almost.

2.2.2 Result discussion for 3sum with caching

The caching should make the 3sum algorithm quicker and it also does. It instead is $O(N^2)$ as seen in the power linear fit. The computed suggests something less than $O(N^2)$.

2.2.3 Result discussion for 3sum

We can clearly see that the 3sum with caching is a lot quicker than the brute force variant. This can be seen in both the data provided by the tests made and by the mathematical models calculated with values. The growth increases a lot when brute forcing instead of using caching.