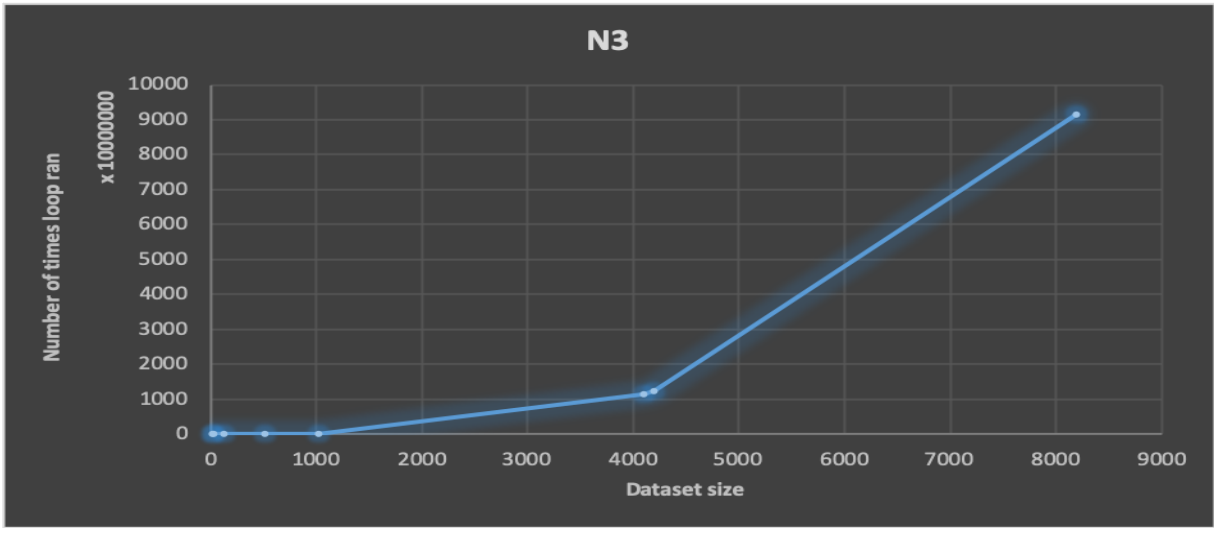
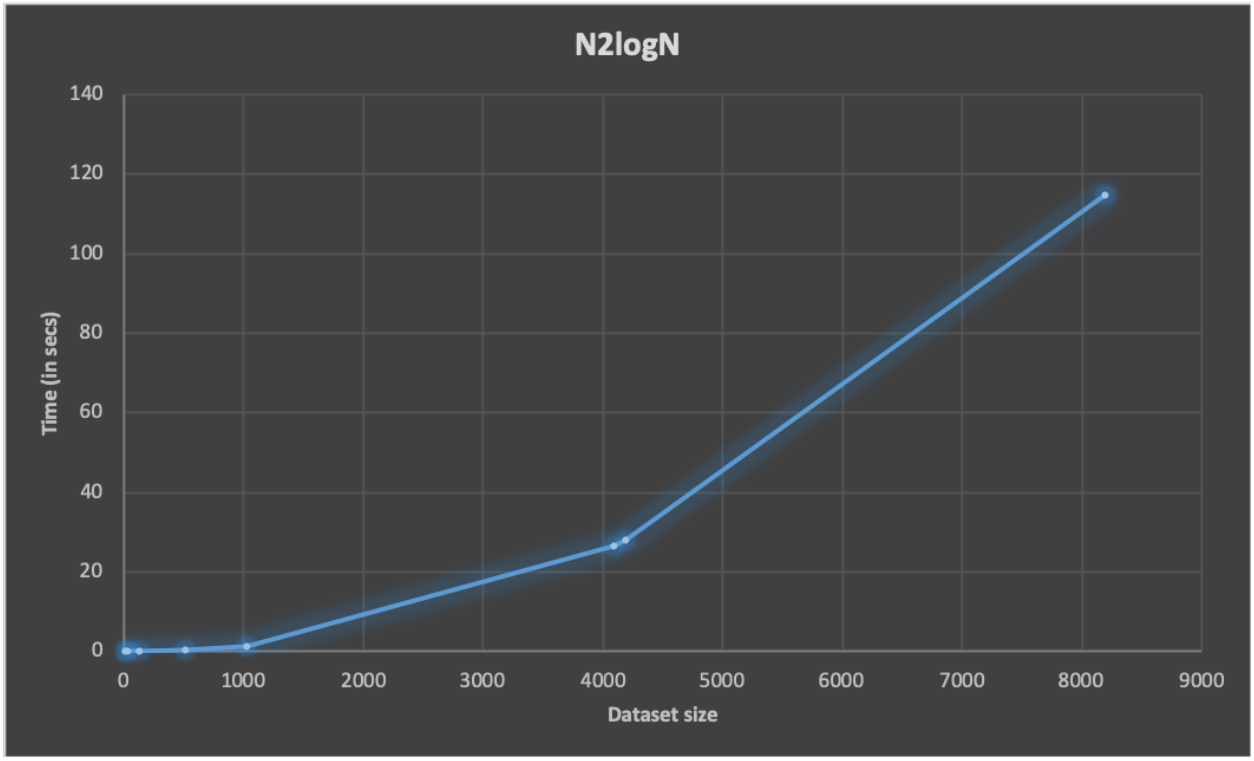
Data Structures and Algorithms
ECE 16:332:573 Spring 2020
Shounak Rangwala
netID:snr85

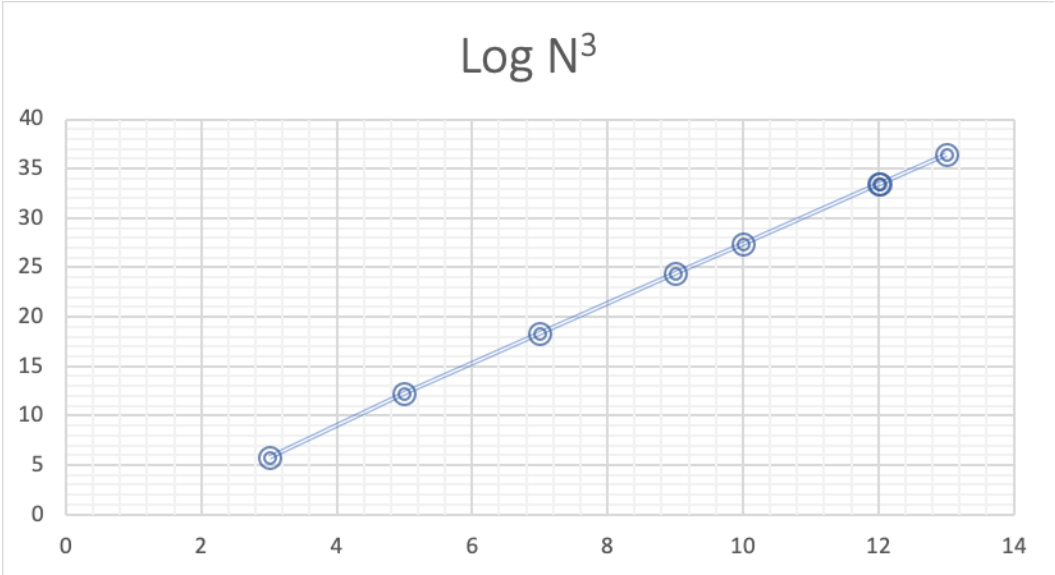
Q1)
For the naïve implementation of the $O(N^3)$ complexity and the sophisticated implementation of $O(N^2 logN)$ complexity, I made a python script which would read the values of the dataset into a list. I would then loop over the list. To calculate the time taken, I used the time library. I started the time just before I started the loop and ended it after the count of 3 sum possibilities were found. This way I could calculate the time it took to run the logic for the algorithm only and not the other superfluous stuff.
I made an observation that the time taken by the computer to run a script would take different times for different instances so I decided to take an average of 2 times and report it in the graph to make the plot.
For the naïve implementation, I used a count variable which counts the number of times the loops ran, instead of the timer because the at that time I felt that there may be a difference in the slope and using hypothesis would be easier in counter format, but there was no difference. The time shown in the plot is measured in seconds.

Data Size	N^3	$N^2 logN$
8	56	0.0
32	4960	0.0
128	341376	0.0360000133514
512	22238720	0.3109998703
1024	178433024	1.38300013542
4096	11444858880	26.4830000401
4192	12268800000	28.0069999695
8192	91592417280	114.809999943

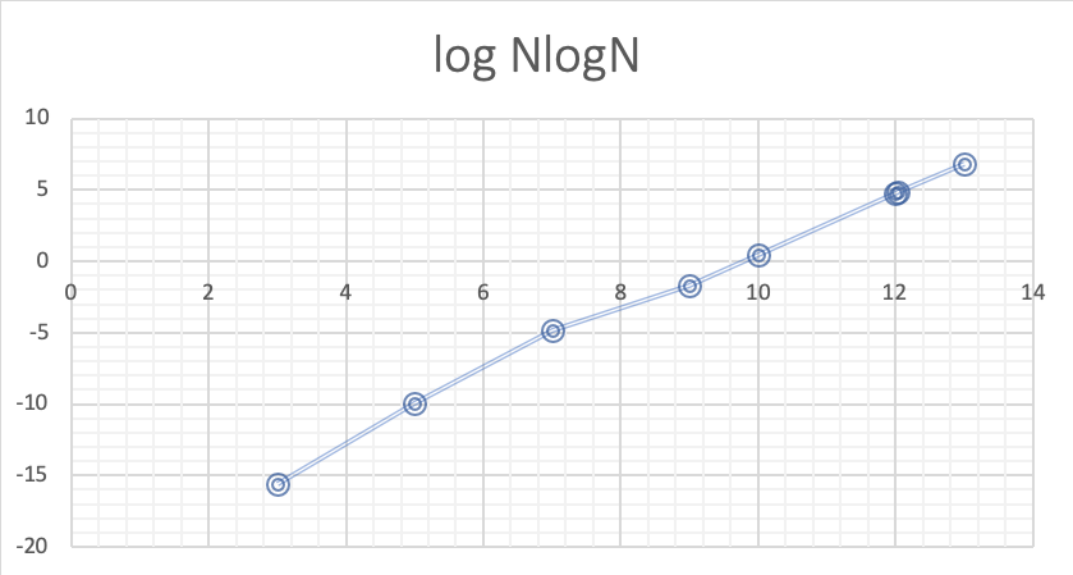




As you can see from the Log- log chart of N^3 complexity, the value for x= 10 is 27.41080741 and for x=12 is 33.4139802. This gives the slope of the curve to be almost exactly 3 which is what is expected.



In the below Log Log plot for the N^2logN plot, the expected slope should be around 2. This can be confirmed by taking the value at x = 10 which is 0.467801298 and x= 12 which is 4.897715549. The slope is seen to be almost 2.2 which can fit with our hypothesis of the complexity being N^2log N.

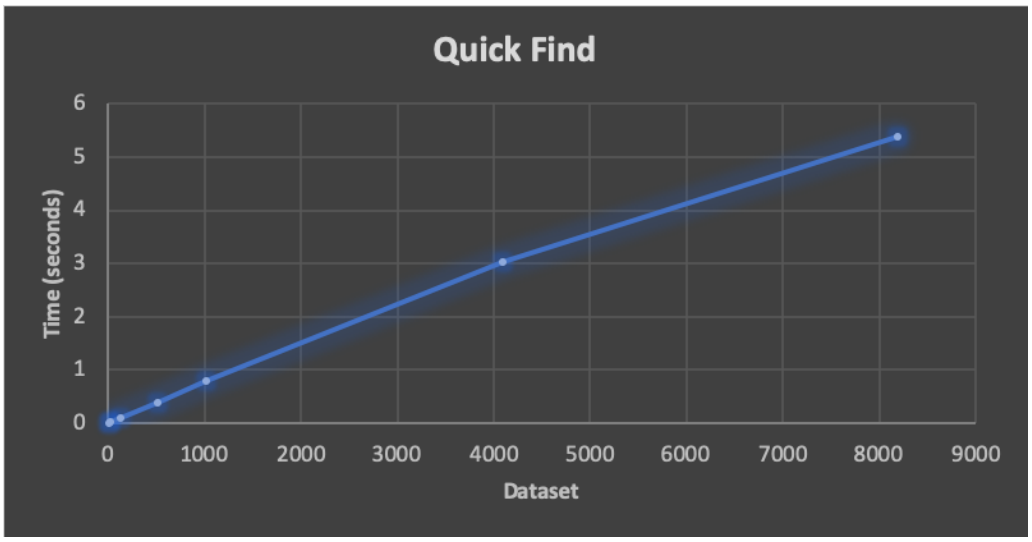


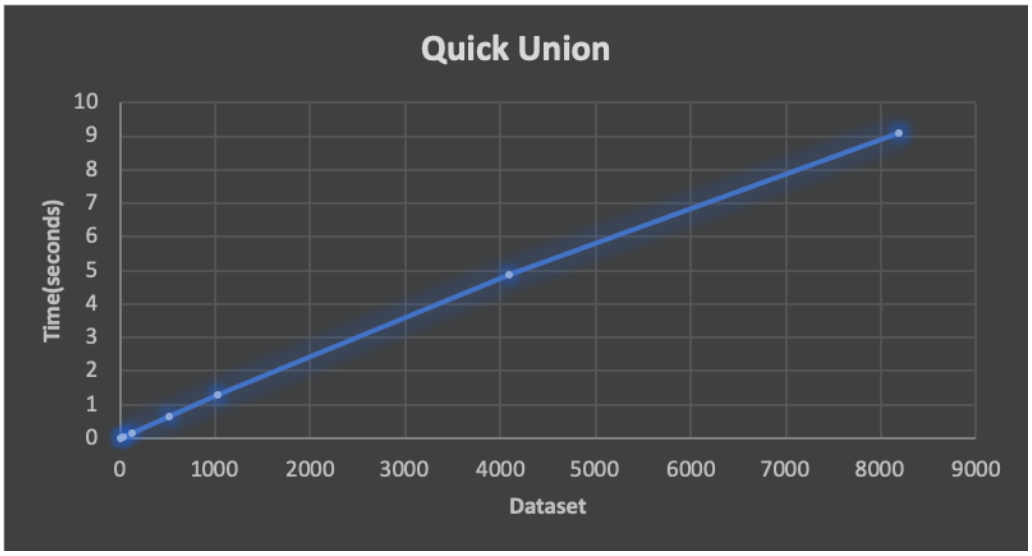
Q2)

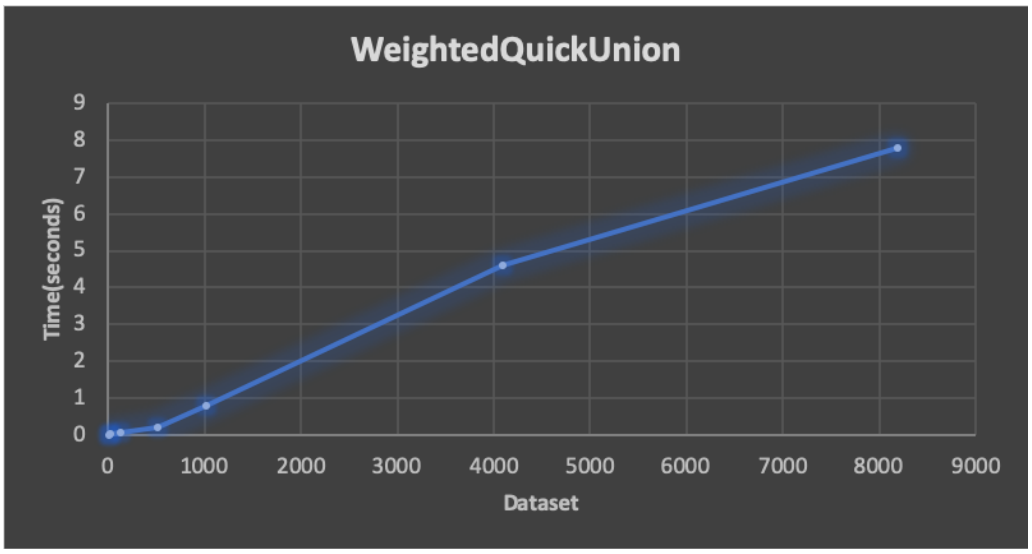
For this question, I followed the similar strategy of reading the dataset values from the text file into 2 lists called pArr and qArr. I tried to stick to the same logic for the algorithm as discussed in the slides. I also set up the counter before and after the main execution of the algorithm so I can calculate the time it took to run the code. Since the time varied for different instances, I took the average of 2 readings to make the plot.
The time is measured in seconds.

Data	Quick find	Quick union	Quick union weighted
8	0.006382942199707031	0.0100016593933	0.00399994850159
32	0.022845983505249023	0.0399994850159	0.0140001773834
128	0.09750771522521973	0.160000324249	0.0520000457764
512	0.38894081115722656	0.629999637604	0.1945688000077
1024	0.7862179279327393	1.27999782562	0.8000943333342
4096	3.0353012084960938	4.87999916077	4.5885447000006
8192	5.3907599449157715	9.10999774933	7.800000228799

In the quick find case: I think that the complexity is O(k*N) where k is the number of finds to be done before the union. The reason is that there is no clear hypothesis that can be observed from the data or the log plot of it. The worst-case scenario id $O(N^2)$ but this is not shown in this data set because it may be a good dataset.
Quick union: observing the complexity it can be observed to be almost linear rather than the N^2 complexity of the worst case scenario.
Quick union weighted: I maintained another array in my script with the size of the tree in it. The performance of this was better than Quick Union as expected but was not as good as Quick find. The complexity of this case is O(NlogN) in the worst case. N = 8192.


Quick Find
Time (seconds)
0
1
2
3
4
5
6
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
Dataset


Quick Union
Time(seconds)
0
1
2
3
4
5
6
7
8
9
10
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
Dataset


WeightedQuickUnion
Time(seconds)
0
1
2
3
4
5
6
7
8
9
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
Dataset

Q3)
Big Oh marks the upper limit of the growth function
For the Q1 naïve solution:
$g(N) = N^3$
let cf(N) should be some value that cf(N)<g(N) for all N>Nc. Nc should be 3 because we are talking about 3-sum problem. More accurate analysis can be done using math models.
f(n) should be ideally N(N-1)(N-2)/6 because its literally selecting 3 elements from N objects.
For any value of c>=1, cf(N) is greater than g(N).
For the sophisticated solution:
$g(N) = N^2 \log N$
f(N) = N(N-1)logN.
This is because, we are selecting 2 elements from the N elements and running binary search on the rest of the elements.

I think that given different variations of c there can be different values of Nc, but assuming that c=1, we can use the assumption we used in the earlier example that Nc= 3.

For the Q2:
Quick find and Quick union solutions have an $O(N^2)$ complexity. Because of their quadratic complexity we can assume the f(n) to be ax^2+bx+c format. Weighted quick union has a complexity of O(NlogN). It should be the fastest of the three, but it is difficult to assume any random value of Nc for these growth functions.
Since the problem depends on comparing 2 values of random integers, finding if they are connected or not and then connecting them so that the record can be maintained in the array; we can assume that Nc should be at least equal to or more than 2 so that a comparison can be made. It is difficult to make concrete hypothesis because it would take finding the exact equation of the growth function.

Q4)
I defined a function that has two variables, lowest and highest. I would pass an array of unsorted integers into this function to get the maximum distance from. Both lowest and highest would be initialized to the first element of the list.
In the for loop, I would go through all the entries in the list and make comparisons at each iteration. These comparisons would change the value of highest and lowest in a way such that at the end of the loop highest-lowest would be equal to the maximum distance in the array.

At each step, there would be a comparison and a reassignment. Worst case is also the best case scenario where there are 2 comparisons and 1 assignment so basically 3 operations each time. The complexity would be O(3N).

Q5)
Given a sorted array, my function would initialize pointers at the first and the last elements of the list and run the loop over the N elements of the array. I would compare the addition of these 2 pointers, if the result was positive, then I would shift the pointer to the last element to

the left by 1. If the result was negative, I would shift the pointer to the first element to the right by 1. If the result was 0: I would shift both the elements by 1 towards the inside of the array. This way,
Best case scenario : includes complimentary pairs only which makes the loop run only N/2 times.
Worst case: no complimentary pairs, loop runs maximum of N times.

Part 2:
To get a $O(N^2)$ complexity for 3-sum problem, I would first implement a sort of the array of unsorted elements. This would take $O(N^2)$ complexity because it is basic selection sort in the worst case scenario, which is also the best case scenario.
In the main implementation of the function, I would loop through the sorted array. For each element in the array I would search for a pair in the rest of the array whose sum would be equal to the compliment of the element. This way the sum of the triplet would be 0.

This implementation would take O(N*N) complexity (N for the loop and N for the searching of the number of pairs in the rest of the array at each iteration). Thus the total worst case complexity of the algorithm would be:
Sorting+searching
$O(N^2) + O(N^2) = O(N^2)$.

The data taken to verify includes a hardcoded array that addresses all border conditions.