ANALYSIS REPORT FOR ASSIGNMENT-1

NAME- MAREESH KUMAR ISSAR
NET ID- mi251
COURSE- 16:332:573 DATA STRUCTURE AND ALGORITHMS

ANSWER 1

3-SUM "NAÏVE" IMPLEMENTATION (O(N^3))

The data collected from the 3-Sum "Naive" implementation program is given in Table 1 along with its plot (Figure:1).

N	TIME (in seconds)
8	0.000001
32	0.000049
128	0.006768
512	0.173750
1024	1.390805
4096	91.545063
4192	95.631274
8192	740.779685

Table:1

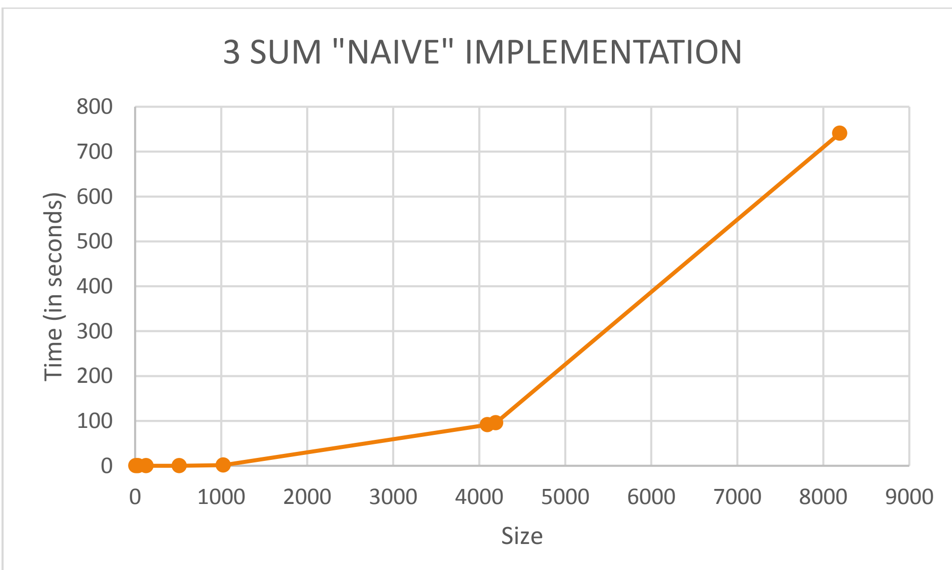


Figure:1

3-SUM "SOPHISTICATED" IMPLEMENTATION (O(N^2 lg N))

The data collected from the 3-Sum "Sophisticated" implementation program is given in Table 2 along with its plot (Figure:2).

N	TIME (in seconds)
8	0.000000
32	0.000021
128	0.000525
512	0.011703
1024	0.053593
4096	1.071119
4192	1.140989
8192	4.705654

Table:2

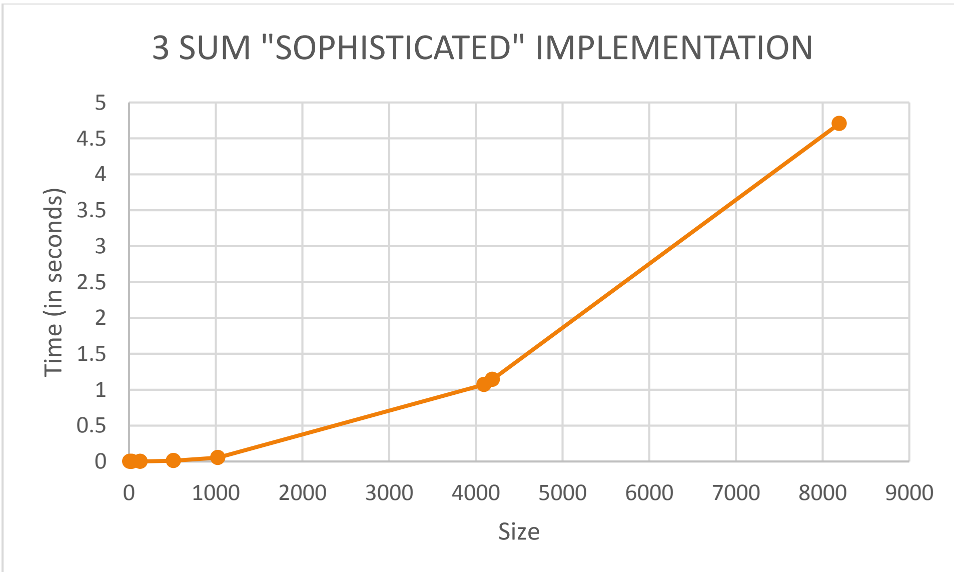


Figure:2

Table:3 shows the data comparing the two implementations (in log base 2 scale), while Figure:3 represents this data graphically. In Figure:3 orange line depicts the time for the “Naïve” implementation while red line depicts the “Sophisticated” implementation.

To find the runtime cost of each implementation as a function of the input we use the following equations:

$$\lg(T(N)) = b\lg(N) + c$$

$$T(N) = aN^b \quad , a = 2^c$$

Here T(N) is the running time and N is the input size.

To find the value of a and b, we use the equation of a line passing through two points:-

$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$$

Here, (x_1,y_1) and (x_2,y_2) are the two points.

For "Naïve" implementation- using(9,-2.525) and (10,0.476) we get b=3.001 and c= -29.534.

$$T(N) = 1.286 \times 10^{-9} N^{3}$$

For "Sophisticated" implementation- using(9,-6.417) and (10,-4.222) we get b=2.195 and c= -26.172.

$$T(N) = 1.322 \times 10^{-8} N^{2.195}$$

From the above computed T(N) values, we find that the "Sophisticated" implementation gives us the best performance for large input size. This is evident from the equation of T(N) (in case of "Sophisticated" implementation) as it has the least value for the exponent of N.

N (lg scale)	TIME in seconds (lg scale) "Naïve"	TIME in seconds (lg scale) "Sophisticated"
3	-19.932	-20.084
5	-14.317	-15.539
7	-7.207	-10.895
9	-2.525	-6.417
10	0.476	-4.222
12	6.516	0.0991
12.03	6.579	0.19
13	9.533	2.234

Table:3

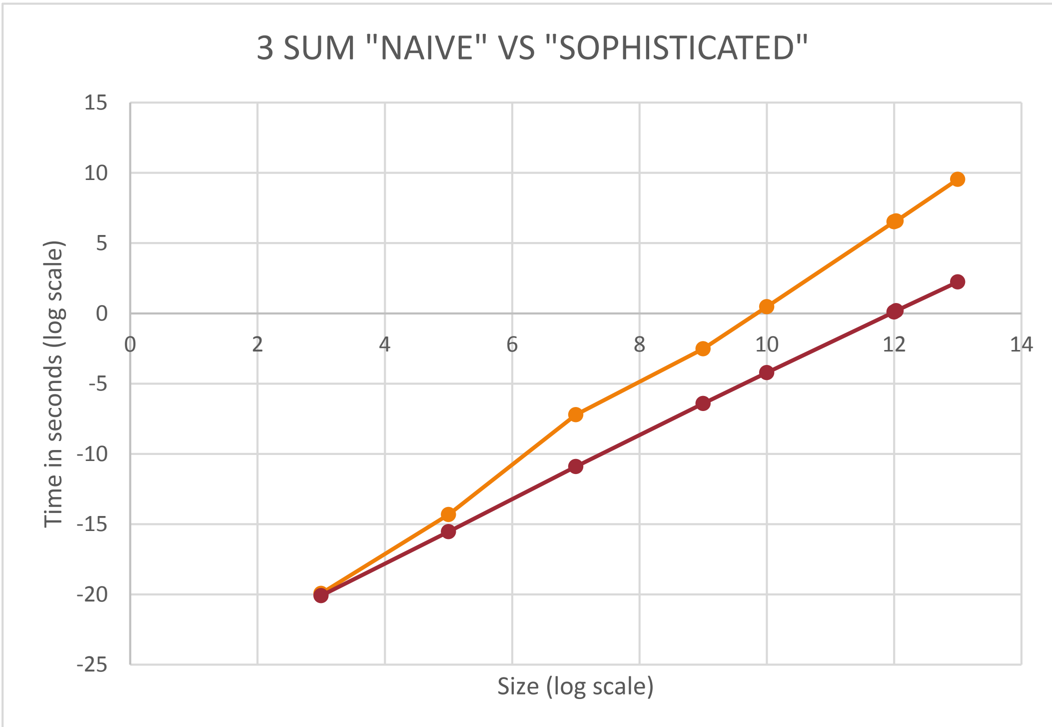


Figure:3

ANSWER 2

QUICK-FIND

NOTE:- While calculating the execution time, the cout statement printing the union pair was commented.

The data collected from the Quick-Find implementation program is given in Table 4 along with its plot (Figure:4).

N	TIME (in seconds)
8	0.000576

32	0.001527
128	0.009134
512	0.019094
1024	0.038094
4096	0.158546
8192	0.284547

Table:4

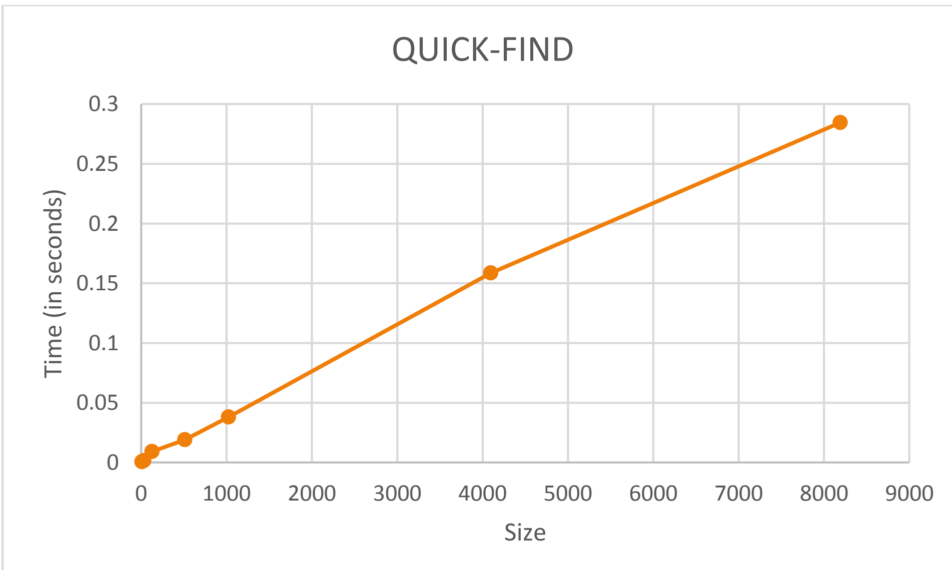


Figure:4

QUICK-UNION

The data collected from the Quick-Union implementation program is given in Table 5 along with its plot (Figure:5).

N	TIME (in seconds)
8	0.000241
32	0.000247
128	0.000248
512	0.000287
1024	0.000372

4096	0.000918
8192	0.011915

Table:5

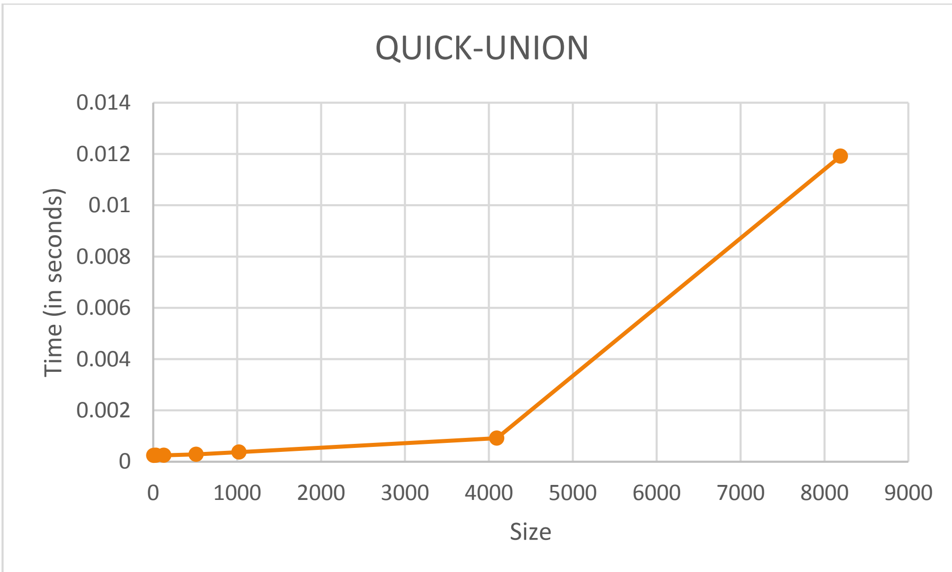


Figure:5

QUINCK-UNION WITH WEIGHT BALANCING

The data collected from the Quick-Union with weight balancing implementation program is given in Table 6 along with its plot (Figure:6).

Table:6

N	TIME (in seconds)
8	0.000384
32	0.000401
128	0.000408
512	0.000454
1024	0.000526
4096	0.001050
8192	0.001815

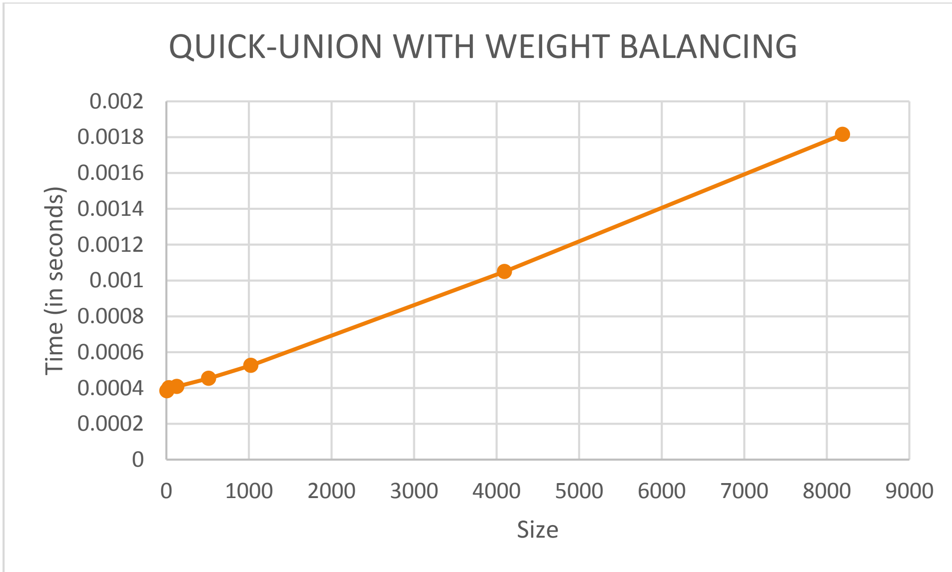


Figure:6

Table:7 shows the data comparing the three Union-Find implementations (in log base 2 scale), while Figure:7 represents this data graphically. In Figure:7 orange line depicts the time for the Quick-Find implementation, red line depicts the Quick-Union implementation and the blue line represents the Quick-Union with weight balancing implementation.

To find the runtime cost of each implementation as a function of the input we use the following equations:

$$\lg\left(T\left(N\right)\right) = b\lg\left(N\right) + c$$

$$T\left(N\right) = aN^{b} \quad , a = 2^{c}$$

Here T(N) is the running time and N is the input size.

To find the value of a and b, we use the equation of a line passing through two points:-

$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$$

Here, (x_1,y_1) and (x_2,y_2) are the two points.

For Quick-Find implementation- using(12,-2.657) and (13,-1.813) we get b=0.844 and c= -12.785.

$$T\left(N\right)=1.417\times10^{-4}N^{0.844}$$

For Quick-Union implementation- using(12,-10.089) and (13,-6.391) we get b=3.698 and c= -54.465.

$$T\left(N\right)=4.022\times10^{-17}N^{3.698}$$

For Quick-Union with weight balancing implementation- using(12,-9.895) and (13,-9.106) we get b= 0.789 and c= -19.363.

$$T\left(N\right)=1.483\times10^{-6}N^{0.789}$$

From the above computed T(N) values, we find that the Quick-Union with weight balancing gives us the best performance (amongst the three implemented methods) for large input size. This is evident from the equation of T(N) (in case of Quick-Union with weight balancing) as it has the least value for the exponent of N.

N (lg scale)	TIME in seconds (lg scale) Quick-Find	TIME in seconds (lg scale) Quick-Union	TIME in seconds (lg scale) Quick-Union with weight balancing
3	-10.76164357	-12.01867923	-11.34660607
5	-9.355084223	-11.98320134	-11.28411014
7	-6.774537495	-11.97737226	-11.25914323
9	-5.710736825	-11.76666164	-11.10502008
10	-4.714292406	-11.39240976	-10.89264958
12	-2.657026615	-10.08921823	-9.895394957
13	-1.813261126	-6.391077238	-9.105814736

Table:7

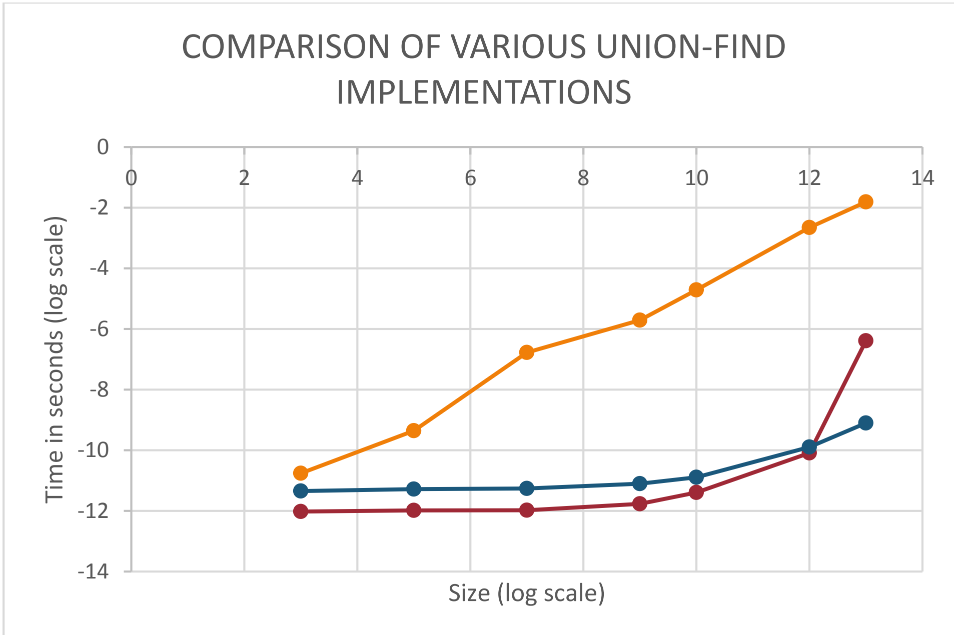
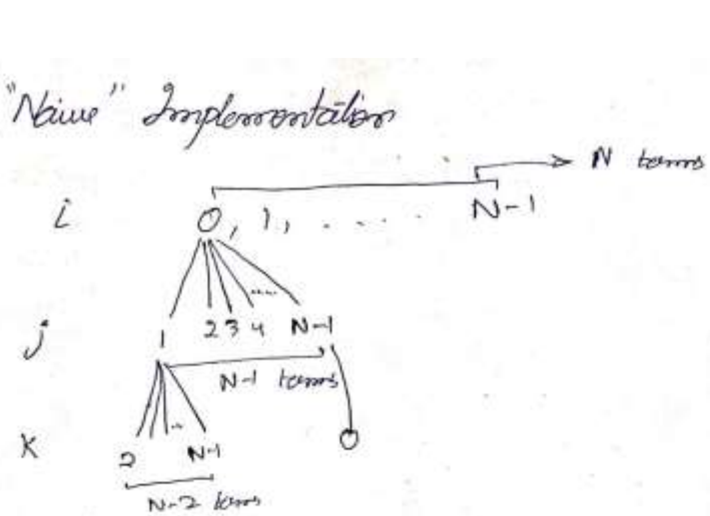


Figure:7

ANSWER 3

For 3-Sum "Naive" Implementation

We have



∴ for a particular value of i we have

$$N-2+N-3 \dots +1+0 \quad \text{terms}$$

$$= \frac{(N-2)(N-1)}{2} \sim \frac{N^2}{2}$$

∴ for N, i terms we have $\sim N \times \frac{N^2}{2} \sim \frac{N^3}{2}$ array access

Now, using the defination of "Big-Oh" $\boxed{T(N) \text{ is } O(f(N)) \text{ if } T(N) \le c f(N) \ \forall N \ge n_0}$

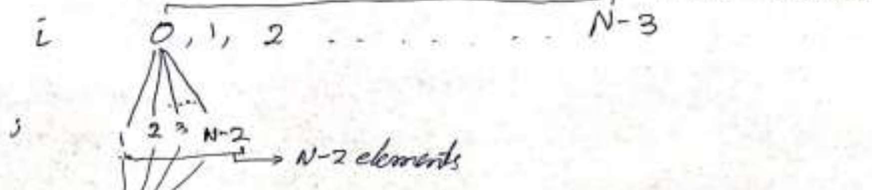
$$\boxed{\frac{N^3}{2} \le 2N^3 \quad \text{for } N \ge 1}$$

$\boxed{\text{Here, we assume } n_0 = 1 \text{ and } c = 2}$

We can clearly see that the above in equation if $N \ge 1$ then ~~N≥3~~ the above equation is true. ∴ 3 Sum "naive" implementation is $O(N^3)$ $\boxed{N_c = 1}$

2. For 3-Sum "Sophisticated" Implementation

We have



K $\lfloor \log_2 N+1 \rfloor$ → worst case - maximum no. of array access done by binary search

∴ for a particular i value we have

$$\lfloor \log_2 N+1 \rfloor \; N-2 \text{ terms}$$

$$\sim N \log_2 N \text{ terms}$$

∴ for $N-2$ i terms we have

$$\sim (N-2) \times N \log_2 N$$

$$\sim N^2 \log_2 N \text{ array access}$$

Now, using the definition of "Big-Oh" we have

$$\boxed{N^2 \log_2 N \leq 2 N^2 \log_2 N \quad \text{for all } N \geq 1}$$

where $c = 2$
$n_0 = 1$

We see that the above eq. is always true as terms on both the sides are positive for $N \geq 1$ $\boxed{N_c = 1}$

1. For Quick-Find implementation

The connected / find algorithm takes 1 array access for a single object ∴ For N objects, N array accesses

The union algorithm goes through each N elements for a single object ∴ For N objects it takes N^2 array accesses

Combining the above algorithms take $N^2 + N$ array access in total

Now, using the definition of "Big-Oh" we have

$$\boxed{N^2 + N \leq 2N^2 \quad \text{for all } N \geq 1}$$

where $c = 2$
$n_0 = 1$

when $N \geq 1$ we know that $N \leq N^2$

$$\therefore N^2 + N \leq N^2 + N^2$$

$\boxed{N_c = 1}$

For Quick-Union implementation

The connected / find algorithm takes N array accesses in the worst case (when there is a single chain) for 1 object.
∴ for N objects, N^2 array accesses

The union algorithm takes 1 array access but after finding the roots. In the worst case (single chain) it could take N array accesses to find the root for a single object.
∴ For N objects, N^2 array accesses (including the cost of finding the root)

Combining the above algorithms takes N^2+N^2 array accesses in total

Now, using the definition of "Big-Oh" we have

$$\boxed{2N^2 \leq 4N^2 \quad \text{for all } N \geq 1 \quad \text{where } c=4,\ n_0=1}$$

where for $N \geq 1$ the above inequality always hold

3. For Quick-Union with weight balancing implementation

The connected / find algorithm takes log N array accesses for 1 object. This is due to the fact that depth of any node is at most log N. ∴ for N objects it takes N log N array accesses

The union algorithm takes 1 array access but after finding the roots. Since the depth of any node is at most log N, it could take log N array accesses. ∴ for N objects, N log N array accesses

Combining the above algorithms take $N\log N + N\log N$ array accesses in total

Now, using the definition of "Big-Oh" we have

$$\boxed{2N\log N < 4N\log N \quad \text{for all } N \geq 1 \quad \text{where } c = 4,\ n_0 = 1}$$

$\boxed{N_c = 1}$

where for $N \geq 1$ the above inequality holds

NOTE :- For this implementation, I have not considered the operations required for initialization of the array. If initialization is considered then there will be $N + 2N\log N$ array accesses in total and

$$\boxed{N[1 + 2\log N] < 4N\log N \quad \text{for all } N \geq 2 \quad \text{where } c = 4,\ n_0 = 2}$$