HW1

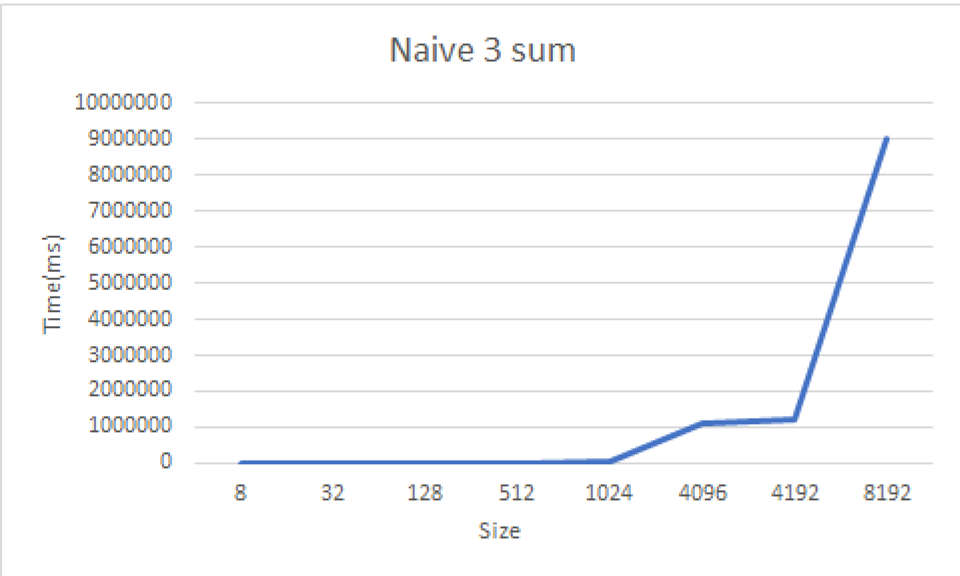
Q1

The runtime for the given data is as follow:

Size / Time(ms)	Naive 3 sum	Smart 3 sum
8	0.02	0.05
32	0.50	1.00
128	31.25	15.62
512	2109.38	406.25
1024	17062.50	1812.50
4096	1124015.63	35343.75
4196	1215734.38	39203.13
8192	9034207.23	155703.13

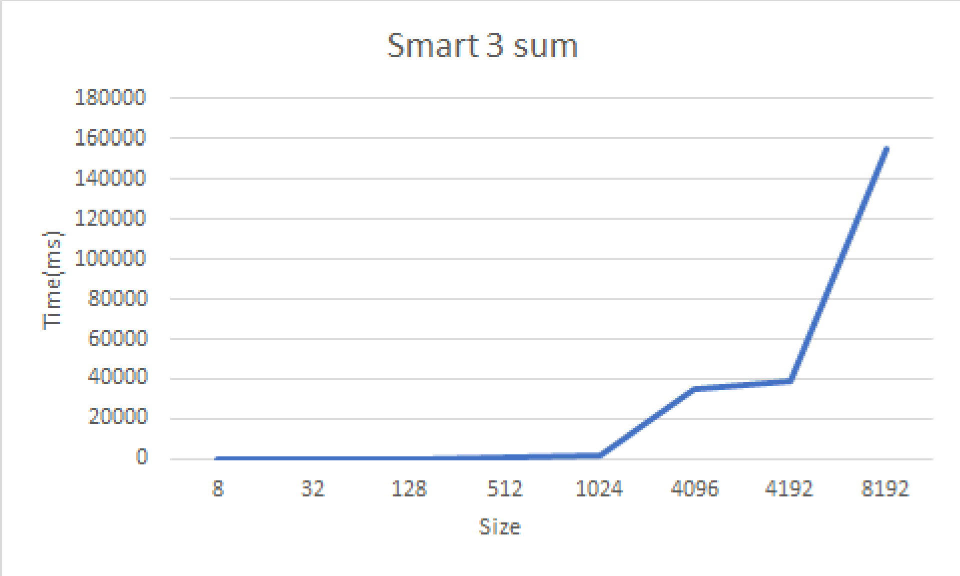
Naive 3 sum

When the size is double, the runtime is 2^3 times longer, e.g., when size is changed from 4096 to 8192, the runtime is 8.3 times longer, which shows it has $O(N^3)$ growth rate



Smart 3 sum

When the size is double, the runtime is $2^2 log2$ times longer, e.g., when size is changed from 4096 to 8192, the runtime is 4.4 times longer, which shows it has $O(N^2 logN)$ growth rate



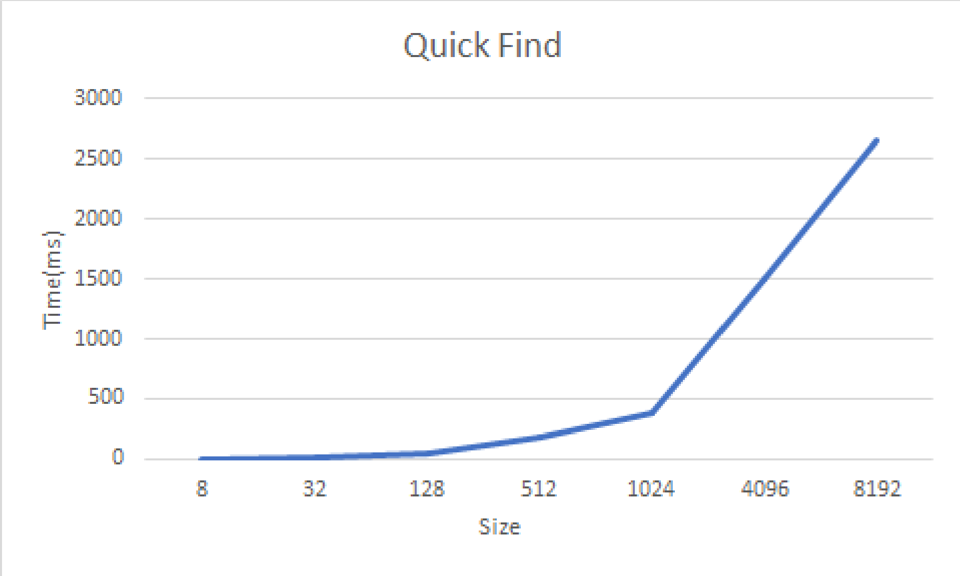
Q2

The runtime for the given data is as follow:

Size / Time(ms)	Quick Find	Quick Union	Weighted Quick Union
8	3.13	0.23	0.24
32	12.22	0.20	0.22
128	48.12	0.31	0.30
512	188.55	0.52	0.61
1024	386.26	0.89	1.02
4096	1503.79	3.47	3.81
8192	2659.27	14.10	7.97

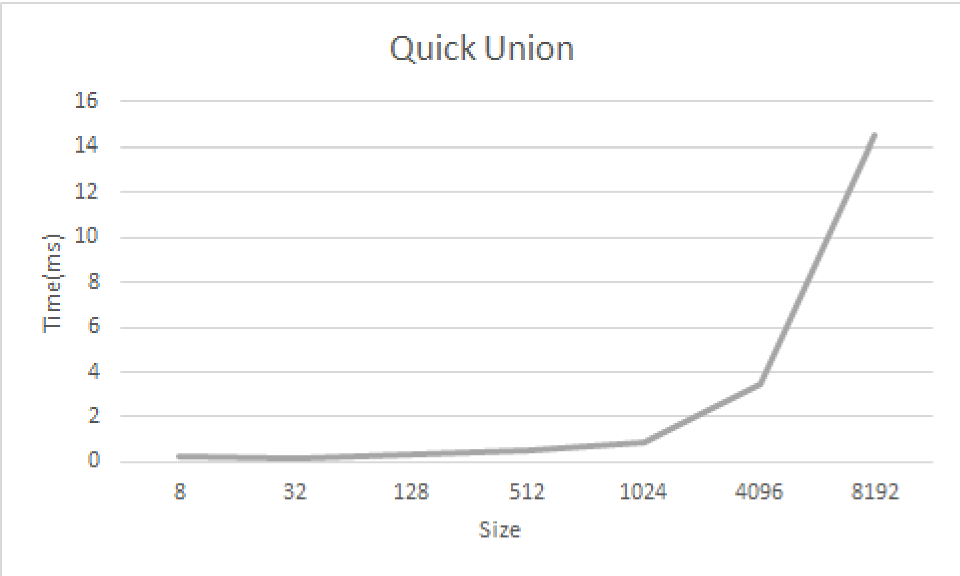
Quick find

union() is $O(N)$, **find()** is $O(1)$,and **read()** N data with each one conduct both **union** and **find**, that is $O(M * (N+1))$, so overall the algorithm has $O(N^2)$. For example, when size is changed from 1024 to 4096, the runtime is $3.89 \approx 4$ times,



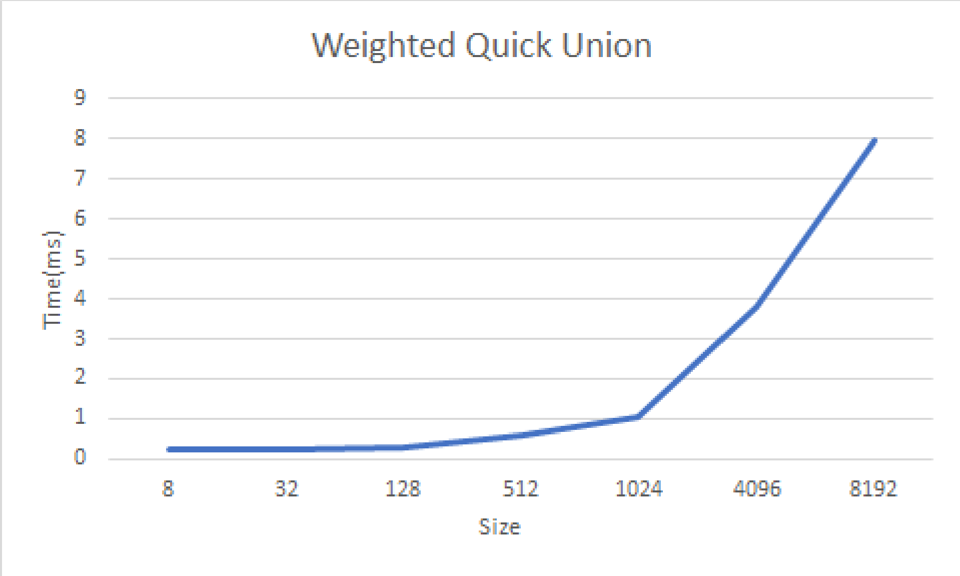
Quick union

The result shows a very small growth rate on small size data, but as the size grow, the growth rate is close to $O(N^2)$. The reason is, as more points are involved, the *root* would be "deeper" to reach.



Weighted quick union

Result shows that the runtime is better than original quick union. The grow rate is reaching $O(NlogN)$ as the size is greater.



Q3

I use **curve_fit(func,x,y)** from **scipy.optimize** (and **numpy**) to perform curve fitting on my test result.

Naive 3 sum

Hypothesis : $F(N) = a + bN^3$

I get :

$$a = -5.73887675e + 02$$

$$b = 1.64336470e - 05$$

So I let : $c = 1$ and $N_c = 1$.

Smart 3 sum

Hypothesis : $F(N) = a + bN^2 logN$

I get :

$$a = 7.50690982e + 01$$

$$b = 2.57614766e - 04$$

So I let: $c = 1$ and $N_c = 7$.

Quick Find

Hypothesis : $F(N) = a + bN^2$

I get :

$$a = 2.12797912e + 02$$

$$b = 3.88638493e - 05$$

So I let: $c = 1$ and $N_c = 15$.

Quick Union

Hypothesis : $F(N) = a + bN^2$

I get :

$$a = 3.31506992e - 01$$

$$b = 2.04184706e - 07$$

So I let: $c = 1$ and $N_c = 1$.

Weighted Quick Union

Hypothesis : $F(N) = a + bNlogN$

I get :

$$a = 2.45783432e - 01$$

$$b = 1.04681628e - 04$$

So I let: $c = 1$ and $N_c = 2$.

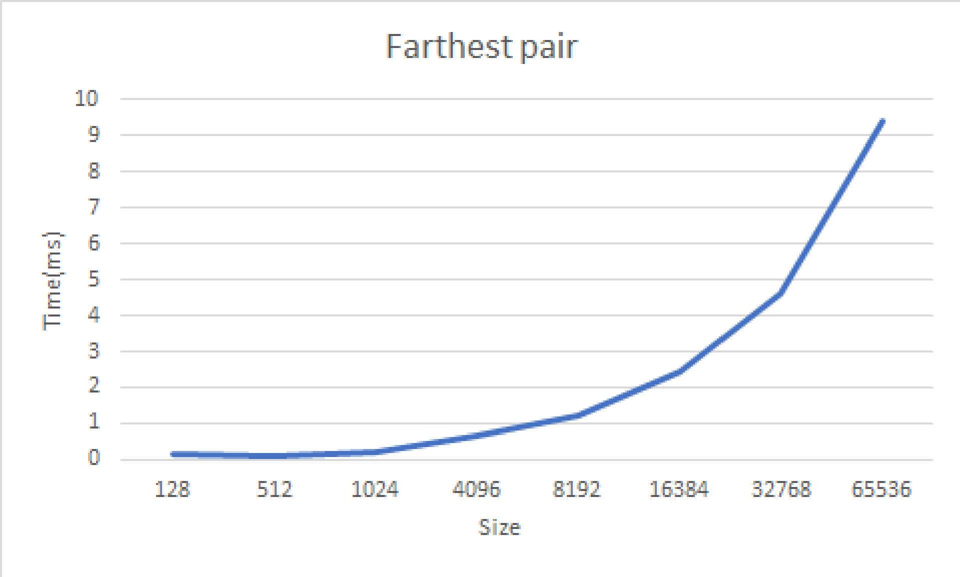
Q4

My implementation uses two local variants to store the maximum value and the minimum value during the iteration.

I use *numpy.random.randint(low,high,len)* from **numpy** to generate random data with different sizes to test my implementation.

Size/Time(ms)	Farthest pair
128	0.15
512	0.12
1024	0.20
4096	0.64
8192	1.24
16384	2.45
32768	4.60
65536	9.39

Result shows a $O(N)$ growth rate as required.



Q5

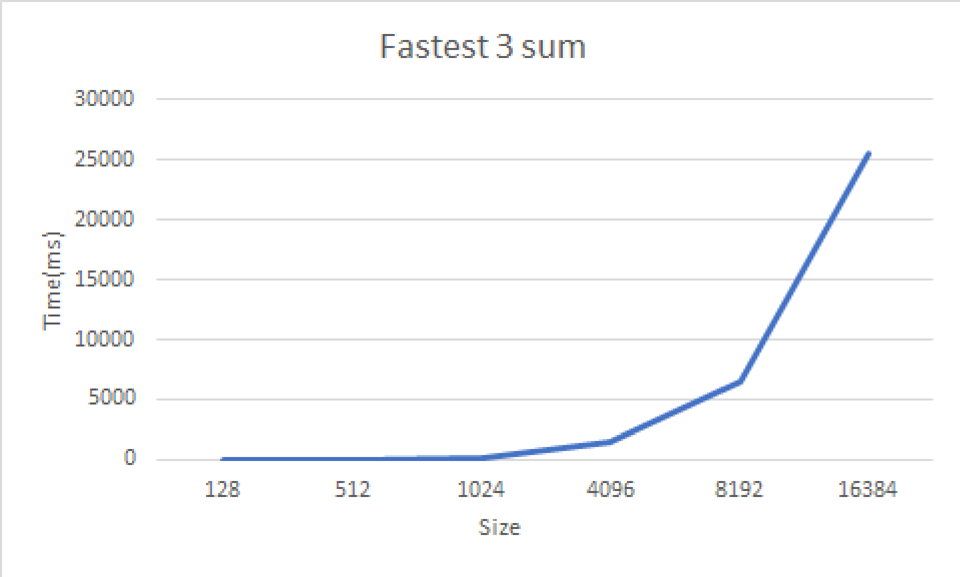
My implementation uses two local indexes, one for a smaller number going from the left to the right on the array, the other one represents a bigger number going from the right to the left.

By maintaining those two index, I can compare **(current num+ small num + big num)** with **0**, if it is greater, the right index decrease and choose a smaller **big num**, and if it is smaller, the left index increase and choose a bigger **small num**.

I use *numpy.random.randint(low,high,len)* from **numpy** to generate random data with different sizes to test my implementation.

Size/Time(ms)	Fastest 3 sum
128	0
512	15.63
1024	93.75
4096	1546.88
8192	6421.88
16384	25640.63

The result shows a $O(N^2)$ growth rate as required.


Fastest 3 sum
30000
25000
20000
15000
10000
5000
0
Time(ms)
128
512
1024
4096
8192
16384
Size