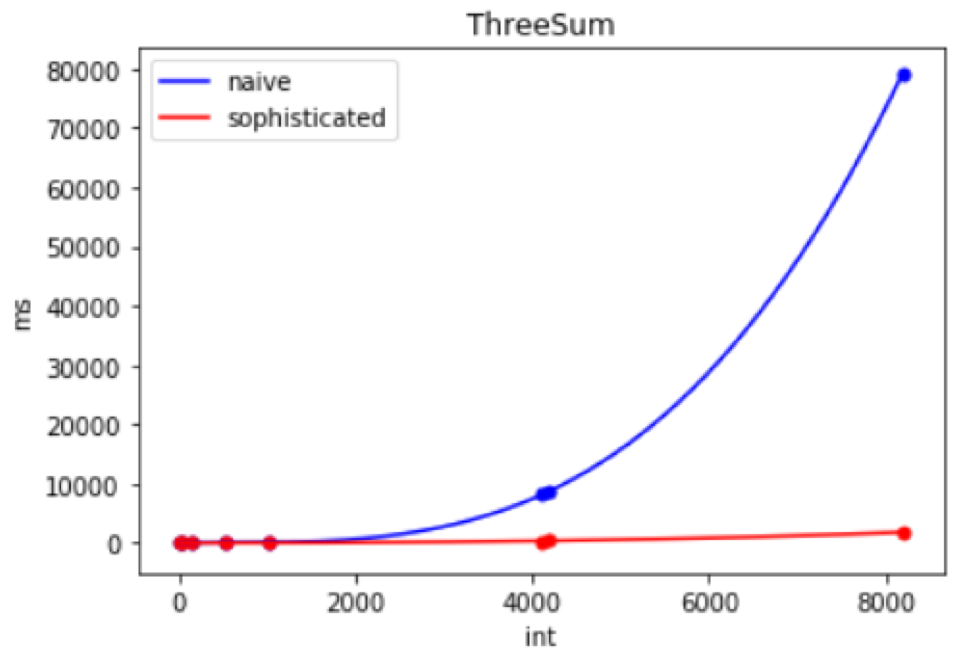
Data Structure & Algorithm Homework 1

Zhuohang Li

Q1.

Data / Time(ms) / Name	8	32	128	512	1024	4096	4192	8192
naïve	1	1	10	33	141	8155	8718	79196
sophisticated	0	1	3	11	28	385	402	1814



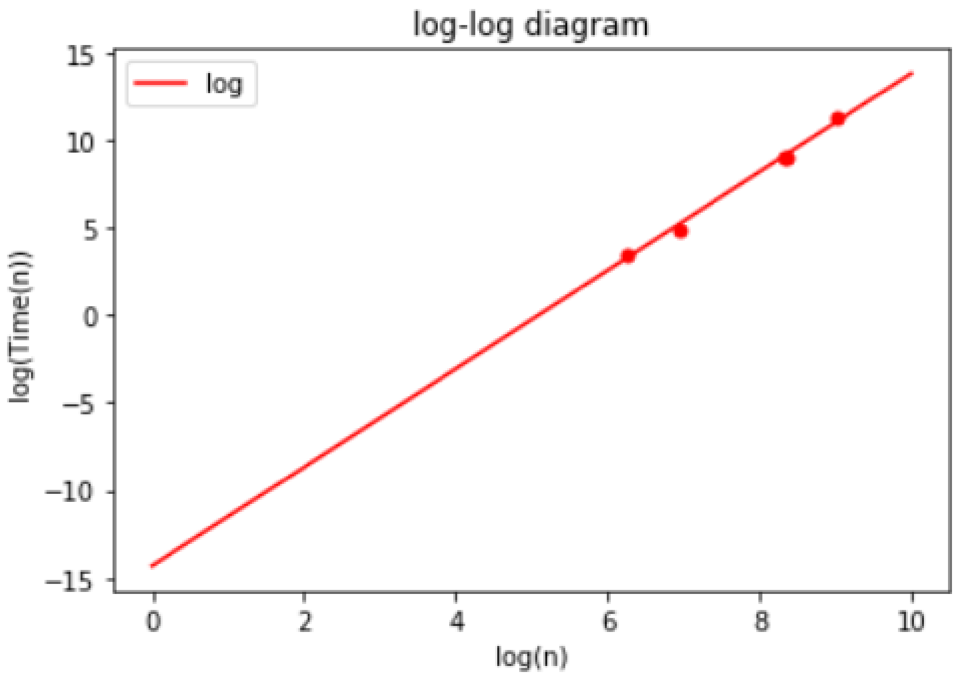
As is showed in the graph, the run time of the naïve implementation of three-sum problem is growing rapidly with data size. Eventually the sophisticated implementation is nearly 40 times faster as the naïve one when dealing with 8192 size of data. We know this is because naïve implementation, with the growth rate of $O(N^3)$, grows much quicker than sophisticated implementation, with the growth rate of $O(N^2 \log N)$.

```
public static int count(ArrayList<Integer> a) {
        int N=a.size();
        int count=0;
        for(int i=0;i<N;i++) {
                for(int j=i+1;j<N;j++) {
                        for(int k=j+1;k<N;k++) {
                                if(a.get(i)+a.get(j)+a.get(k)==0)
                                        count++;
                        }
                }
        }
        return count;
}
```

Naïve implementation

In the naïve implementation we are using a 3-level loop structure to search for triplets add to sum of zero. The outer loop starts from 0 to n-1 which means it iterates for n times. The inner loops do not have exactly n iterations but is also associated with n. Therefore, as result of arbitrary calculation, the total performance is related to $n^{3.}$

And indeed, if we do a log-log diagram of the run time and the data size, we get a line with the slope of roughly 3, which confirms the hypothesis.



```
slope= 2.8160125206492723
```

```
public static int count(ArrayList<Integer> a) {
        int N=a.size();
        int count=0;
        for(int i=0;i<N;i++) {
                for(int j=0;j<N;j++) {
                        if(binarySearch(a,-a.get(i)-a.get(j))>j)
                                count++;
                        }
                }
        return count;
}
```

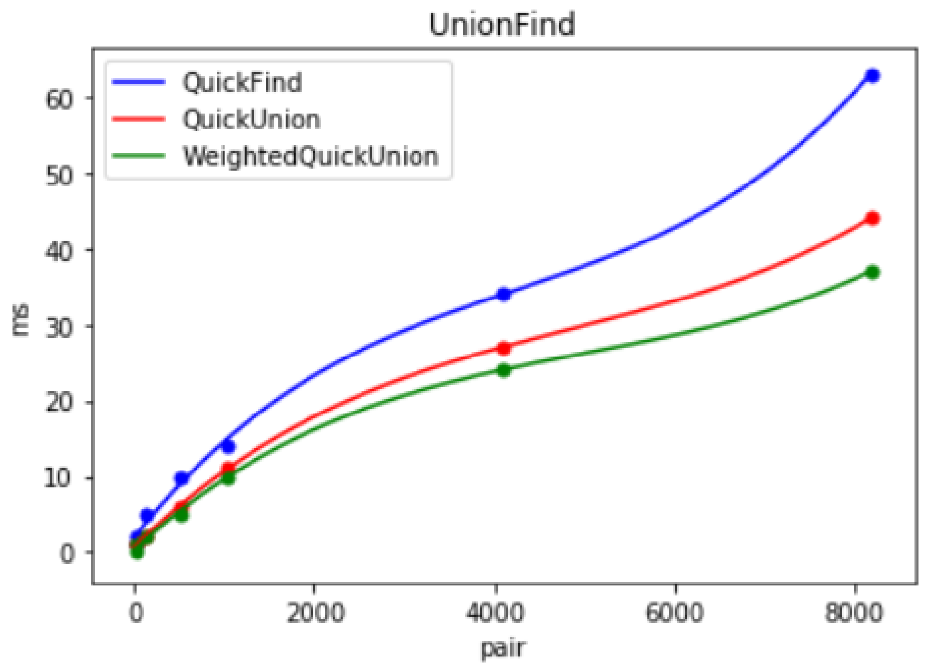
sophisticate implementation

The sophisticate implementation is using binary search to reduce the complexity to O(N^2logN).

The execution time of binary search will differ from run to run. The best-case is the target locates at the middle of the array, which can be find with only 1 operation. The worst-case is to search the whole array only to find the target is not in it, in which case the complexity is O(logN). Since all the given data given is positive, there is no such triple that sums to zero. That means it is always the worse-case scenario for the binary search. Consider the outer 2-level loop, for each multiplies the complexity by N, so we get a final complexity of O(N^2logN). Since the complexity of the sorting algorithms is less than O(N^2logN), so the overall performance of the whole algorithm is still O(N^2logN).

Q2.

Data / Time(ms) / Name	8	32	128	512	1024	4096	8192
Quick Find	1	2	5	10	14	34	63
Quick Union	1	1	2	6	11	27	44
Quick Union with weight	0	1	2	5	10	24	37



The test process is to union every given pair of components, which also includes find operation because we have to find if those components are already connected first before we decide to union them. *Quick Find* provides find() function with only one operation, but has to traverse the entire array to union. *Quick Union*, on the contrary, has to loop to find the root of a component when find() function is called, but only need one operation to union once the two component is found. *Quick Union with Weight* makes sure to attach the smaller tree to the big one every time we do union, and therefore increases the stability of this algorithm. From the diagram we can see *Quick Union with Weight* is slightly better than the other two union-find algorithms.

Q3.

1) For Q1:

For naïve implementation, the if statement is executed for every triplet. To count how many time IF statement is executed is equal to count how many distinct triplets we can select from N numbers. So total number F(n) will be:

$$\frac{n!}{3!\,(n-3)!} = \frac{n(n-1)(n-2)}{6} = \frac{n^3}{6} - \frac{n^2}{2} + \frac{n}{3}$$

So the C and g(n) will be:

$$\frac{1}{6}n^3$$

Solve F(n)-Cg(n)>0 to get:

$$n > \frac{2}{3}$$

Therefore, Nc=$\frac{2}{3}$

For sophisticated implementation, similarly, to count the number of time binary search is executed, is equal to count how many distinct pairs we can select from N numbers. That is:

$$\frac{n!}{2!\,(n-2)!} = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

In the worst case, binary search need log(n) operations to find the element. Therefore, total number F(n) is:

$$\frac{\log(n)\, n^2}{2} - \frac{\log(n)\, n}{2}$$

So Cg(n) equals to:

$$\frac{\log(n)\, n^2}{2}$$

Solve F(n)-Cg(n)>0 to get:

$$n > 1$$

Therefore, Nc=1

2) For Q2:

Given a set with n component:

For Quick Find, the find operation takes only 1 step. Union operation needs n steps in the worst case.

For Quick Union, the find operation takes n operations in the worst case. Union operation takes, in the worst case, n-1 steps to find the pair and 1 step to union them. In total, union operation could need n steps including the cost of find.

For Weighted Quick Union, the tree is always balanced, so the depth of the tree is at most $\log_2(n)$. This is because the depth of the tree only increases when it is attached to a bigger tree, which means the size of the tree is at least doubled after this union operation. Given n components the size can double at most $\log_2(n)$ times. Therefore, the find operation will need $\log_2(n)$ steps in the worst case. The union operation will need $\log_2(n)$ steps including the cost of find.

This is valid since n>1, therefore Nc=1.

Reference: 1. Algorithms 4th edition, by Robert Sedgewick & Kevin Wayne

2. http://blog.csdn.net/guduruyu/article/details/70313176