Atul Srivastava
Data Structures and Algorithms Assignment 1
Professor Shantenu Jha
RUID: 170002071

***In all questions with data, I manually ran the program looking for different sizes and configurations and recorded it on an excel sheet to generate all of my graphs.**
***github repo : https://github.com/ASriv98/ECE_573_Data_Structures,**

Question 1
Inspiration for this problem's implemented algorithms came from this blog site:
http://home.acastano.com/blog/3-sum-nzp-algorithm/

This question asks the student to compute the 3-sum algorithm in various different implementations with different runtimes. In particular, first, I programmed a naive approach, with an n^3 runtime. Then, I programmed a more efficient version that has an n^2 * log (n) runtime.

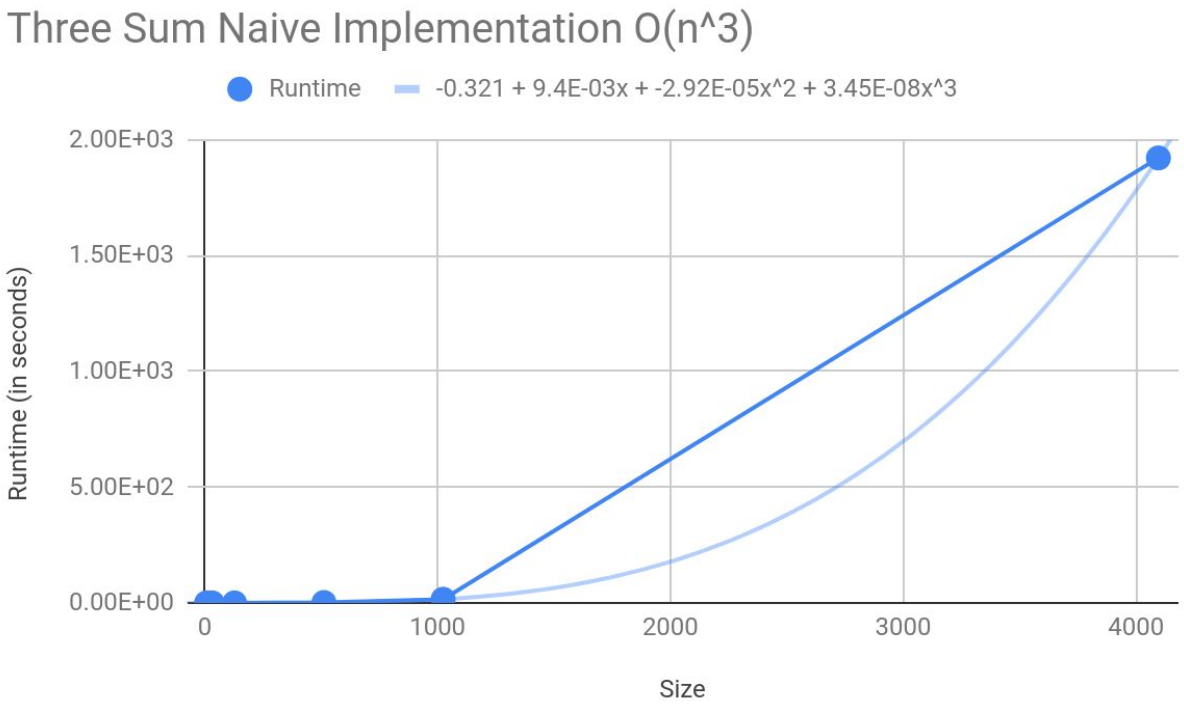
Data:

	O(n^3)	O(N^2 log n)
Size	Runtime	Runtime
8	1.80E-05	3.46E-05
32	4.94E-04	4.81E-04
128	2.78E-02	8.92E-03
512	1.67E+00	1.78E-01
1024	15.71378884	0.7838504314
4096	1921.234092	13.67514119
4192	14.37104961	14.21649585
8192	55.28222916	53.69807696

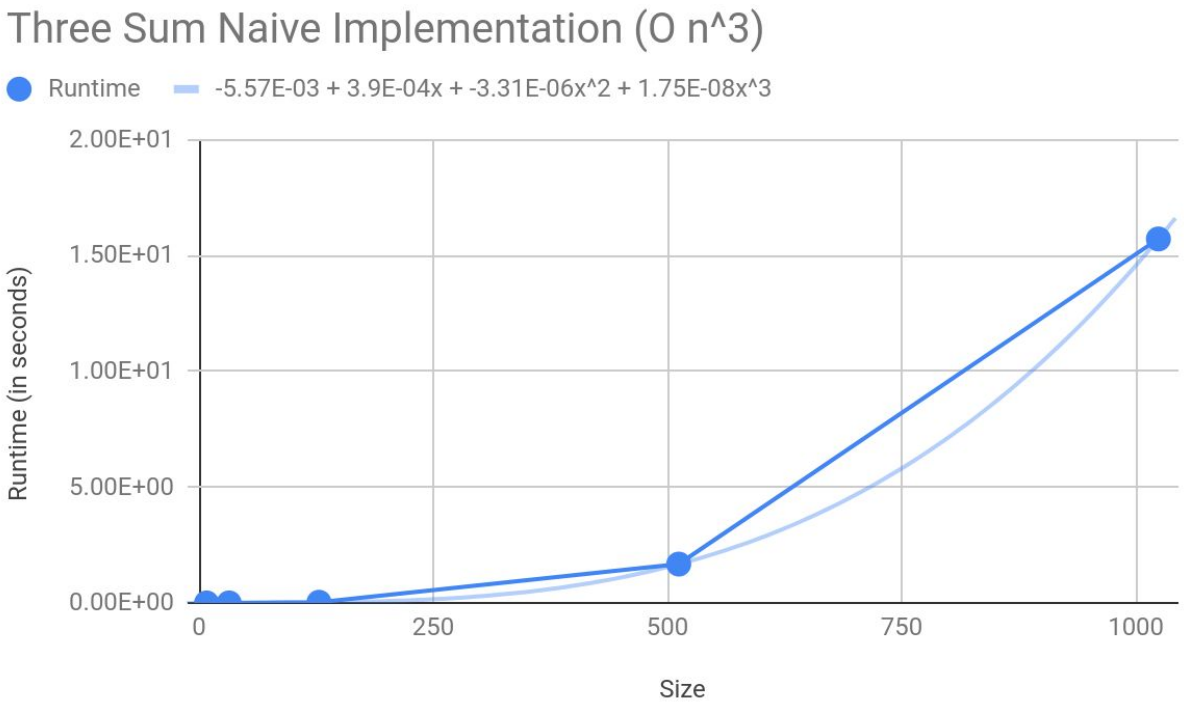
$O(n^3)$ Implementation

This algorithm essentially runs through the dataset three times, searching each individual pair in three for loops. I tested values up until 1024 each 5 times, and after that tested the value of 4096 once (~30 minutes) and didn't run the larger two datasets until completion.

The first graph includes the data point of one trial of 4096, and the graph below does not.



This graph shows the data when only using the datasets of 8, 32, 128, 512, 1024. This graph exhibits a much better exponential curve.

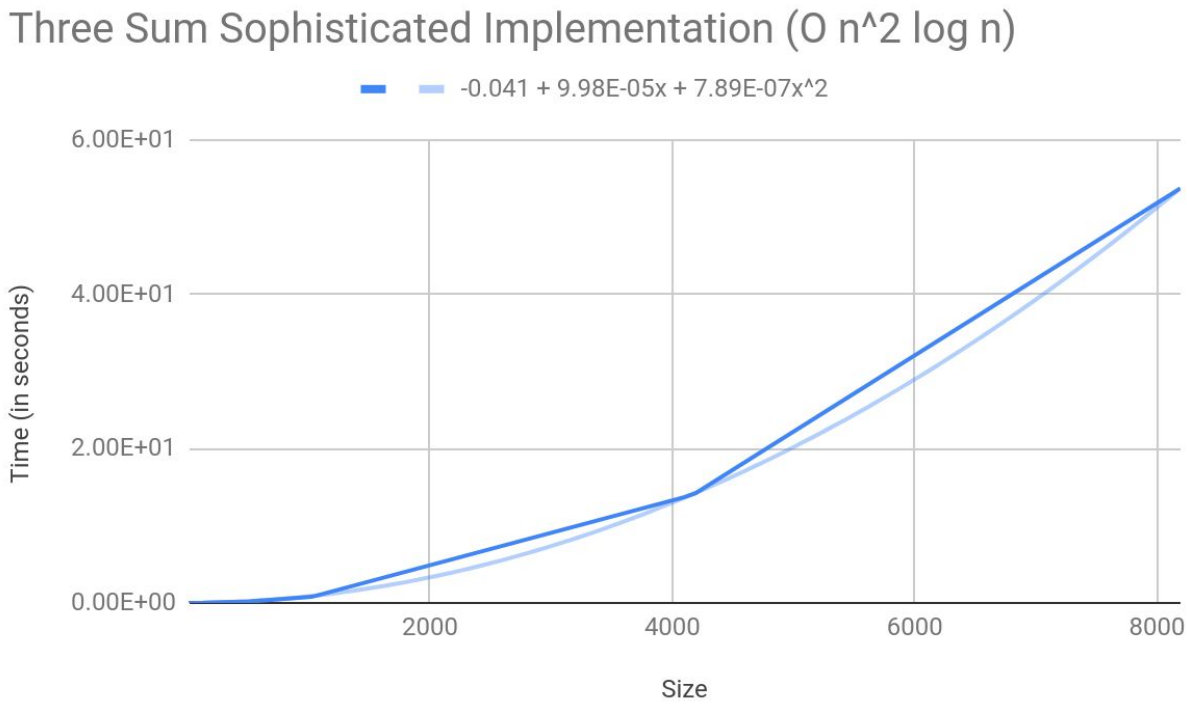


If I use the doubling hypothesis with the data averages at 512 and 1024, I get the following relation:

$log_2(\frac{15.71378}{1.67}) = 3.23$ *which is approximately* ~ 3.

O($n^2 logn$*) Implementation*

This algorithm first sorts all of the given data (in n log n time, using python's native inbuilt sorting algorithm). Then, afterwards, I iterate over the array twice. We do this in order to find the first two elements of the three sum, which takes a total O(n^2) time. Then, we are searching for a third element that we know will hit the target (the negative sum of the first two elements), and we accomplish this by performing a log(n) binary search. In total, we get a total runtime of O($n^2 logn$) *time*.



Using the doubling hypothesis with the same set of data we did for the previous implementation, we get

$log_2(\frac{0.7838504314}{1.78E-01}) = 2.1387$, which I think can be considered close to $O(n^2 log(n))$

Question 2

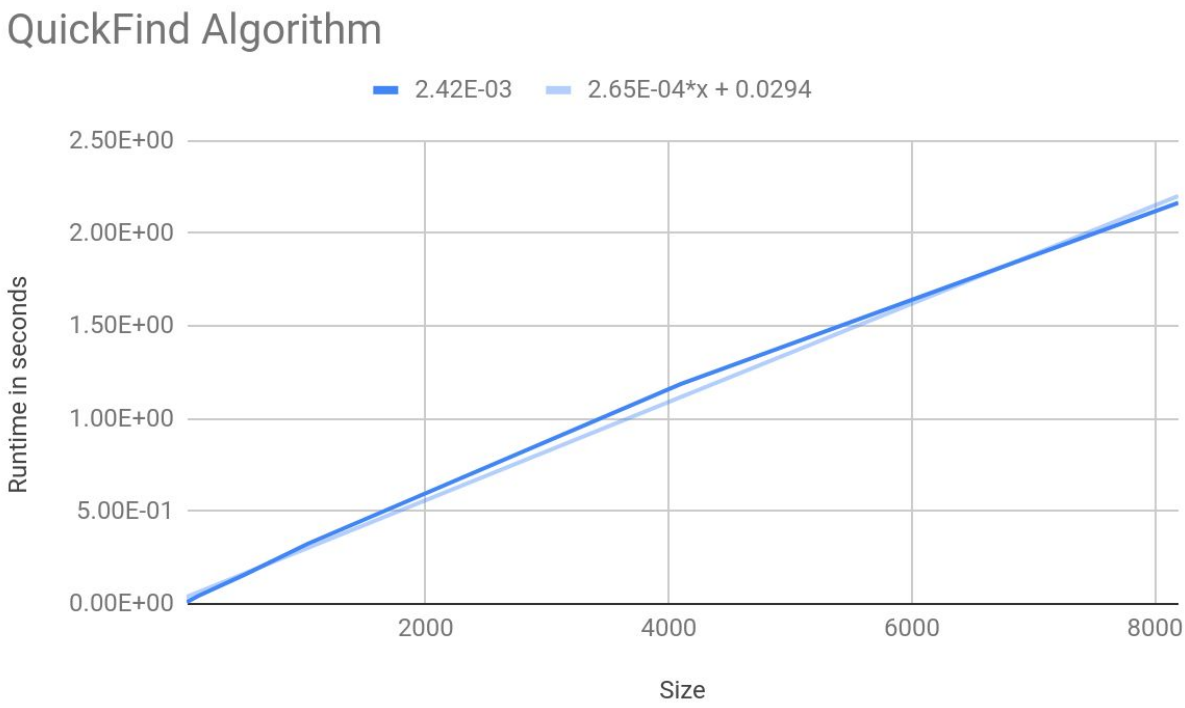
Used this blog site as a resource while creating the implementation for these algorithms: http://www.studies.nawaz.org/posts/union-find-algorithm/?fbclid=IwAR2_6oTau-H9pvfPSTczUMJH_mvC4xdbi-aCxdZxYy2JcoK0IyT4A-cQ9XI

QuickFind Algorithm

The first algorithm creates an array of size n, which in our case is assumed to be 8192 for all input sizes. We first check if two components are connected, and if they are not, we perform the union of the pairs. The total algorithm is of order O(n*m). M represents the size of the input (ranging from 8-8192), whereas n represents the constant 8192 size. At most, we can have 8192 x 8192 connections. This means we have a worst case running time of O(n^2).

Size	Runtime

8	2.42E-03
32	9.74E-03
128	4.25E-02
512	1.58E-01
1024	0.3232214451
4096	1.186569929
8192	2.164452314

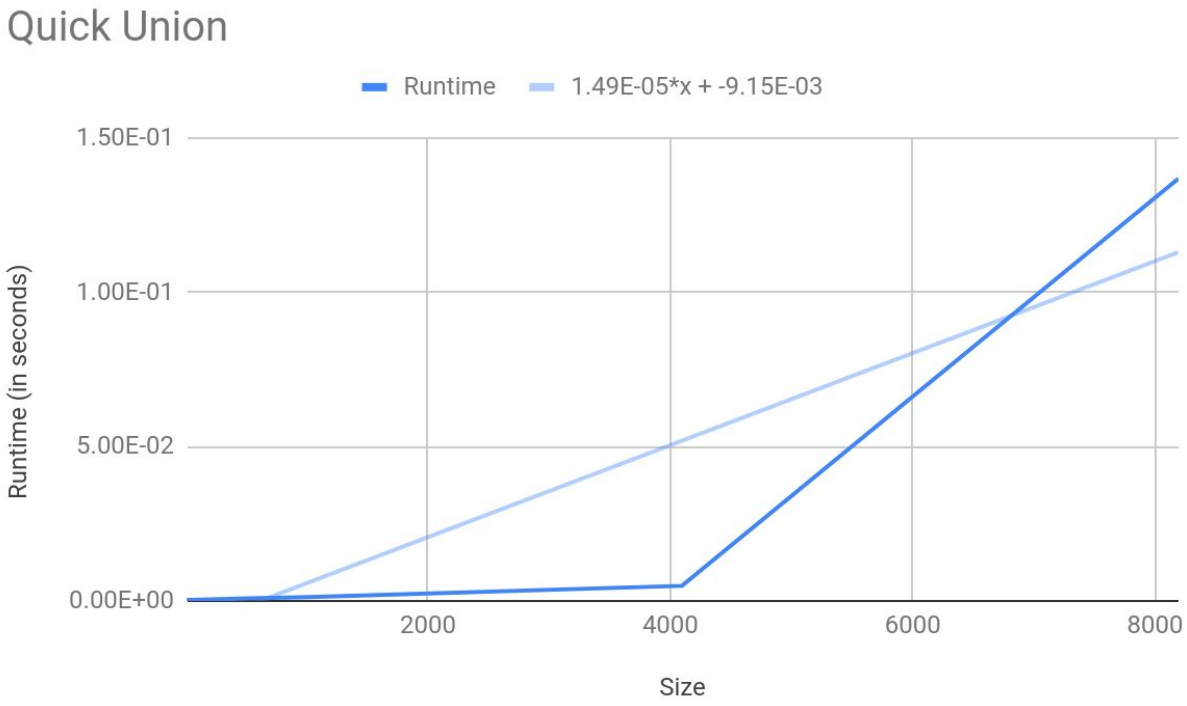


QuickUnion Algorithm

The quick union algorithm is slightly more efficient than quick find. The graph is constructed similar to a tree, and initially, we set the root as themselves.When the union occurs, we set the root of one node as equivalent to the that of the other node. We still have our array of size of n (which in our case is 8192). Now, when we are doing a find, we just check the roots to see if they are equal. When we do a union, we simply make one of the pairs root a child of the others, which will link all connected pairs together. This happens in linear time O(n).

Quick Union	
Size	Runtime
8	2.03E-04

32	2.66E-04
128	3.46E-04
512	7.73E-04
1024	0.001237710317
4096	0.004880030955
8192	0.1367096901

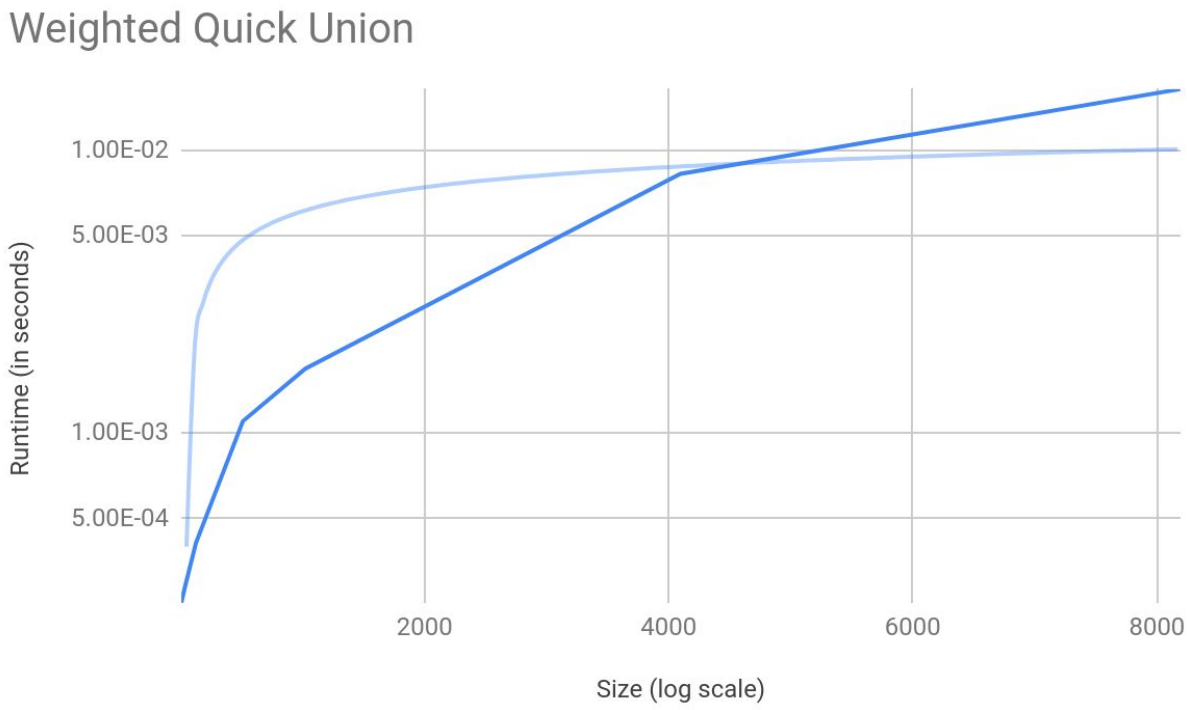


Weighted Quick Union Algorithm

This algorithm takes O(log(n)) time and I tried to fit it on a log scale with a logarithmic trend to the graph. The reason the complexity is log(n) is because unlike a regular quick union algorithm, in the weighted quick union algorithm, we are making sure that we are connecting the root of the smaller tree to the larger one.

Weighted Quick Union	

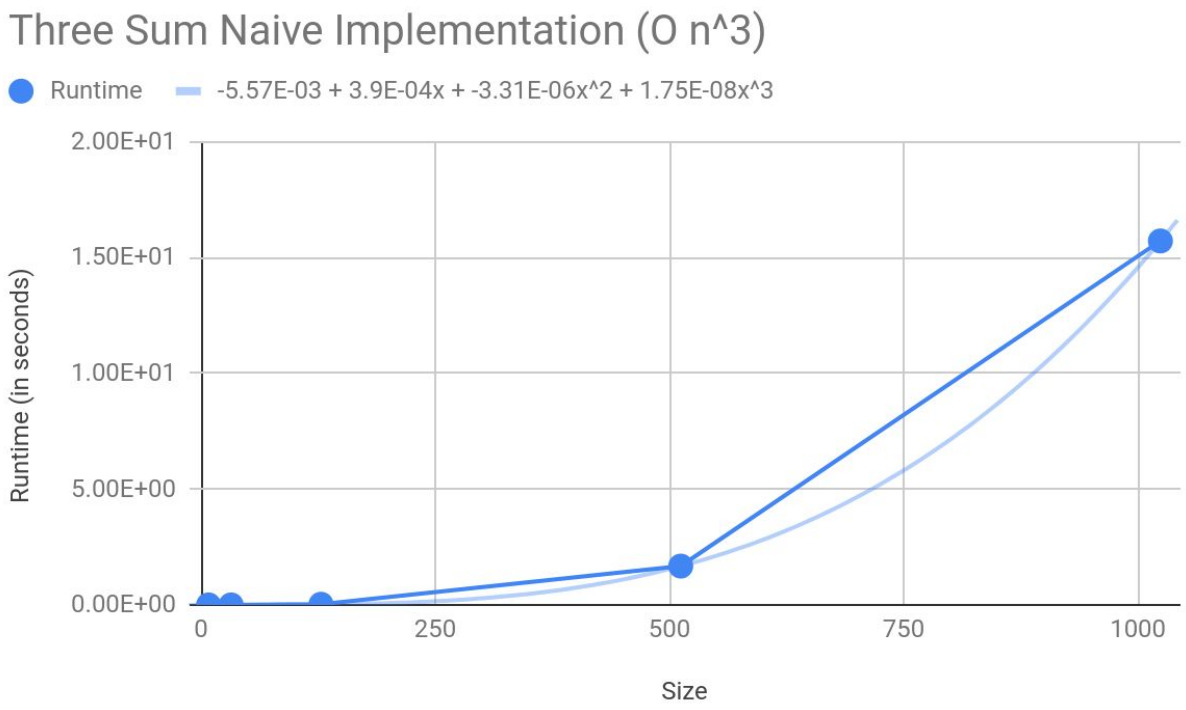
Size	Runtime
8	2.51E-04
32	2.80E-04
128	4.10E-04
512	1.10E-03
1024	0.001687765121
4096	0.008231719335
8192	0.016404708223



Question 3

$O(n^3)$ Three Sum Implentation

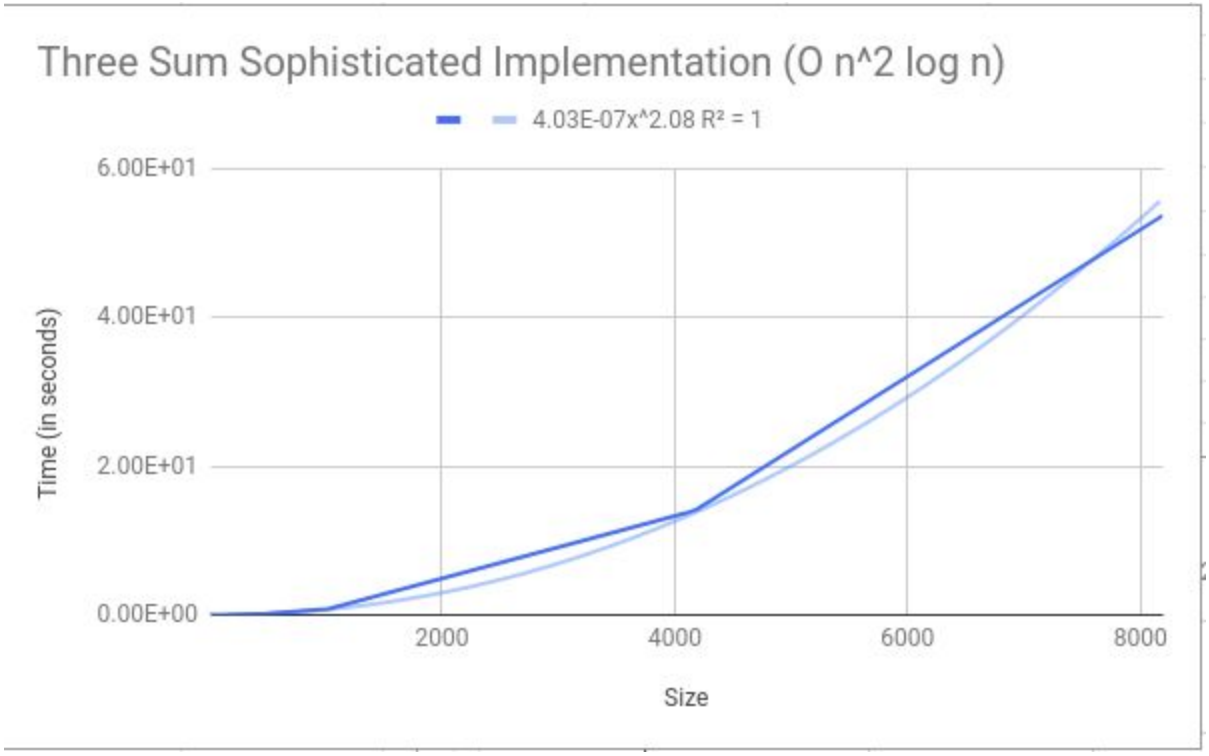
Using curve fitting tool present in excel, we obtain the simulated equation that models the curve fitted line (shown in the graphs above). Using the highest degree exponent, we have



$f(x) = 1.75 * 10^{-8}x^3$, $f(n) = 1.75 * 10^{-8}n^3$, and I am ignoring the lowering degree polynomials because at a large dataset they will not contribute to the running time.

Using the definition of big O, we set $g(n) = n^3$ and c = 1. So we can set Nc = 1 as well. This satisfies the relation of $F(n) < c * g(n)$

$O(n^2 logn)$ Three Sum Implementation



In this problem, our $g(n) = n^2 \log n$

The f(n) given by the excel tools best fit equation for the given data gives our function to be $f(n) = 4.03 * 10^{-7} * x^{2.08}$, which is close to the $n^2 log n$ relationship. If we set c = 1 in the relationship of

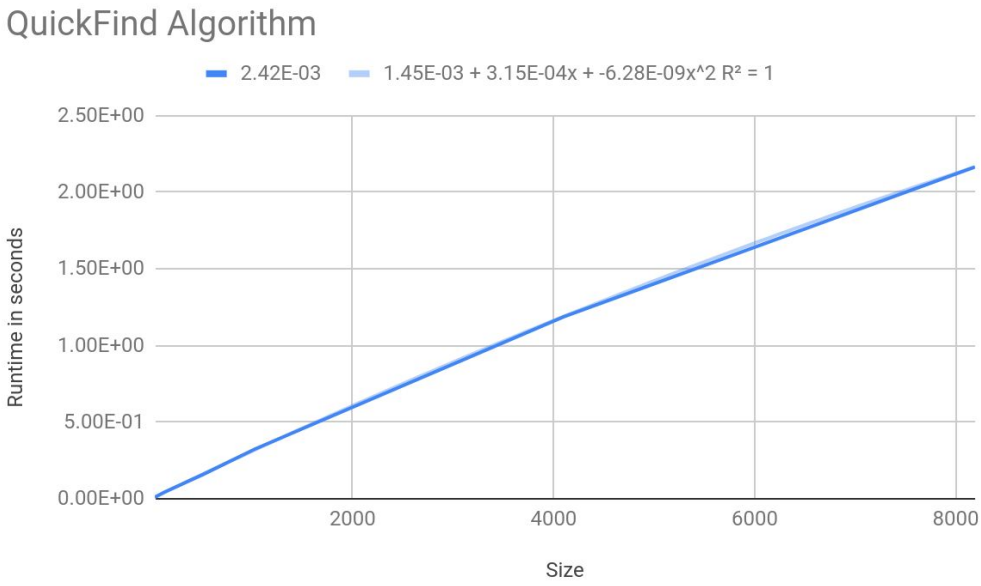
$f(n) < c * g(n)$, we can set Nc = 100.

QuickFind

$g(n) = n^2$

$f(n) = 6.28 * 10^{-9} n^2$

If you set c = 1, you can have NC at a range from 1 to $6.28 * 10^{-9} * NC < 1$ to satisfy the relation of $f(n) < c * g(n)$



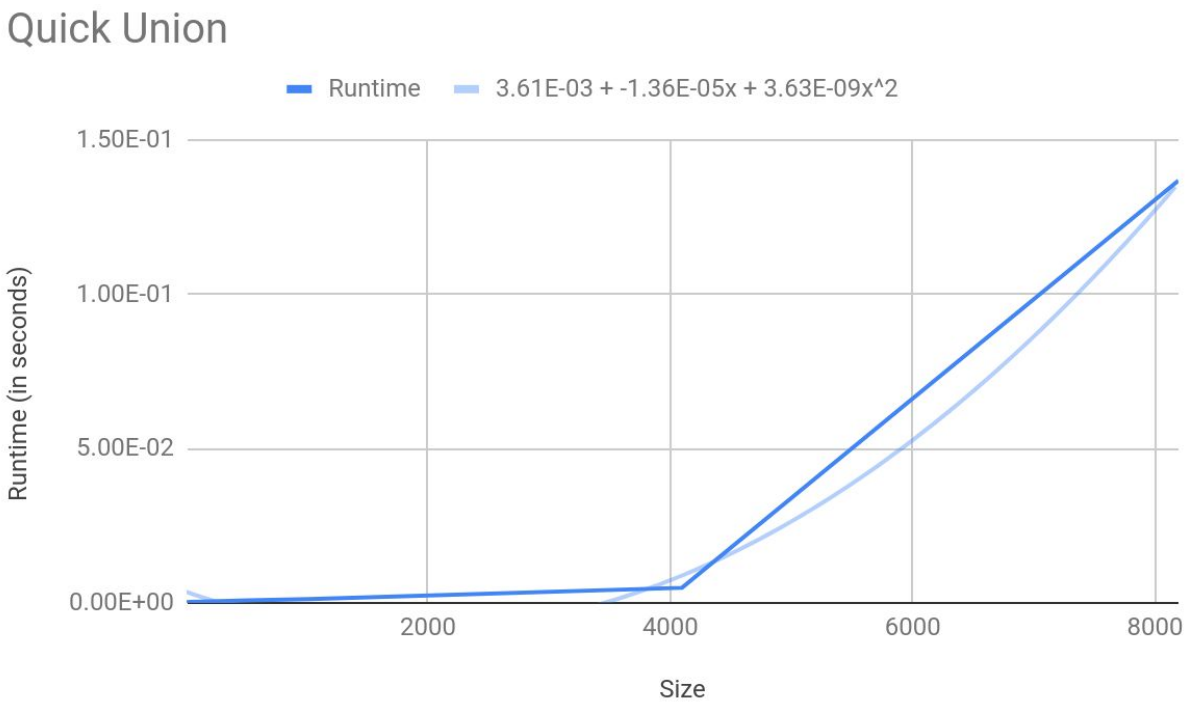
QuickUnion

$g(n) = n^2$

$f(n) = 3.63n^2$

If you set c = 1, you can have NC at a range from 1 to $3.63 * 10^{-9} * NC < 1$ to satisfy the relation of $f(n) < c * g(n)$.

So we can have NC=10 and C=10



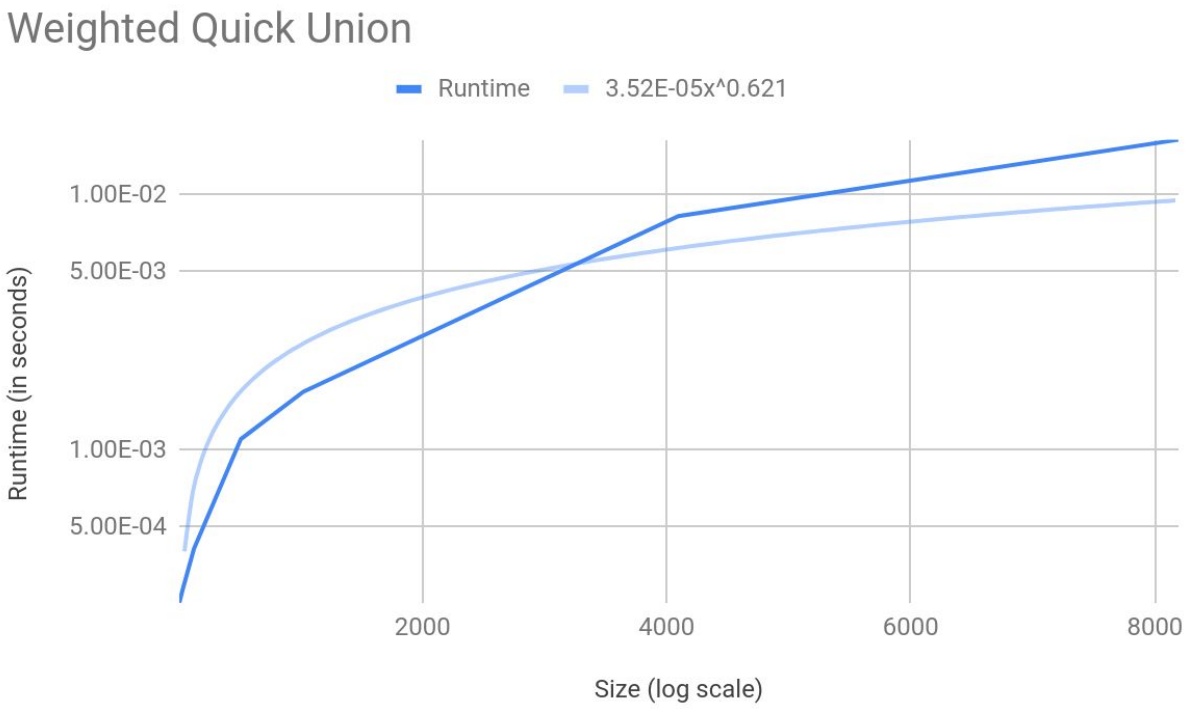
QuickUnion Weighted

$g(n) = n * log(n)$

$f(n) = 3.52 * 10^{-5} n^2$

If you set c = 1, you can have NC at a range from 1 to $3.52 * 10^{-5} * NC < 1$ to satisfy the relation of $f(n) < c * g(n)$.

So we can have NC=10 and C=10

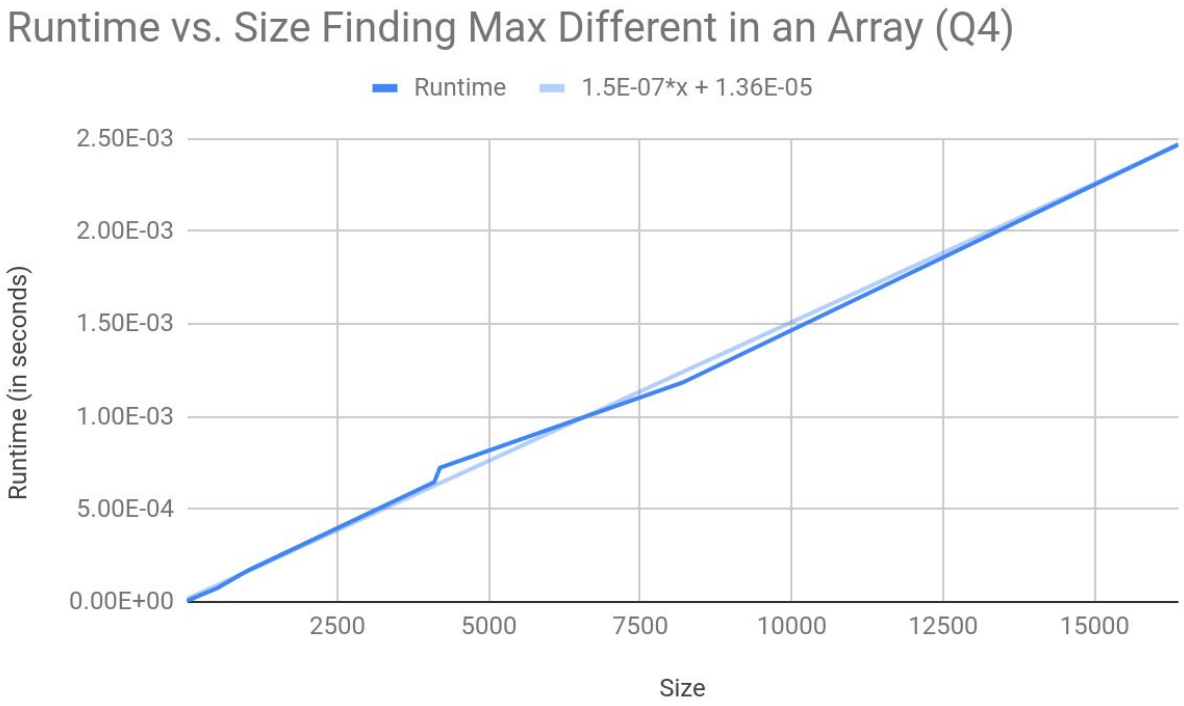


Question 4

This problem asks the student to find the maximum difference between any two elements in an array. So, with one linear loop through the data set, we can check whether any given element is either a maximum value in the array or a minimum value in the array (this will work for absolute value as well). For testing, I ran this multiple times with the given data set 10 times and added a larger data set as well.

Size	Runtime (in s)
8	2.91E-06
32	6.91E-06
128	2.11E-05
512	7.38E-05
1024	0.0001697063446
4096	0.0006456851959
4192	0.0007246017456

8192	0.001182985306
16384	0.002468633652



From the graph of the data and the data itself, it is clear that this algorithm follows a linear run time and is O(n).
Also, when adding a trend line to the data, we see that it is to the order of n, fitted by the lighter blue line above.

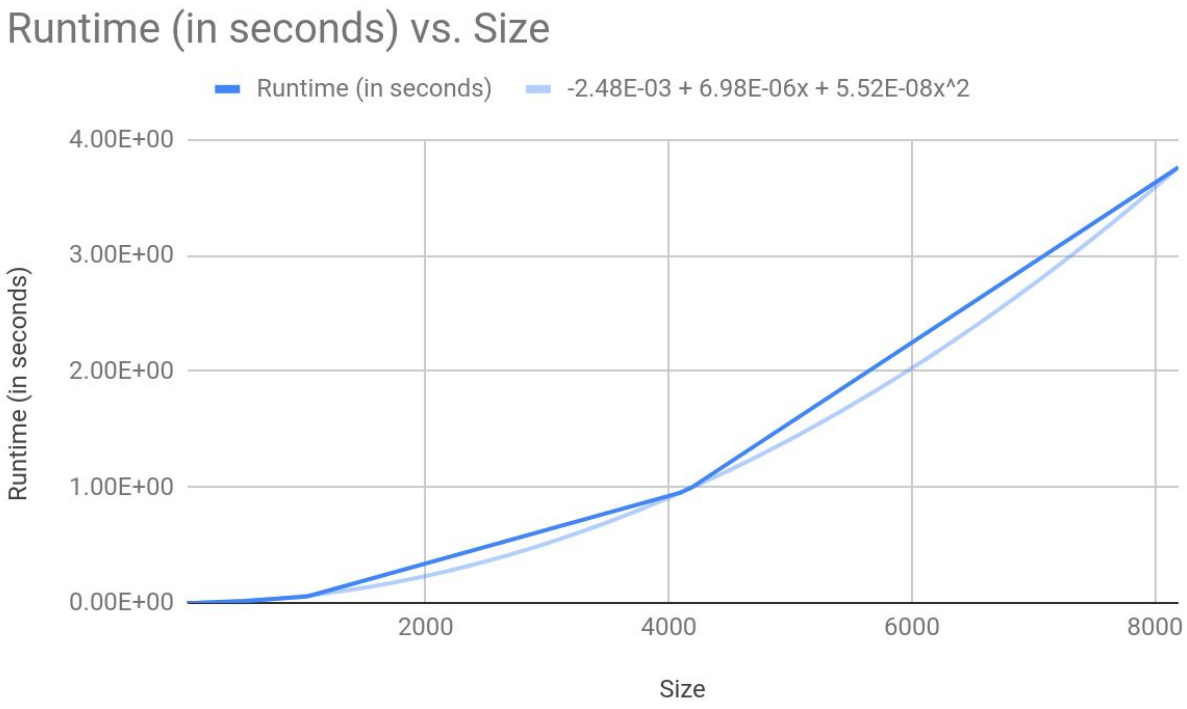
Question 5

Again, this questions implementation algorithm uses information from the following two sites:
http://home.acastano.com/blog/3-sum-nzp-algorithm/
https://www.geeksforgeeks.org/find-a-triplet-that-sum-to-a-given-value

This question asks the student to program a three sum algorithm that is O(n^2) time complexity. In order to do this we still use the two for loop approach that we used in the O($n^2 * log(n)$), and we are searching for a third value that can give us the target sum. However, instead of using a

binary search, we will now use a hash table, which gives us an O(1) look up time. The reason we can use a hash table. We populate the hash table before we begin the the loops in O(n) time, and then we proceed to sort the input data in O(n*log(n)) time. After that, we follow the same algorithm but with hashmap lookup instead.

O(n^2) implementation	
Size	Runtime (in seconds)
	1.87E-05
32	8.64E-05
128	9.53E-04
512	1.63E-02
1024	0.05588359833
4096	0.9530358791
4192	0.9996739864
8192	3.760603762



Using the doubling hypothesis, similar to question 1, we can see that this implementation is in fact $O(n^2)$. I am going to use the data from 1024 and 512.

$$log_2(\frac{0.05588359833}{1.63E{-}02}) = 1.78 \sim 2$$

The doubling hypothesis supports the claim that it is in fact an $O(n^2)$ function. Also, when doing a polynomial regression on the graph, we see that the equation given is a quadratic to the power of x^2 (see above).