Data Structures and Algorithms

Homework # 1

Eren Erdogan

Question 1:

For question 1 we were asked to analyze the two versions of 3-sum problem: One brute force method that takes $O(N^3)$ and an other that uses binary search to make it faster and takes $O(N^2 \lg N)$ were lg is log with base 2.

For each method, integer values were read from a text file and stored in an array before running the testing.

The brute force $O(N^3)$ 3-Sum approach used 3 nested for loops. Assuming each loop takes $O(N)$, multiplying them all together is how we get $O(N^3)$.

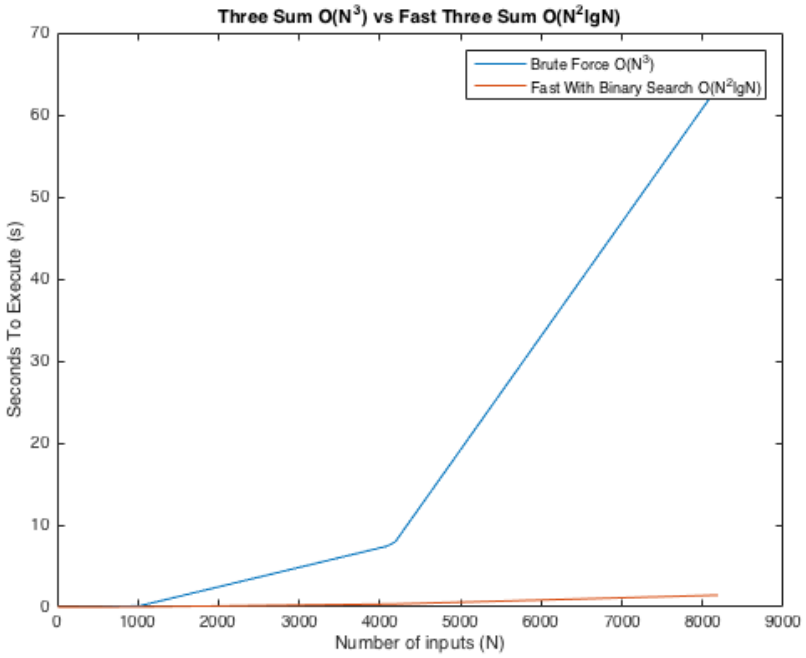
The faster binary search $O(N^2 \lg N)$ method uses 2 for loops but then instead of searching each element one by one for the third comparison, we binary search the array looking to see if ($-array[i] - array[j]$) was in it. If it was then we could conclude that there was a successful 3 sum triple.

The two for loops each took $O(N)$, so multiplying them together we get $O(N^2)$. The binary search though only took $O(\lg N)$, and therefore multiplying everything total together is how we get the total $O(N^2 \lg N)$ run time.

In the table below we can see the results we got for our running of each of these algorithms with different input sizes. Please see the file HW1_#1_Testing_Results.txt to see the testing that was done for this question.

Brute Force Approach vs. Faster Approach With Binary Search Results		
N (number of inputs)	**Brute Force Approach (N^3) time (s)**	**Faster Approach With Binary Search (N^2*lgN) time (s)**
8	0	0
32	0	0.001
128	0.004	0.001
512	0.028	0.007
1024	0.138	0.019
4096	7.432	0.353
4192	7.961	0.366
8192	63.476	1.442

We can then plot these values in MATLAB and see the results on the next page.



We can see that these results validate our theoretical values of $O(N^3)$ for brute force and $O(N^2 \lg N)$ for the fast approach with a binary search. It is obvious to see that the binary search method is significantly faster. While it is not a perfect replica of a theoretical y=N^3 and y = N^2lgN plot, it is as accurate as we can get since we only had 8 different data points to plot. As we increase the amount of data, the plot should become more and more similar to the theoretical plot.

Question 2:

For this question we had to implement four versions of the different quick-union algorithms that we discussed in class. The four different versions were (i) Quick-Find, (ii) Quick Union, (iii) Weighted Quick Union, (iv) Weighted Quick Union with Path Compression.

This implementation was similar to the implementation of questions 1. Each different algorithm was run using the same data set and then the time it took to complete each was outputted onto the screen. Please see the file "HW1_#2_Testing_Results.txt" to see the testing that was done.

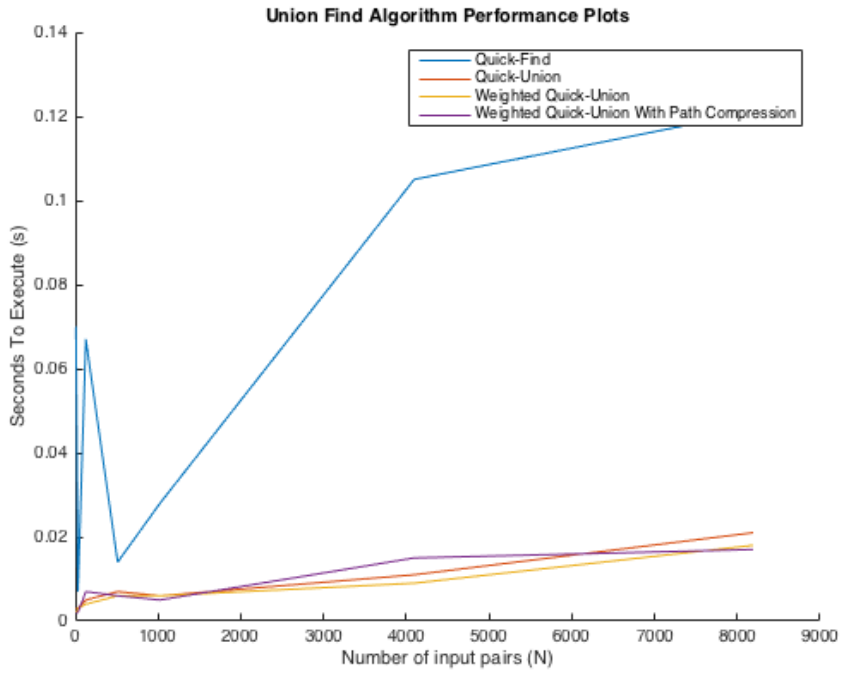
The result from this testing can be seen in the table on the next page.

Run Time of Different Union-Find Algorithms				
Number of Pairs Inputted	Quick Find	Quick Union	Weighted Quick Union	Weighted Quick Union w/ Path Compression
8	0.07	0.003	0.002	0.002
32	0.07	0.003	0.003	0.002
128	0.067	0.005	0.004	0.003
512	0.014	0.007	0.006	0.006
1024	0.028	0.006	0.006	0.005
4096	0.105	0.011	0.009	0.015
8192	0.121	0.021	0.18	0.017

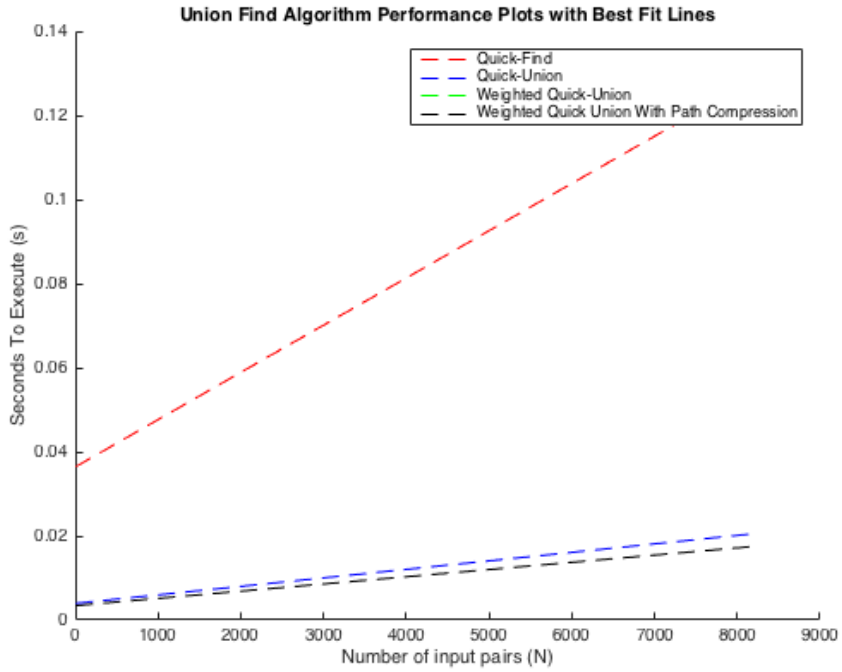
From the table we can see that our testing results support the theoretical speed ranks that were discussed in the book. As we get to larger and larger sets of data it is easy to see that Quick-Find is the slowest followed by Quick Union, Weighted Quick Union, and Weighted Quick Union with Path Compression in that order.

To get a better idea of these results we can plot the values again using MATLAB.

Please see below for this plot,



Our data points are a little all over the place for the smaller values for various reasons so to get a clearer picture of our results, it'd be better to plot a best fit line for the data from our experiment. This can be seen below,



Now it is much clearer to view and compare the results of our different algorithms. The weighted quick-union and weighted quick-union with path compressions lines are so similar they are essentially on top of each other. However from looking at our data, specifically for the largest value of 8192 pairs, we know that the weighted quick-union with path compression is faster as the pairs of inputs gets larger.

Lastly for the review of this question we'll determine the run time cost as a function of the input size. We'll start with Quick-Find.

Quick-Find

For Quick-Find, the find operation takes only $O(1)$ time since it only has 1 array access that is has to do. However for the union operations, the inner loop will run N times since it has to run for the entire length of the input size. Therefore, the run time and cost of the union operation is $O(N)$, which is far too slow!

Quick-Union

Next, we'll look at Quick-Union which is an improved approach to the Quick-Find algorithm. For Quick-Union the find operation will run until it reaches the root node of it's path. This will take $O(tree\ height)$, since it is directly dependent on the tree

height in the given path it is analyzing. Worst case, all the nodes will be in the same path but this is unlikely. The union operation calls the find operation twice and then changes a link in the tree. Therefore since each find operation has an order of growth $O(tree\ height)$, the union operation will also have and order of growth $O(tree\ height)$.

Weighted Quick-Union

The Weighted Quick-Union takes the benefits of the quick-union and improves it with one simple step. It does so by linking the root of smaller tree to root of larger tree. Therefore by doing so the depth of any node is at most lg (N) which will lower the worst case run time of both the union and find methods each to now $O(\lg N)$.

Weighted Quick-Union With Path Compression

Lastly we have the Weighted Quick-Union With Path Compression. This works just like the Weighted Quick-Union but just after computing the root of p, set the id[] of each examined node will point to that root. By doing so it is spreading the tree to be wide as possible and does so to prevent any long paths that would slow down an operation. Doing so reduces the running time of our union and find operations to just $O(\lg^* N)$. $\lg^* N$ is the iterated logarithm function. It refers to the number of times that the logarithm function must be iterated until the resulting number is less than 1. This function grows very slowly and can pretty much be considered to be constant, however it is not exactly constant.

Therefore now that we've finished the run time costs, we can summarize our results in the table below for the worst-case run times,

Run Time Costs of Union-Find Algorithms		
Algorithm	**Union**	**Find**
Quick-Find	$O(N)$	$O(1)$
Quick-Union	$O(tree\ height)$	$O(tree\ height)$
Weighted Quick-Union	$O(lgN)$	$O(lgN)$
Weighted Quick-Union with Path Compression	$O(lg^* N)$	$O(lg^* N)$

Therefore in conclusion we can conclude that for M union-find operations on a set of N objects the worst-case order of growth for the entire program would be the values in the table on the next page.

Order of growth for M union-find operations on a set of N objects	
Algorithm	**Worst Case Run Time**
Quick-Find	$O(MN)$
Quick-Union	$O(MN)$
Weighted Quick-Union	$O(N + MlgN)$
Weighted Quick-Union with Path Compression	$O(N + Mlg^*N)$

Question 3:

In this question we are asked to estimate the value of N_c for both Q1 and Q2 and explain support our result with our empirical data.

So we are looking to confirm the formal definition: F(N) = O(g(N)) for positive constants c and N_c, such that F(N) < cg(N) for all N > N_c.

We'll start with the first question.

Three Sum

For the three sum brute force algorithm the inner loop would be,

$$F(N) \cong \left(\frac{N^3}{6} - \frac{N^2}{2} + \frac{N}{3}\right)$$

and,

$$g(N) = N^3$$

We can pick c to be any constant greater than 0 so lets pick $1/6$ for simplicity. Then now we need to estimate an N_c such F(N) < cg(N) for all N > N_c.

From looking at,

$$\left(\frac{N^3}{6} - \frac{N^2}{2} + \frac{N}{3}\right) < \frac{1}{6}N^3$$

We see that the two side are equal for N = $2/3$. Therefore if we pick a number N = 1 and plug it in we get,

$$0 < 1/6$$

So we can say that for $N > \frac{2}{3}$; $F(N) < \frac{1}{6} g(N)$ and conclude that our N_c estimate is $\frac{2}{3}$.

From looking at our empirical data we see that the plotted lines best represents line like $g(N) = N^3$ for the brute force algorithm. Therefore we can confirm that our data indeed resulted in an $O(N^3)$ run time.

Three Sum Fast

Next we'll look at the fast sum method. For the sake of being brief, I will not go over the methodology again since it was just explained in detail. I'll just give the $F(N) < cg(N)$ and then give the estimate for N_c. For the Fast Three Sum, I'll treat F(N) as approximately

$$F(N) \cong \frac{N(N-1)\lg N}{2!}$$

g(N) is

$$g(N) = N^2 \lg(N)$$

so taking c = 1, we get

$$\frac{N(N-1)\lg N}{2!} < N^2 \lg(N)$$

These are equal when N = 1. So lets try N=2. We get

$$2 < 4$$

So we can say that for $N > 1$; $F(N) < 1 * g(N)$ and conclude that our N_c estimate is 1.

From looking at our empirical data we see that the plotted lines best represents a line like $g(N) = N^2 \lg(nN)$ for the binary search fast algorithm. Therefore we can confirm that our data indeed resulted in an $O(N^2 \lg(N^2))$ run time.

Now we'll analyze the algorithm for question 2.
We'll go in order of slowest to fastest so we will start with the Quick-Find algorithm.

Quick-Find

From looking at the code for the quick-find union method algorithm we can make an estimate on the F(N) function. This estimate is,

$$F(N) \cong 3N + 3$$

g(N) is

$$g(N) = N$$

so taking c = 4, we get

$$3N + 3 < 4N$$

These are equal when N = 3. So lets try N=4. We get

$$15 < 16$$

So we can say that for $N > 3$; $F(N) < 4 * g(N)$ and conclude that our N_c estimate is 3.

From looking at our empirical data we see that the best-fit plotted lines represents a line like $g(N) = N$ for the quick-find algorithm. Therefore we can confirm that our data indeed resulted in an $O(N)$ run time.

Quick-Union

From looking at the code for the quick-union union method algorithm we can make an estimate on the F(N) function. This worst-case estimate is,

$$F(N) \cong 2N + 1$$

g(N) is

$$g(N) = N$$

so taking c = 3, we get

$$2N + 1 < 3N$$

These are equal when N = 1. So lets try N=2. We get

$$5 < 6$$

So we can say that for $N > 1$; $F(N) < 3 * g(N)$ and conclude that our N_c estimate is 2.

From looking at our empirical data we see that the best-fit plotted lines represents a line like $g(N) = N$ for the quick-union algorithm. Therefore we can confirm that our data indeed resulted in an $O(N)$ run time.

Weighted Quick-Find

From looking at the code for the weighted quick-union union method algorithm we can make an estimate on the F(N) function. This estimate is,

$$F(N) \cong 2\lg(N) + 3$$

g(N) is

$$g(N) = \lg(N)$$

so taking c = 5, we get

$$2\lg(N) + 3 < 5\lg(N)$$

These are equal when N = 2. So lets try N=4. We get

$$7 < 10$$

So we can say that for $N > 2$; $F(N) < 5 * g(N)$ and conclude that our N_c estimate is 2.

From looking at our empirical data we see that the best-fit plotted lines represents a line like $g(N) = \lg(N)$ for the weighted quick-union algorithm. Therefore we can confirm that our data indeed resulted in an $O(\lg(N))$ run time.

Weighted Quick-Find With Path Compression

From looking at the code for the weighted quick-union union with path compression method algorithm we can make an estimate on the F(N) function. This estimate is,

$$F(N) \cong 2lg^*(N) + 2$$

g(N) is

$$g(N) = lg^*(N)$$

so taking c = 4, we get

$$2\,lg^*(N) + 2 < 4\,lg^*(N)$$

These are equal at about N ≈ 1.10. So lets try N = 8. We get

$$6 < 8$$

So we can say that for $N > 1.10$; $F(N) \leq 4 * g(N)$ and conclude that our N_c estimate is 1.10 .

From looking at our empirical data we see that the best-fit plotted lines represents a line like $g(N) = lg^*(N)$ for the weighted quick-union with path compression algorithm. Therefore we can confirm that our data indeed resulted in an $O(lg^*(N))$ run time.

Question 4:

We'll begin by looking at the first part of this question:

Is $2^{N+1} = O(2^N)$?

The answer to this is yes.

When looking for the Big Oh notation we discard any lower order terms and any leading coefficients. 2^{N+1} can be re-written as $2 * 2^N$ therefore when we are doing the Big Oh notation we throw away the coefficient 2 and only look at the function 2^N.

Now for the next part of this question:

Is $2^{2N} = O(2^N)$?

The answer to this question is no.

The 2 in the exponent of the equation is not a coefficient, it is directly part of the function. It will have a drastic effect on the order of growth. We can confirm our conclusion in the table below.

$\boldsymbol{O(2^{2N})}$ ***vs*** $\boldsymbol{O(2^N)}$		
Input of size N	$\boldsymbol{O(2^{2N})}$	$\boldsymbol{O(2^N)}$
10	1,048,576	1024
1,000	1.15E 602	1.07E 301
1,000,000	9.8E 602,059	9.9E 301,029

We can see that the results in $O(2^{2N})$ significantly bigger. This data confirms our answer to this question that no, they are not equal.

Question 5:

For this last question we are asked to compare the order of growth for 30 different functions and organize them into a sorted order.

My approach for this question was straightforward and simple. For each function I tested its value with 3 different input N sizes that were, ten, one thousand, and one million.

Then based of these values, most importantly the biggest one of 1 million, I would be able to organize these in a sorted order from smallest to largest.

The results and sorted order can be seen in the table on the next page.

NOTE: lg*(lgN) is above ln(ln(N)) because even though for 1 million, ln(ln(N)) is bigger, as the data gets bigger and bigger, lg*(lgN) is getting smaller than ln(ln(N)).

For example for N = 1E 18, lg*(lgN) is 3 and ln(lnN) is 3.72.

Order of Growth Sorted in Ascending Order				
Equation	Rank	N = 1,000,000	N = 1,000	N = 10
1	1	1	1	1
lg(lg*(N))	2	1.585	1.585	1
N^(1/(lgN))	3	2	2	2
lg*(lgN)	4	3	2	2
ln(ln(N))	5	2.63	1.933	0.834
lg*(N)	6	3	3	2
(lgN)^(1/2)	7	4.464	3.157	1.823
2^(lg*N)	8	8	8	4
ln(N)	9	13.82	6.907	2.303
2^((2lgN)^1/2)	10	79.55	22.077	5.97
lg^2(N)	11	397.267	99.317	11.035
((2)^1/2)^(lgN)	12	1,000	31.622	3.162
(3/2)N	13	1,500,00	1,500	15
2^(lg(N))	Tie-14	1,000,000	1000	10
N	Tie-14	1,000,000	1000	10
lg(N!)	16	1.85 E 7	8529.398	22
NlgN	17	1.993E 7	9965.78	33.22
N^2	Tie-18	1E 12	1,000,000	100
4^(lgN)	Tie-18	1E 12	1,000,000	100
N^3	20	1E 18	1E 9	1,000
(lgN)!	21	1.98E 18	3.35E 6	9.11
N^(lg(lg(N))	Tie-22	7.98E 25	8.93E 9	53.954
(lgN)^(lgN)	Tie-22	7.98E 25	8.93E 9	53.95
2^N	24	9.9E 301,029	1.071E 301	1024
N*(2^N)	25	9.9E 301,035	1.07E 304	10240
e^(N)	26	3.03E 434,294	1.98E 434	22026.47
N!	27	8E 5,565,708	4E 2,567	3,628,800
(N+1)!	28	8E 5,565,714	4.028E 2570	39,916,800
2^(2^(N))	29	10^(10^(10^5.4)))	10 ^ (10^300)	1.8 E 308
2^(2^(N+1))	30	10^(10^(10^(5.48)))	10^(10^(300.81))	3.23E 616