Homework-1

Data Structures and Algorithms

Name: Vikhyat Dhamija

RU ID: 194003013

Net Id : vd283

Q1. We discussed two versions of the 3-sum problem: A "naive" implementation (O(N^3)) and a "sophisticated" implementation (O(N^2 lg N)). Implement these algorithms. Your implementation should be able to read data in from regular data/text file with each entry on a separate line. Using Data provided under resources (hw1-data.zip) determine the run time cost of your implementations as function of input data size. Plot and analyze (discuss) your data.

Solution:

A. Program code has been attached in the folder in two parts:

1. Naïve Implementation- (O(N^3))

2. Sophisticated Implementation - (O(N^2 lg N))

{ Note the respective programs display the counter for determining the complexity as suggested by the professor in the class}

Total Time cost of any algorithm can be calculated by considering the time taken by each statement * how many times that statement is executed

But in order to simplify , as explained by the professor and the course book and notes , if we need to measure the performance of an algorithm , we use Simplification i.e. Simplification 1

i.e. Cost model where we Use some basic operation as a proxy for running time.

As suggested , I am using the number of times loops are executed for array accesses and the comparisons.

Hence I have used counter in the program as a proxy for the cost i.e. the time. And then I am plotting the cost vs input size graph for the both implementation of the two implementation of the 3 sum Algorithm.

A. Naïve Implementation

S.No.	Input Size(n)	Log(n)	Counter Value (proxy for time/cost)(T)	Log(T)	Slope of Log T plot(that determine the order of the cost function)
1.	8	3	56	5.807354922	3.234385
2.	32	5	4960	12.27612441	3.052439
3.	128	7	341376	18.38100211	3.012784
4.	512	9	22238720	24.40657042	3.004237

5.	1024	10	178433024	27.41080741	3.001587
6.	4096	12	11444858880	33.41398062	3.000762
7.	4192	12.033423	12268800000	33.5142751	3.00052
8.	8192	13	91592417280	36.41450911	2.801116

B. Sophisticated Implementation

S.No.	Input Size (n)	Log(n)	Counter Value (proxy for time/cost)(T)	Log(T)	Slope of Log T plot(that determine the order of the cost function)
1.	8	3	31	4.95419631	2.698924
2.	32	5	1307	10.35204343	2.393334
3.	128	7	36075	15.13871177	2.26457
4.	512	9	832939	19.66785132	2.208922
5.	1024	10	3850923	21.87677285	2.173073
6.	4096	12	7.83223e+07	26.2229198	2.143886
7.	4192	12.033423	82310600	26.2945749	2.146664
8.	8192	13	3.46791e+08	28.36949122	2.182269

Important Note for running the program implementation of the algorithms:

a. **That for running the algorithms/programs it is assumed that the files are stored in the same folder as the program**

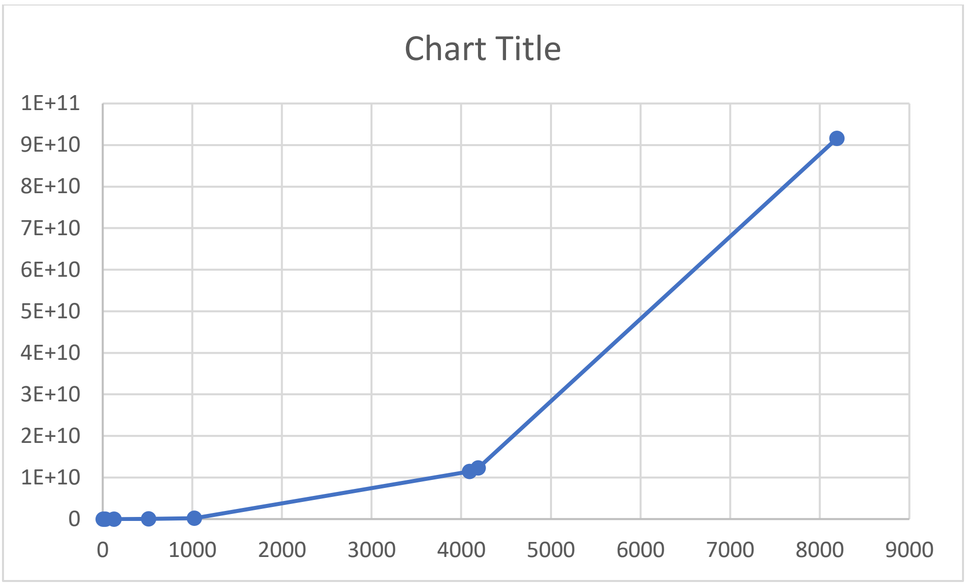
b. **And there is a need to just change the 8/32 like numbers in the file name and the N macro defined above as per the size of the input dataset , in order to see the results of the execution of program with various input sizes .**

c. **Program will display the counter value that determines the cost of the algorithm.**

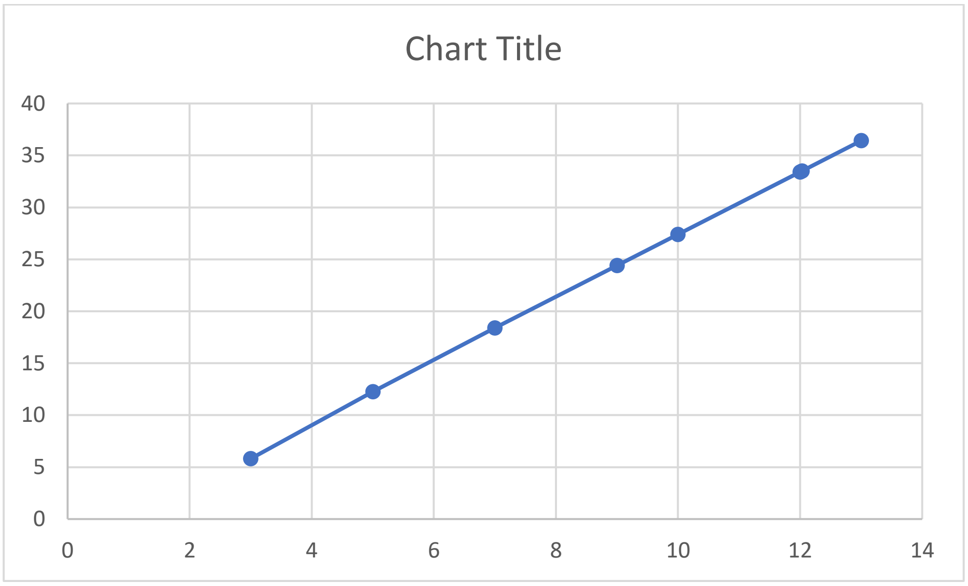
Plots for this **cost vs input size** for both the above algorithms

A. Naïve Implementation

T Vs. N Plot

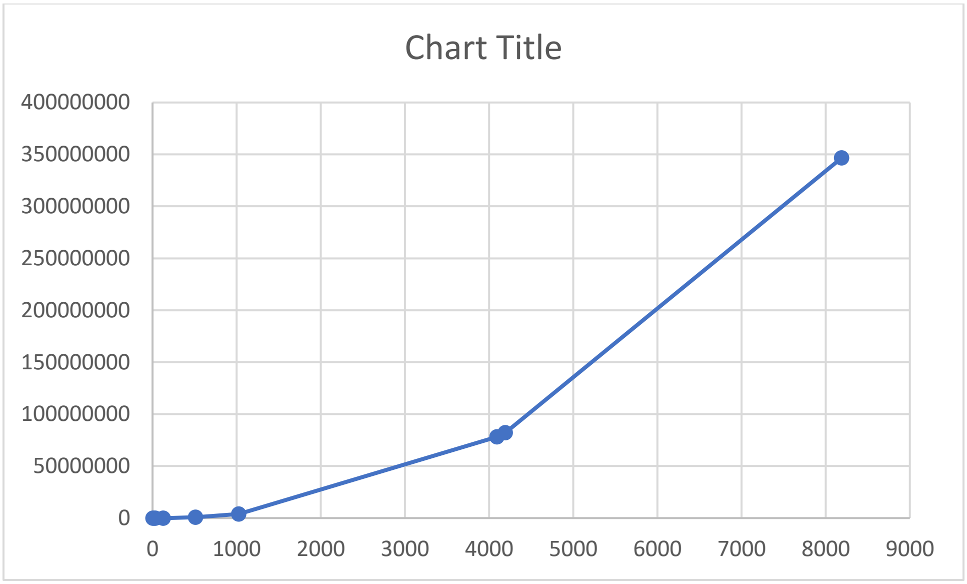


Log-Log Plot Log T Vs. Log N

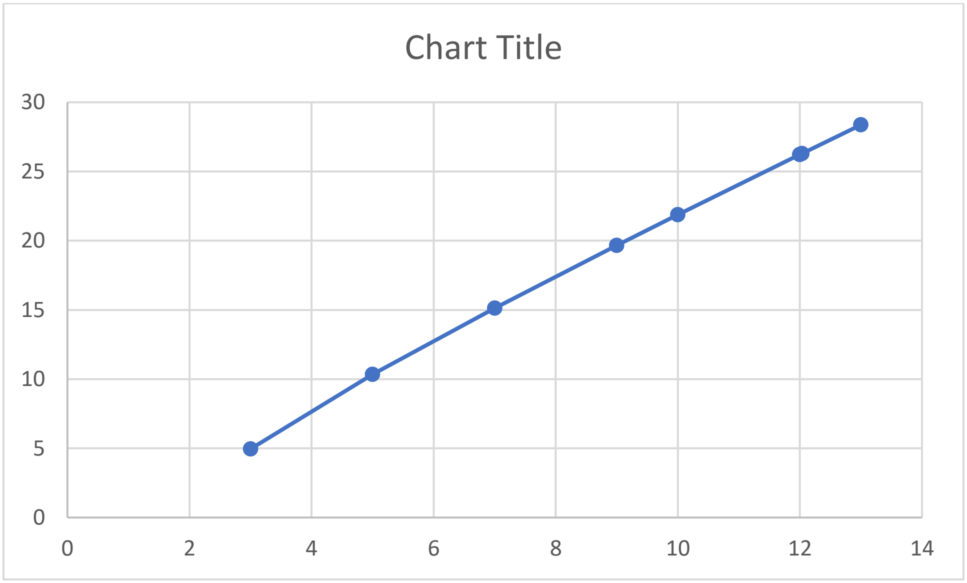


B. Sophisticated Implementation

T Vs. N Plot



Log-Log Plot Log T Vs. Log N



From the Plots Conclusions:

a. Log-Log Plot for the order of Growth of the Naïve Implementation show the Linear Curve with slope of 3. Clearly demonstrating ~ n^3 order of growth as expected. .(as Log-Log Plot becomes 3Log n)

b. Log-Log Plot for the order of Growth of the Sophisticated Implementation show the almost Linear Curve with slope nearby of 2 . Clearly demonstrating ~ n^2 Log n(where Log N is definitely due to binary search where the search divides the array into half every time) order of growth as expected.(as Log-Log Plot becomes 2 Log n +Log(Log n)).

c. It is important to note that for the sophisticated implementation of the algorithm first the sorting of the array is being done , in my case the insertion sort algorithm is used which as discussed in the class have O(N^2) complexity but in case of partial sorted ones it has even less array accessed which is used to measure the complexity of the sorting algorithm.

   As it was clearly observed that the algorithm has two parts

a. Sorting

b. Two Loops + Binary Search

As the total complexity of sorting has been known so B which obviously will be having more cost because of two loops and binary search been running simultaneously.

So when algorithm has parts A + B then following the fact that O(A)+O(B) is max(O(A),O(B)){Note as smaller one just increase the proportionality of the maximum cost algorithm} . So as explained previously B was considered for the complexity of the algorithm.

Note in the third question further it is clearly explained why the O notation function for a and b are O(N^3) and O(N^2 Log N) respectively.(N and n both interchangeably have been used to denote the number of elements from which three pair sum problem has to be solved)

Q2. We discussed the Union-Find algorithm in class. Implement the three versions: (i) Quick Find, (ii) Quick Union, and (iii) Quick Union with Weight Balancing. Using Data provided here determine the run time cost of your implementation (as a function of input data size). Plot and analyze your data.

Note: The maximum value of a point label is 8192 for all the different input data set. This implies there could in principle be approximately 8192 x 8192 connections. Each line of the input data set contains an integer pair (p, q) which implies that p is connected to q.

Recall: UF algorithm should

a. **read in a sequence of pairs of integers (each in the range 1 to N) where N=8192**
b. **calling find() for each pair: If the members of the pair are not already connected**
c. **call union() and print the pair.**

Ans:

As same as above in the previous question , here also we will use the certain operations that are repetitive in nature as the proxy for the time cost of these algorithms as is done in for understanding the overall complexity and comparisons of the algorithm as suggested by the professor , notes and the course book.

Let us move to algorithms :

1. **Quick Find Algorithm**

Theoretical analysis of this algorithm clearly suggest that for <u>**every find**</u> only one operation is needed i.e.

For N Finds of pairs that whether they are connected or not there will be N **operations so theoretical complexity is N only for the input Data Set.**

And for Unions of N Pairs(e1,e2) as for every union operation , the algorithm has to change the labels of all elements of group of e1 with the label of e2 which is the Array[e2].Hence in every case of union we have to search all elements of the group of e1 (i.e. the set of elements to which the e1 is connected) in order to you can say to merge this group of elements with the group of e2 (i.e. the set of elements to which the e1 is connected) because now the elements of both of the sets will all be connected because of the union the bridge between the two set has been created now.

Because of the above , **in the worst case N * M complexity will be there where N is the number of pairs in the dataset and M is the size of Data Set which in our case is of 8192.**

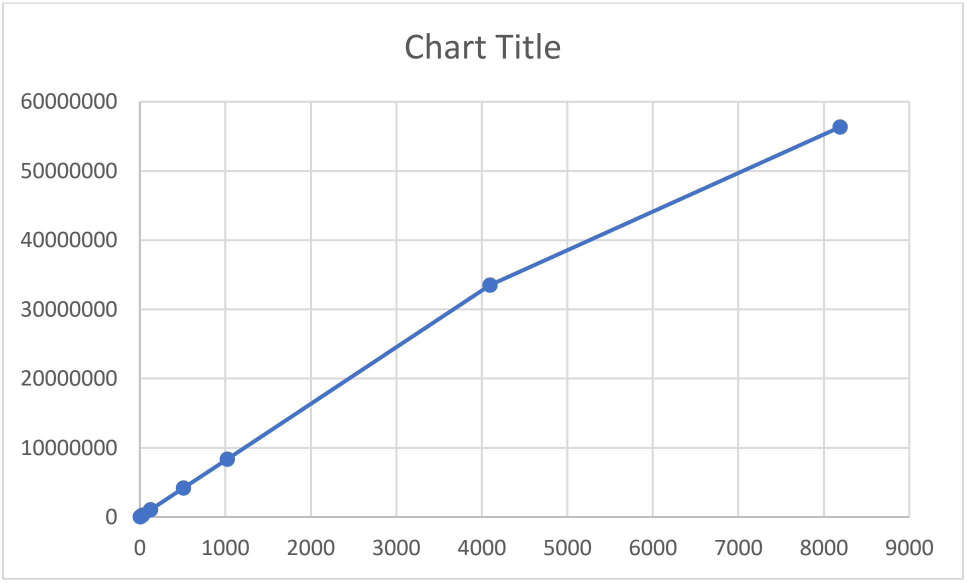
Hence overall algorithm complexity can be suggested as O(N * M) as explained previously O(A)+O(B) is maximum of O(A,B) and N will become less in comparison to N * M which is generally (N^2 because generally the length of the array is same as the input data set)

Graphs(Note I have considered the whole complexity of the program in which the find of a pair is done and then the union so theoretically it is N*M +M)

Input Data Set(N)	Counter	Theoritical (N+M * N)
8	65544	65544
32	262176	262176
128	1048708	1048704
512	4194812	4194816
1024	8389634	8389632
4096	33525796	33558528
8192	56352792	67117056

From the above Data , it can be seen that the theoretical value and the complexity counter value is almost same and for higher values data as of 8192 there is a difference because many were connected already hence for those values union operation was not performed.

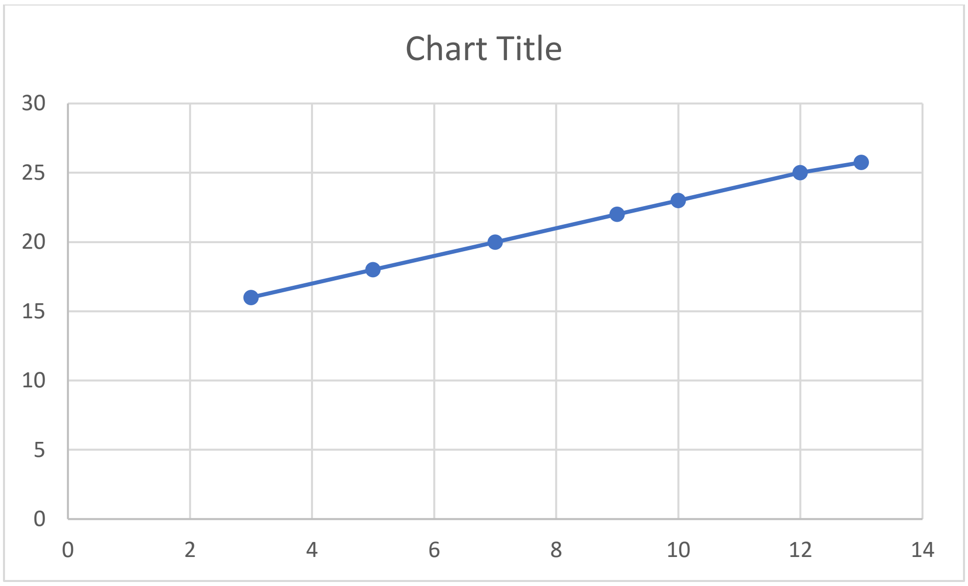
Graph



X-Axis : N i.e. size of the input dataset

Y-Axis : Complexity counter value

Log Log Plot



2. **Quick Union Algorithm**

Theoretical analysis of this algorithm suggest that in quick union algorithm our purpose is to have quick union – for this algorithm we are associating every element with the root i.e. the set element associated with the particular root are the one particular connected set.

So in order to connect the two unconnected set we just make the one rooted tree the sub tree of the other root .

In quick union we find the root of the two elements to be connected and the root value associated with the first element is changed to the root of the second element. So now the two elements reside under one root and hence are connected.

In the find function we have to find the root of the two elements and see whether they are same or not.

So complexity of the whole algorithm in the simplistic manner can be seen as :-

1. Union :- Complexity(Root_algo) + Complexity(Root_algo)
2. Find:- Complexity(Root_algo) + Complexity(Root_algo)

So root algo is the major time consuming operation in the algo which will decide upon the complexity or the cost of the algorithm.

Root algo complexity is maximum i.e. in the worst case M i.e. the size of the array.

So for N Pairs

For Union : it can also be maxim N *M

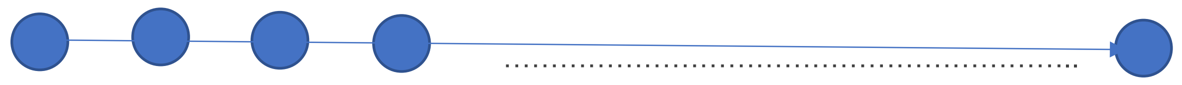
For Union : it can also be maxim N *M

(Please note this is in the worst case and i.e. why we say that this quick union is also slow when we look it in comparison to the previous algorithm of quick find as here also in the worst case the complexity reaches N * M in the worst case.)

So Run Time cost of overall algorithm seems to be proportional to the N * M in the worst case or can be said as O(N*M)

<u>Further Discussion to prove the above point</u>

Now the complexity of Root algo is in the worst case is N^2(or N* M) in the quick Union case as



1…………..2…………3………..4…………………………………………………………………….M

So here the element1 has root the Nth element has M-2 elements in between

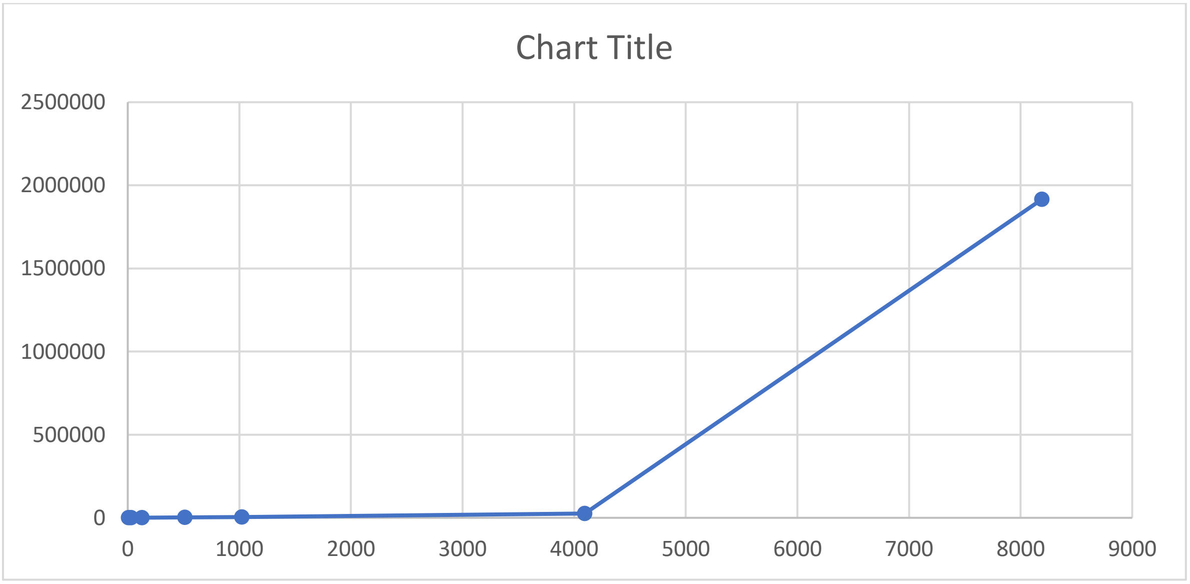
Please note that this is the worst case

Hence in our case the worst case function can be 4 * N*M with other constant time values

Input Data Set	Complexity Counter
8	32
32	128
128	516
512	2104
1024	4362
4096	25313
8192	1916710

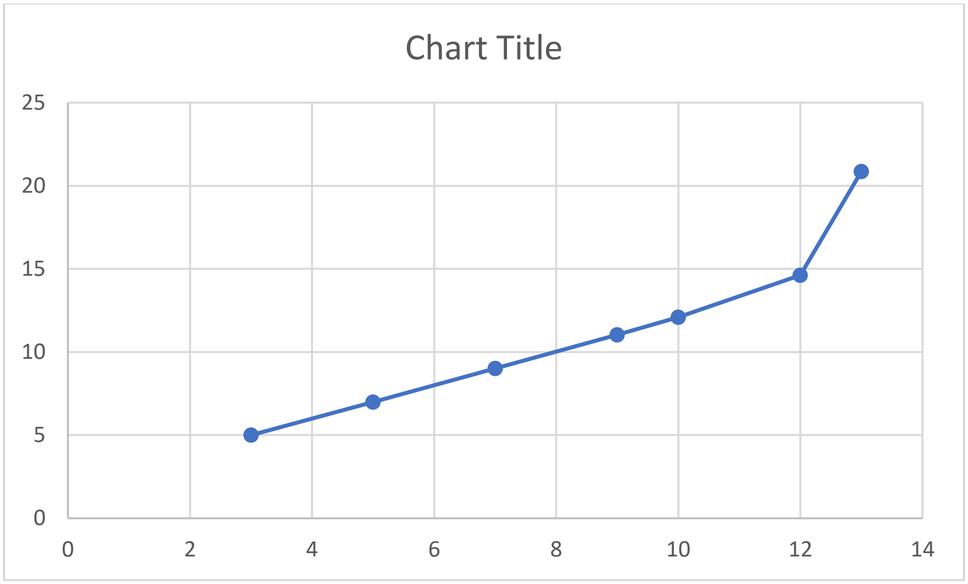
In the above Data Set , observations can be easily made that :-

1. As in our data sets of various values 8 , 32 …..to….. 1024 where the most pairs are not connected and only the roots , which are the values accessed by only array[element1] , are to be changed hence only one operation is required to connect or union the pairs the complexity is 4* N (4 is as the root is called 4 times) so basically proportional to N
2. But for 4096 and especially in 8192 where there are connected nodes and the root calculation happens then the complexity of the algorithms increases For example in case of 8192 it becomes (1916710) which is less than maximum (8192 * 8192 i.e. N *M) but still high.
3. From the graph below it can be seen that it was quite linear till 4096 and then for 8192 it suddenly jumped because of the fact as explained in the previous point.



(Note in order to cover all the Y Axis Values which range from low to very high such graph is produced)

But here it can be observed that till 1024 it is almost linear as explained above and for other values it is spiking as explained above.



4. Quick Union Weighted

The same as in the quick union algorithm , the working of this algorithm suggest that in quick union algorithm our purpose is to have quick union – for this algorithm we are associating every element with the root i.e. the set of elements associated with the particular root are the one particular connected set.

So in order to connect the two unconnected set we just make the one rooted tree the sub tree of the other root .

BUT THERE IS A DIFFERENCE BETWEEN THIS WEIGHTED ONE WITH THE QUICK UNION ONE AS DISCUSSED ABOVE.

Note that this algorithm almost works the same way as that of the previous quick Union algorithm but here not as in previous case just two trees are joined i.e. the root of one tree is not just made attached to the root of the destination element.

But the size of the tree of the one element is compared with the size of the tree of the destination element with whom to be connected. (size here considered are the number of elements in the tree)

So here the smaller tree just become the subtree of the larger tree.

This methodology just decreases the root finding steps in our this algorithm.

Complexity (Root_algo)

As learnt this logic from the course book and notes , Algorithms by Robert Sedgewick that when the smaller tree is connected to the larger tree then the base node height or you can say the depth of the lowest nodes can only be increased by 1 .

Further the larger tree with which this smaller one is getting connected is equal to or greater than this small tree.

The point to understand is that at max that suppose the one element in consideration is the lowest node in the smaller sub tree whose depth can also be maximum increased by Log N {Why? }

Because in the worst case for this lowest node is that when it is always the part of the smaller tree to be union and **in that case , note that this can happen Log N times**(?? Explained below) and the node can get down to depth log N.

Why ?? Because when there is a union of small + Big tree then big is at least equal or greater than the smaller one so the resultant tree is at least 2 * (size of the smaller one) or even greater than that.

So , S + 2* S + 4* S …………… for the worst case S is 1 and

1+2+4+……+N/2

N/((2^x)=N){ for 1} +2+4……….+N/4+N/2

So here in the above equation , x are the maximum steps for smaller tree to get down and x is clearly Log N(base 2).

Hence with detailed discussion above it is clearly stated that by joining the trees with their respective sizes comparison , the root of any node(considering the lowest one) can be found out by Log N steps to reach root.

Hence in the worst case (Big O is about the worst case cost function) , the Cost function will be proportional to Log N hence for this algorithm O(N Log N) { Note N is for N union operations}

Graphical Plots

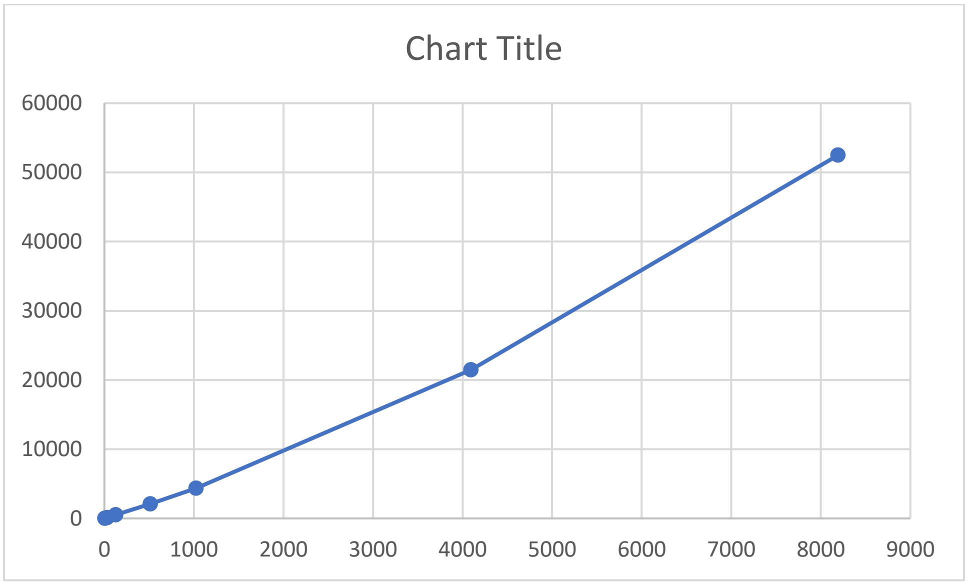
Input Data Set(N)	**Complexity Counter(T)**
8	32
32	128
128	516
512	2102
1024	4346
4096	21459
8192	52473

Log(N)	**Log(T)**
3	5
5	7
7	9.011227
9	11.03755
10	12.08547
12	14.3893
13	15.67929

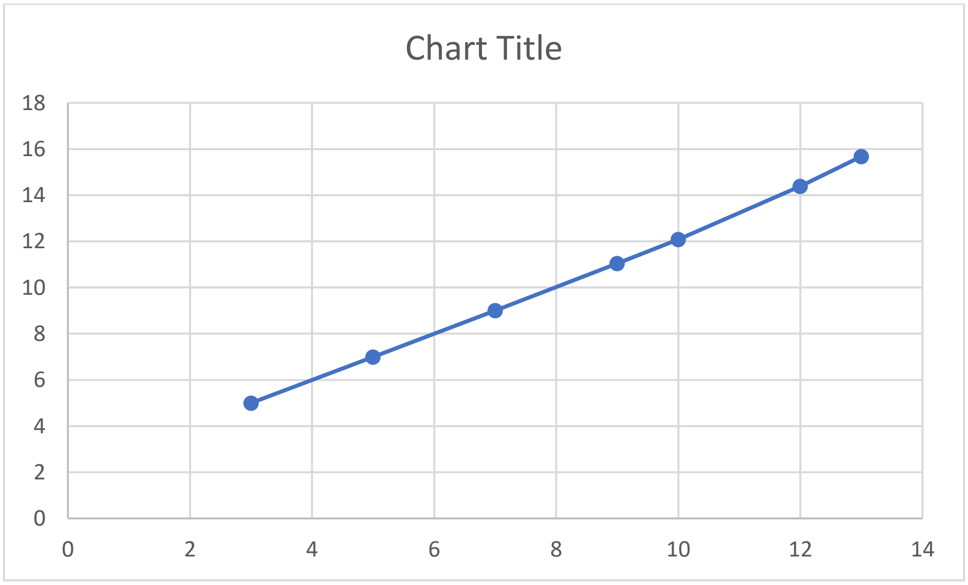
Observations

1. As seen from the observations above it can be seen that this algorithm is the most quickest of all the discussed ones.
2. Further as in the quick union here also the algorithm's complexity revolves around the root function complexity so here also the whole algorithm complexity is proportional to N(for N up to 1024) as in the quick union algorithm case (same explanation as only connections had to be done only one operation has to be done for union)
3. But as the N increases(i.e. 4096 to 8192) the complexity is lower in comparison to even the quick union because here the connection are made after considering the weight balancing and when this way the trees are joined i.e. the smaller is made the sub tree of the bigger one it is the fact that maximum depth of any node can only be Log N so maximum i.e. the worst case time for N pairs union can be N Log N

Graph:



Log Log Plot



Notes regarding program implementation

1. You just need to change the filename of the input pair data set , just replace the 8 ,32 etc. in front of the file name and please not change the Macro which defines 8192 the length of the array.
2. All files must be in the same folder as that of the program other wise the location of the file has to properly mentioned in the name of the file.
3. As asked in the question first , find operation whether connected is done and then the union operation is performed and the pair is printed
4. At last , counter value measuring the complexity is printed.

Q3. Recall the definition of "Big Oh" (where F(N) is said to be in O(g(N)), when F(N) < c (g(N)), for N > Nc) . Estimate the value of Nc for both Q1 and Q2.

More important than the specific value, is the process and reasoning you employ.

Solution:

Big O Notation definition is that :-

Suppose f(n) is the cost function then its cost can be termed as O(g(n))

where c times g(n) i.e. c g(n) is greater than f(n) for N > No (No is same as Nc in the question).

The purpose of this mathematics is that Big O notation i.e. O(g(n)) in above case represents an upper bound like in above case , whether what kind of input is i.e. worst case or easiest case,then the cost function f(n) will always remain under c * g(n) for all values greater than No or Nc.

So in order to answer this question , first we need to go through the cost functions in the worst case of each of the algorithms in the question 1 and question 2 and then using the definition of the Big O notation we have to show that O(g(N)) function stands out to be correct as estimated and determine for what values of N> (?)Nc

A. <u>Naïve Implementation of 3 Sum</u>

   In this algorithm , every 3 pair combination has to be checked for particular sum So , in such case the cost function is Nc3 i.e. (N(N-1)(N-2)/6)

   So f(n)= N^3/6 -N^2/2 +N/3

   Using the ~ (tilde) notation we can say that the cost of the algorithm is ~ N^3/6

   We can say this function seems to be in the order of N^3 so O(N^3) seems to be the big o notation of this function . So Let us see considering this as the proposition.

   <u>Mathematical Proof</u>

   Now if consider this quadratic equation :

   K * N^3 > N^3/6 -N^2/2 +N/3

   (K-1)/6*N^3 + N^2/2-N/3 > 0

   N * ((K-1)/6*N^2 + N^1/2-1/3) > 0

   As we know that input size can not be negative so hence only ((K-1)/6*N^2 + N^1/2-1/3) is positive then the whole equation will be greater than 0

So suppose k=1/6 and Not considering the N outside

N/2-1/3 > 0

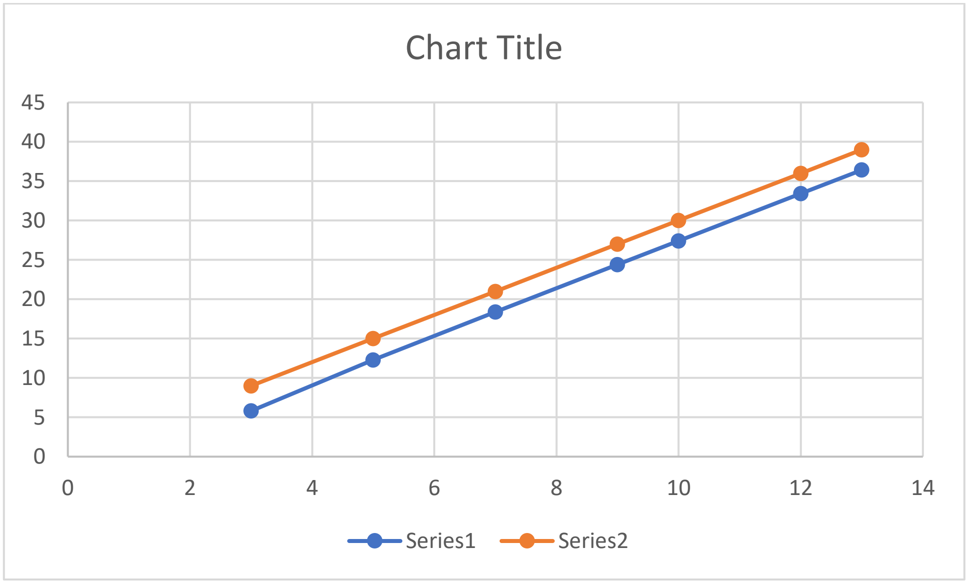
N > 2/3

From this we can say for all values of N greater than 0 i.e. from 1 onwards (as N take all natural numbers) for k =1/6 N^3/6 > (N(N-1)(N-2)/6)

Hence Proved that O(N^3) is valid for the cost function of this algorithm for values of N as described.

Graphically

S.No.	Input Size(n)	Log(n)	Counter Value (proxy for time/cost)(T)	Log(T)	Log (G(N)) Log(N^3) 3* Log N
1.	8	3	56	5.807354922	9
2.	32	5	4960	12.27612441	15
3.	128	7	341376	18.38100211	21
4.	512	9	22238720	24.40657042	27
5.	1024	10	178433024	27.41080741	30
6.	4096	12	11444858880	33.41398062	36
7.	8192	13	91592417280	36.41450911	39



As the two graphs are having the same slope and parallel hence the series 2 i.e. G(n) function is upper bound for all values of N>= 1

B. Sophisticated Implementation of 3 Sum

In the sophisticated implementation of the 3 Sum Problem , what we have done is the :

A. First by using the two loops we have gone through each pair and

B. Then from the rest of the array we have found the third element which is –(sum of above pair)

So our cost function in such case is:

A. Nc2 (N(N-1)/2)
B. Log N (Base 2) (Please it is to be noted that for simplification N is taken but in reality for each pair the array size is not always N for the searching of the third element But as our O notation is for the worst case so Log N can be taken as the worst cost for searching the third element)

Hence our f(n) becomes = N(N-1)/2 * Log N

We can say this function seems to be in the order of N^2 Log N so O(N^2 Log N) seems to be the big o notation of this function . So Let us see considering this as the proposition.

N will always be greater than 0 and in that case it can be easily seen that :

LHS i.e. N^2 Log N

RHS i.e. N(N-1)/2 * Log N

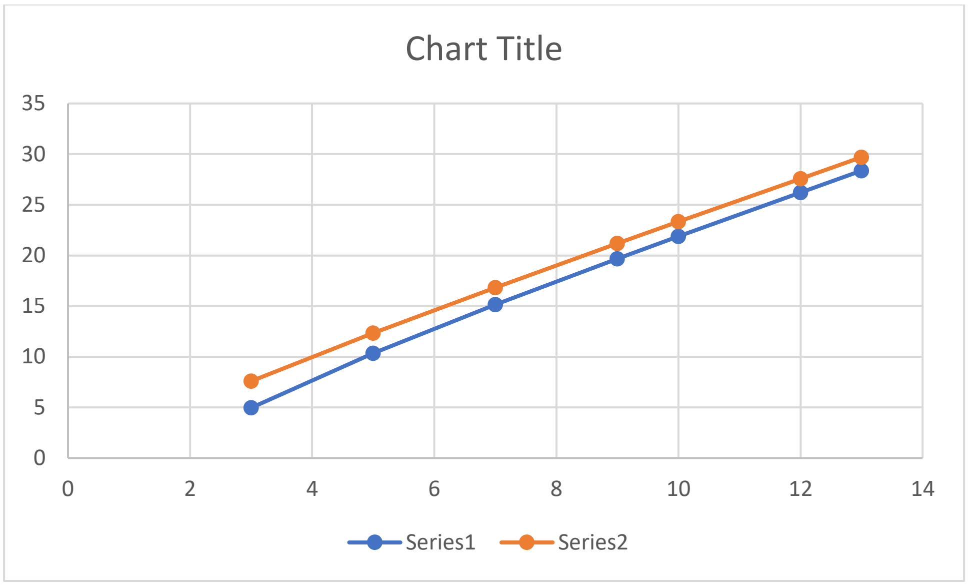
LHS > RHS for k >= 1 and for N > 0 (Note N can take Natural Numbers as the input 1,2, 3………..)

Hence our proposition stands proved .

Graphically can also be shown using the Observational Data in the first question :

S.No.	Input Size (n)	Log(n)	Counter Value (proxy for time/cost)(T)	Log(T)	G(N)= N^2 Log N Log(G(N))=2Log N +Log (Log N)
1.	8	3	31	4.95419631	7.584962501
2.	32	5	1307	10.35204343	12.32192809
3.	128	7	36075	15.13871177	16.80735492
4.	512	9	832939	19.66785132	21.169925
5.	1024	10	3850923	21.87677285	23.32192809
6.	4096	12	7.83223e+07	26.2229198	27.5849625
7.	8192	13	3.46791e+08	28.36949122	29.70043972

Now we see the scatter plot to see the comparison of the two right side columns with respect to input size



Series2 represent our Log representation of our O(G(N)) function and it can be clearly seen that actual cost graph for various values of Input is always less than the Series2(Log representation of our O(G(N)) function)

Hence our point that O(n^2 Log N) represents the Big O Notation of our cost function of this algorithm.

C. Quick Find

In the Quick Find algorithm as explained in the previous question the find operation is called for each pair so for N Pair the find or find connected function requires only one operation so the complexity function is N.

Union operation requires in the worst case that will have to access all the array , so for M size array and N pairs , it means the complexity of N*M.

So , total algorithm complexity is N+N*M.

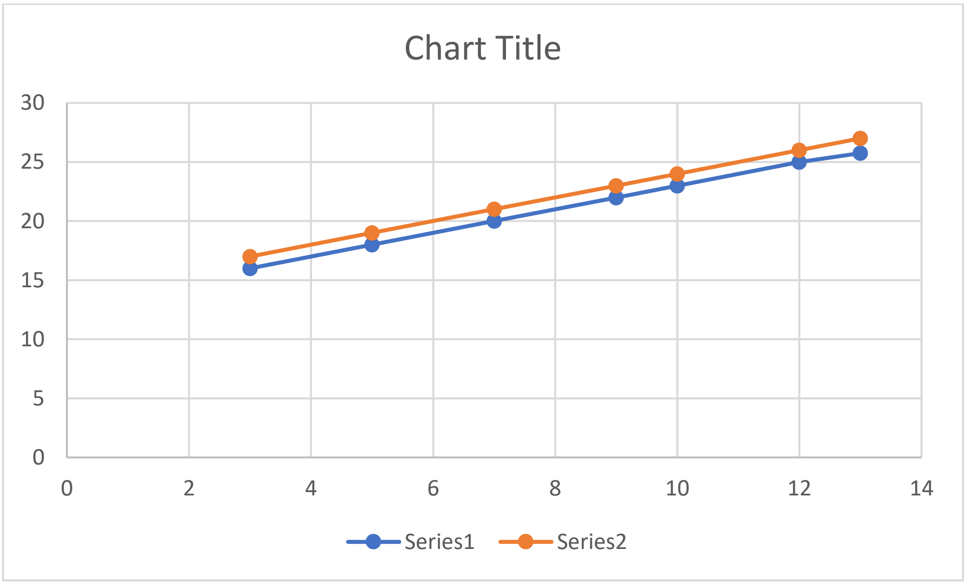
In case as in theoretical discussion as given in the course book , the notes , generally the size of array has size of N for N elements and in the worst case the Union operation requires N^2 operation .i.e. for union of all elements it has to search for all the elements of the array in order to change the one group label to another group (i.e. of the destination element). In the worst case it is supposed that destination element may be the one and the rest of the array is the another group , whose label has to be changed with the label of the destination group i.e. constituted of only the one element.

So by the above discussion , it has been clearly demonstrated that for the Union Operation in the quick find algorithm in the worst case scenario , the complexity O function is **O(N^2) i.e. for all values of N i.e. N>0** and for the Find Operation in the quick find algorithm in the worst case scenario , the complexity so O function is **O(N) i.e. for all values of N i.e. N>0 (obviously N will take the natural numbers from greater than equal to 1)**

Graphical Analysis(Empirical Analysis)

Input Data Set(N)	Counter(T)	Theoritical (N+M * N)	Log(N)	Log(T)	G(N)=N*M*K Log(G(N)=Log N +Log

					8192+Log K (when k=2 Log K =1)
8	65544	65544	3	16.00018	17
32	262176	262176	5	18.00018	19
128	1048708	1048704	7	20.00018	21
512	4194812	4194816	9	22.00017	23
1024	8389634	8389632	10	23.00018	24
4096	33525796	33558528	12	24.99877	26
8192	56352792	67117056	13	25.74798	27



As the two graphs are having the same slope and parallel hence the series 2 i.e. G(n) function is upper bound for all values of N>= 1

D. Quick Union

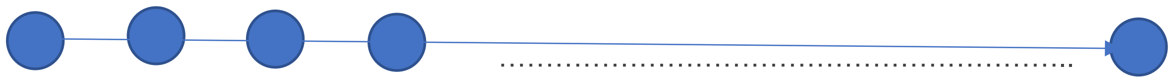
In the Quick Union algorithm , in both the find and the union functions the root algorithm is at the root of both because for find as discussed in the previous question the roots have to be found out and then has to be confirmed whether they are equal or not .

So as discussed in the previous question the complexity of the whole algorithm in the simplistic manner can be seen as :-

1. Union :- Complexity(Root_algo) + Complexity(Root_algo)
2. Find:- Complexity(Root_algo) + Complexity(Root_algo)

So as 4 times the Complexity(Root_algo).

Now the complexity of Root algo is in the worst case is N^2 in the quick Union case as



1……………2…………3…………4………………………………………………………………………N

So here the element1 has root the Nth element has N-2 elements in between

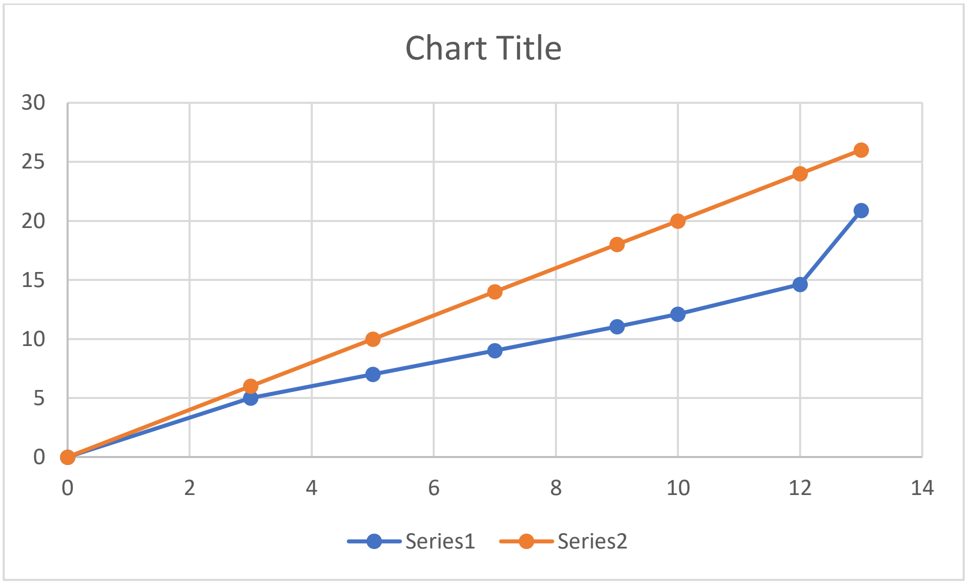
Please note that this is the worst case

Hence in our case the worst case function can be 4 * N^2 with other constant values

So in the worst case it can be written as **O(N^2) for all values of N greater than 0 (obviously N takes value in the natural numbers range) as K * N^2 will be greater than the worst case cost function of this algorithm for K>=1**

<u>Graphically this can also be seen :-</u>

Input Data Set(N)	Counter(T)	Log(N)	Log(T)	G(N)=N^2 i.e. O(N^2) Log(G(N))=2Log N
8	32	3	5	6
32	128	5	7	10
128	516	7	9.011227	14
512	2104	9	11.03892	18
1024	4362	10	12.09077	20
4096	25313	12	14.62759	24
8192	1916710	13	20.8702	26



It can also been seen from the above log log plot of the G(N) Function i.e. the O function and the log log plot of the cost data we got from our program execution(i.e. the empirical analysis) that :

Series 2 graph that is of the G(N) function is the upper bound graph that is the Series1 graph is bounded by the Series 2 graph which is the worst case for all values of N>= 1 for which Log N =0

So , with the discussion and logical analysis we come to particular G (N) function and that is the worst case cost function which is valid for all values of N >=1 and now through our empirical studies we have also shown that our discussion and logical reasoning was right.

E. **Quick Union Weighted**

In the **Quick Union Weighted** algorithm , in both the find and the union functions the root algorithm is at the root of both because for find as discussed in the previous question the roots have to be found out and then has to be confirmed whether they are equal or not .

So as discussed in the previous question the complexity of the whole algorithm in the simplistic manner can be seen as :-

1. **Union :-** Complexity(Root_algo) + Complexity(Root_algo)
2. **Find:-** Complexity(Root_algo) + Complexity(Root_algo)

So as 4 times the Complexity(Root_algo).

Note that this algorithm discussion is going in the same way as that of the previous quick Union algorithm but here not as in previous case two trees are not just joined i.e. the root of one tree is not just made attached to the root of the destination element. But the size of the tree of the one element is compared with the size of the tree of the destination element with whom to be connected.

So here the smaller tree just become the subtree of the larger tree.

This methodology just decreases the root finding steps in our this algorithm.

Complexity (Root_algo)

As learnt this logic from the course book and notes , Algorithms by Robert Sedgewick that when the smaller tree is connected to the larger tree then the base node height or you can say the depth of the lowest nodes can only be increased by 1 .

Further the larger tree with which this smaller one is getting connected is equal to or greater than this small tree.

The point to understand is that at max that suppose the one element in consideration is the lowest node in the smaller sub tree whose depth can also be maximum increased by Log N Why? Because the worst case for this lowest node is that when it is always the part of the smaller tree to be union and **in that case this can happen Log N times**(?? Explained below) and the node can get down to depth log N.

Why ?? Because the when small + Big then big is at least equal or greater than the smaller one so the resultant tree is at least 2* smaller or greater than that.

So , S + 2* S + 4* S …………… for the worst case S is 1 and

1+2+4+……+N/2

N/((2^x)=N){ for 1} +2+4……….+N/4+N/2

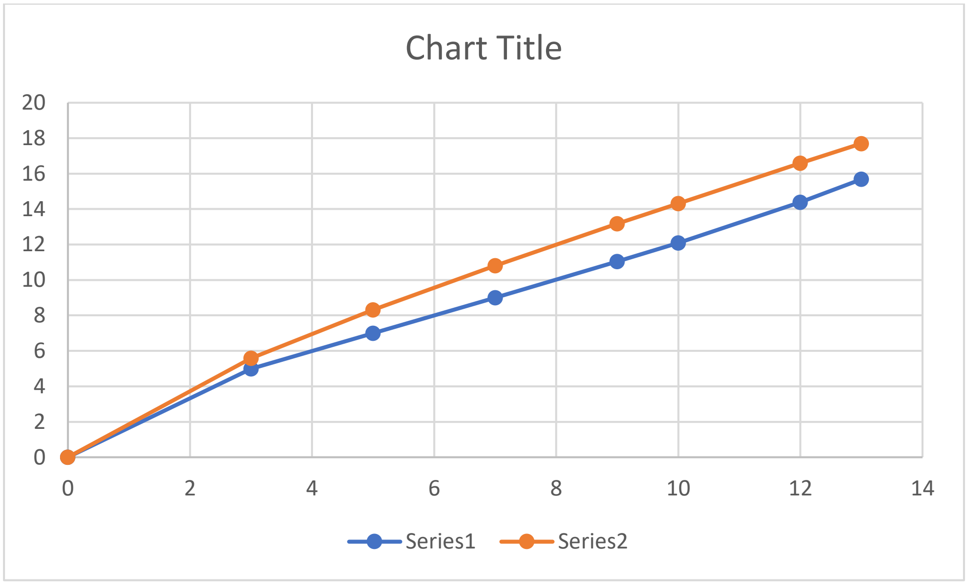
So here in the above equation , x are the maximum steps for smaller tree to get down and x is clearly Log N(base 2).

Hence with detailed discussion above it is clearly stated that by joining the trees with their respective sizes comparison , the root of any node(considering the lowest one) can be found out by Log N steps to reach root.

Hence in the worst case (Big O is about the worst case cost function) , the Cost function will be proportional to Log N hence for this algorithm big O notation is O(N Log N) { Note N is for N union operations} where K * N Log N > Cost function which is (4* Log N + Constant values) for all values of N>=1.

Graphically we can visualize it for our Empirical Analysis

Input Data Set(N)	Complexity Counter(T)	Log(N)	Log(T)	G(N)=K*N*LogN Log(G(N))=Log N +LogLogN +Log K(with k =2)
8	32	3	5	5.584963
32	128	5	7	8.321928
128	516	7	9.011227	10.80735
512	2102	9	11.03755	13.16993
1024	4346	10	12.08547	14.32193
4096	21459	12	14.3893	16.58496
8192	52473	13	15.67929	17.70044



It can also been seen from the above log log plot of the G(N) Function i.e. the O function and the log log plot of the cost data we got from our program execution(i.e. the empirical analysis) that :

Series 2 graph that is of the G(N) function is the upper bound graph that is the Series1 graph is bounded by the Series 2 graph which is the worst case case for all values of N> 1 for which Log N =0

So , with the discussion and logical analysis we come to particular G(N) function which is valid for all values of N and that is the worst case cost function and now through our empirical studies we have also shown that our discussion and logical reasoning was right.

Q4: Farthest Pair (1 Dimension): Write a program that, given an array a[] of N double values, find a farthest pair: two values whose difference is no smaller than the difference of any other pair (in absolute value). The running time of the program should be LINEAR IN THE WORST CASE.

Solution:

Program : Q_4_Farthest_Pair

<u>Logic of the program</u>

For one dimension numbers we have to find the farthest pairs , so mainly we have to find the min , max the pair in the linear time.

So in this we move to each element.

First we pick the two elements and put them in min and max variable (box) and then moves through the whole array in order to maintain the min and max .

For this with each element we compare max if greater than max , that variable (box) is replaced with this number as it is greater than the max and if any number is less than min then min is replaced with this number.

So max and min boxes are maintained and then the difference between the max and min is the distance i.e. the farthest distance between any two elements in the array.

Output :

a. **Displayed the distance between the max and min element and hence farthest distance and the two elements who are the max and min.**
b. **Displayed the number of times loop executed i.e. the counter for checking the algorithm complexity which is less than or equal to length of the array hence complexity O(N) as it can also been seen easily in the program with only one loop and accessing all elements of the array only once max .**

Note:

1. The input array is hardcoded .
2. Please change the macro N for the value of number of elements in the array
3. Then change the input array in the cpp files
4. And then run to see the output for the desired input array.

Please take care of above for running the algorithm for the desired input array.

Q5. Faster-est-ist 3-sum: Develop an implementation that uses a linear algorithm to count the number of pairs that sum to zero after the array is sorted (instead of the binary-search based linearithmic algorithm). Use the ideas to develop a quadratic algorithm for the 3-sum problem

Solution:

An implementation that uses a linear algorithm to count the number of pairs that sum to zero after the array is sorted :

Program : Q5a

Logic:

a. First the insertion sort is done
b. Our goal was to have then that all pairs with the sum equal to 0 in the array to be found out and that too in the linear time.
c. For this what we have done is that from left and right (i.e. min and max we have taken) then till when the left element and right element is greater than 0 we make max turn towards left that is made decreasing till the sum is equal to the 0 or less than that and when this becomes less than that we increase the left side value that is moving towards the right basically increasing the value.

See by the first step by moving the right element towards the left i.e. decreasing the index we have tried to remove those pairs that are outside the test value that is 0 in our case. Then when our sum of pairs reach less than the test value(0 in our case) we increase the smaller side.

And when the left and right side becomes equal we then just increases the left and decreases the right index and then again carry on the same process in the remaining array .

Analogy can be of weighing balance where in first heavy and lightest ones are put there.

If the weight exceed the desired one then decrease the heaviest side weight till less than or equal to the desired weight is reached .

By this those heavy weights are out skirted because even after their sum with the minimum value they are not able to achieve the test sum that is those numbers will not be able to form any pairs .

then when the after decreasing the heavy weights , sum decreases than the desired one then that means that if any pair with the lightest one was to be performed has been done so the lighter side is increased further.

In case of reaching the test sum the weight exact higher than the lighter one and exact lower than the highest one are taken and the process is repeated.

d. **Output :**

   a. We have printed the pairs amounting to the 0.
   b. Count of the number of pairs.
   c. **Counter value which will be less than the (N) number of elements of the array or list of numbers which proves that our algorithm works on the sorted list in less than linear time so maximum O(N) complexity.**

Note for the execution of the program:

1. First that the array is hardcoded.
2. You need to change the values of the array.
3. Using the size method program automatically calculates the number of elements and then using this number works further.
4. **So Please change the values of the array in the cpp file for changing the input set.**
5. **Note that as it was not mentioned so it is assumed that there are distinct numbers in the input array.**

Quadratic algorithm for the 3-sum problem

The same logic to find the 3 sum equal to 0.

Just we move through the above loop to linearly move towards the each element and then for the remaining array we use the above two sum method to find the two numbers whose sum is equal to the –(first loop picked number) so that three number sum become equal to the 0.

Output :

- We have printed the pairs amounting to the 0.
- Count of the number of pairs.
- **Counter value which will be less than the (N^2) where N is the number of elements of the array or list of numbers which proves that our algorithm works on the sorted list in less than quadratic time so maximum O(N^2) complexity.**

Note that as the insertion sort always have complexity O(N^2) that too in the worst case so that O(N^2){For Insertion sort} + O(N^2){ for the rest of the quadratic algorithm as explained} so the overall complexity will be max k* N^2 so O(N^2).

Note that is why in above our counter was always looking for the second part in order to prove that complexity of whole algorithm is O (N^2)

Note for the execution of the program:

1. First that the array is hardcoded.
2. You need to change the values of the array.
3. Using the size method program automatically calculates the number of elements and then using this number works further.
4. **So Please change the values of the array in the cpp file for changing the input set.**
5. **Note that as it was not mentioned so it is assumed that there are distinct numbers in the input array.**