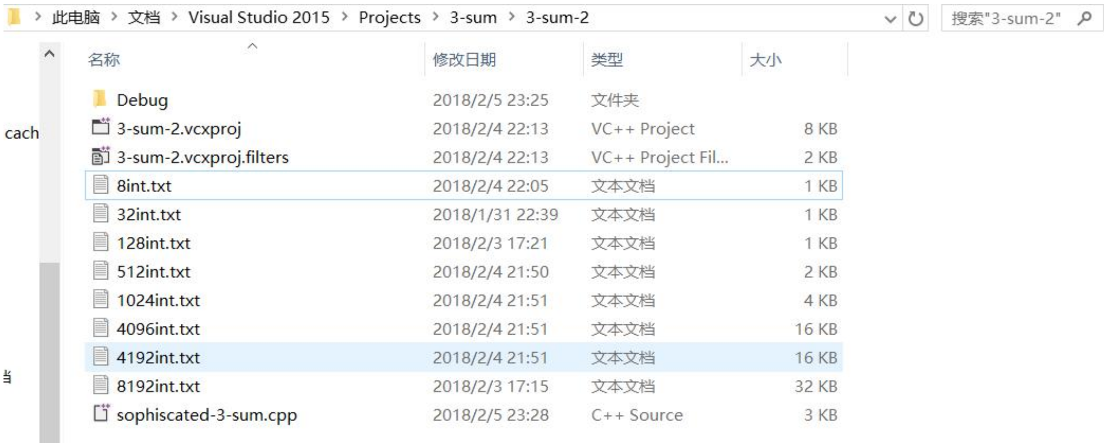
HomeWork 1

Xinyu Lyu 182007396

Instructions：IDE: VS2015 C++

Q1.naive-3-sum/sophiscated-3-sum:

Instructions: First you need to create the project, to process the program you need to change the file name in the "void load_number(int a[])" function. The test txt file can't be simply add to the resource file folder in the IDE, it MUST be manually put under the same catalog together with the debug folder in the project workspace. Also you need input the number of integer in the naive-3-sum/sophiscated-3-sum.cpp in your test file. Like this figure.



Because when using vector as a container it is too slow to load the data when testing the data file with 8192 int. So, I use the original array to reduce the run time. And the original array need you to set the size, sorry for inconvenient.

Q2.quick-find/quick-union/WeightQuickUnion:

For this program, first you need to create the project, than you need to create a project and add "quick-find.h"/"quick-union.h"/ "WeightQuickUnion.h" to head file folder,"quick-find.cpp"/"quick-union.cpp" /"WeightQuickUnion.cpp","main.cpp" these two files into the resource folder in VS 2015 IDE, to change the file name in the member function "void UF::load(vector<int>&a, vector<int>&b)" in the"quick-find.cpp"/"quick-union.cpp"/"WeightQuickUnion.cpp".

The test txt file can't be simply add to the resource file folder in the IDE, it MUST be manually put under the same catalog together with the debug folder in the project workspace.

Q4.Faster-3-sum:

Instructions: For this program you need to change the file name in the "void load_number(vector<int>&a);" function, The test txt file can't be simply add to the resource file folder in the IDE, it MUST be manually put under the same catalog together with the debug folder in the project workspace.

Q5.Farthest Pair:

For this program you need to in put the number of the test numbers. Then follow the instructions to add the test numbers one by one.

Questions and Answers:

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 resource (hw1-1.data.zip) to determine the run time cost of your implementations as function of input data size. Plot and analyze (discuss) your data.

	8int	32int	128int	512int	1024int	4096int	4192int	8192int
3-sum(O(N^3))	0ms	0ms	16ms	188ms	983ms	50563 ms	64968 ms	463578 ms
3-sum(O(N^2 lg N))	oms	0ms	0ms	31ms	125ms	2500ms	2641ms	7672ms

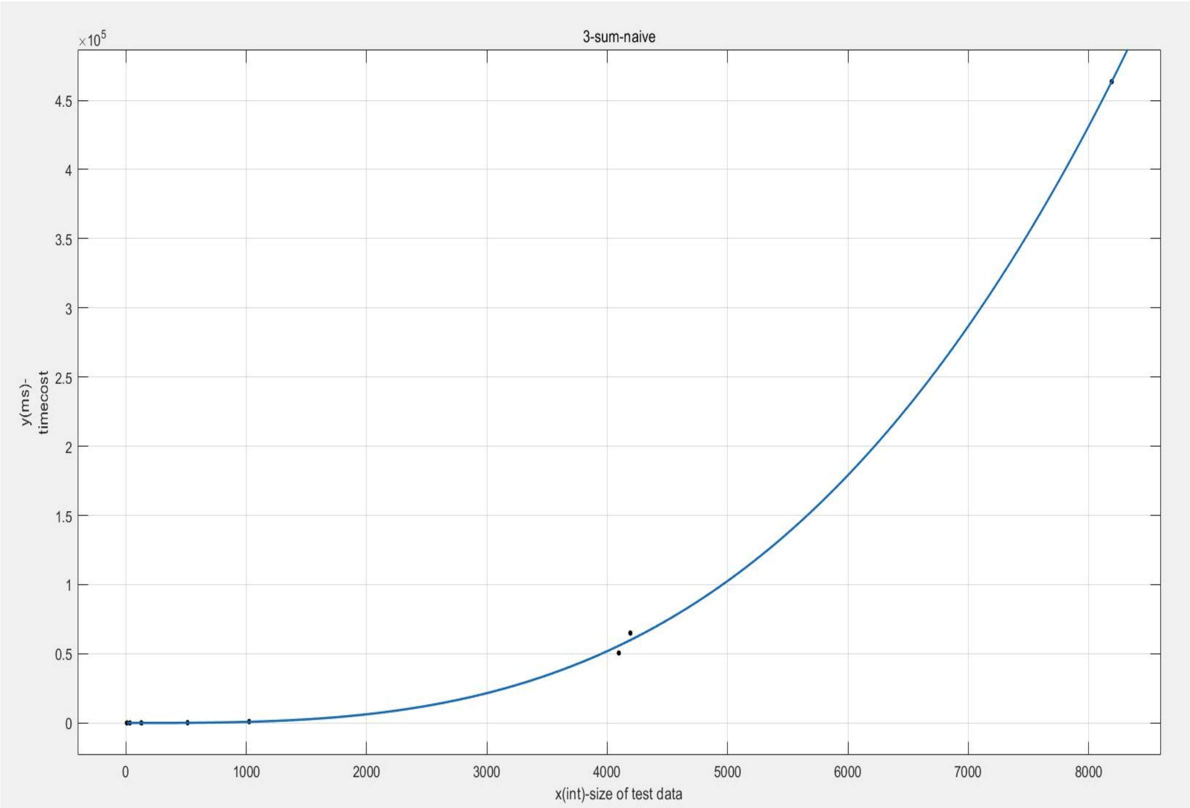


Figure.1

The run time cost of naive 3-sum implementation grows with 3rd-power with the increase of the size of test data from 8-int to 8192-int when using data from hw1-1.data. When running the data from hw1-1.data, the return value "int count" for the 3-sum and 3-sum-2 projects are both 0 for in all the txt flies the figures are all positive. Therefore, running with such data, we can get the running time cost in the worst case-O(N^3). In order to test the function, I add another "extra 8-int" and the result is right. The best case is just like the worst case with Ω(N^3), for in the naive 3-sum we need to traverse all the data in array with 3 loops like the worst case.

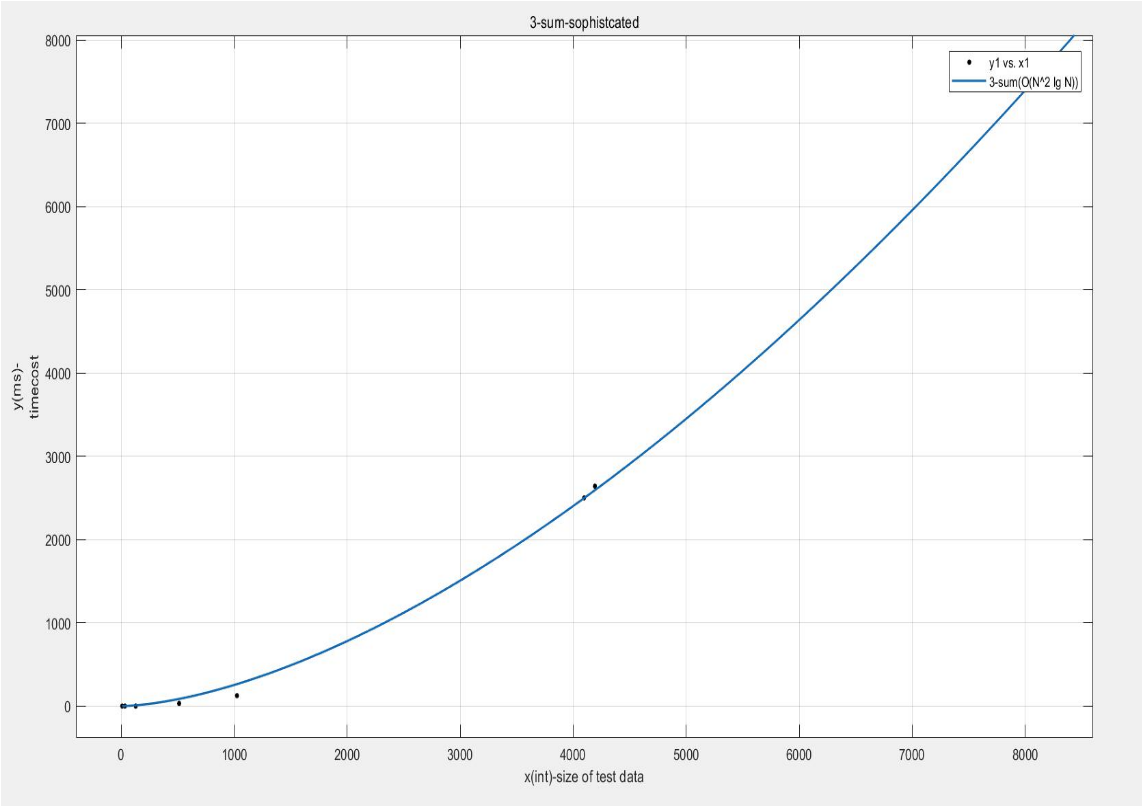


Figure.2

The run time cost of sophisticated 3-sum implementation grows with (N^2*logN)-power with the increase of the size of test data from 8-int to 8192-int, when using data from hw1-1.data. From the figure, we can clearly see that, unlike Figure.1, the curve in Figure.2 increases more smoothly than the former one. That is because running in the worst case the complexity of the sophisticated 3-sum implementation is O(N^2*logN) withsort (N^2) and binary search(logN) and 2 loops for the traverse.

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 (hw1-2.data.zip, under resources) 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

// read in a sequence of pairs of integers (each in the range 1 to N) where N=8192

// calling find() for each pair: If the members of the pair are not already connected

// call union() and print the pair.

	8pair	32pair	128pair	512pair	1024pair	4096pair	8192pair
Quick-find	92ms	406ms	1665ms	5561ms	13905ms	62247ms	106094ms
Quick Union	6ms	32ms	726ms	3144ms	4750ms	15513ms	29037ms
Weight Quick Union	4ms	36ms	716ms	3070ms	4637ms	15324ms	25711ms

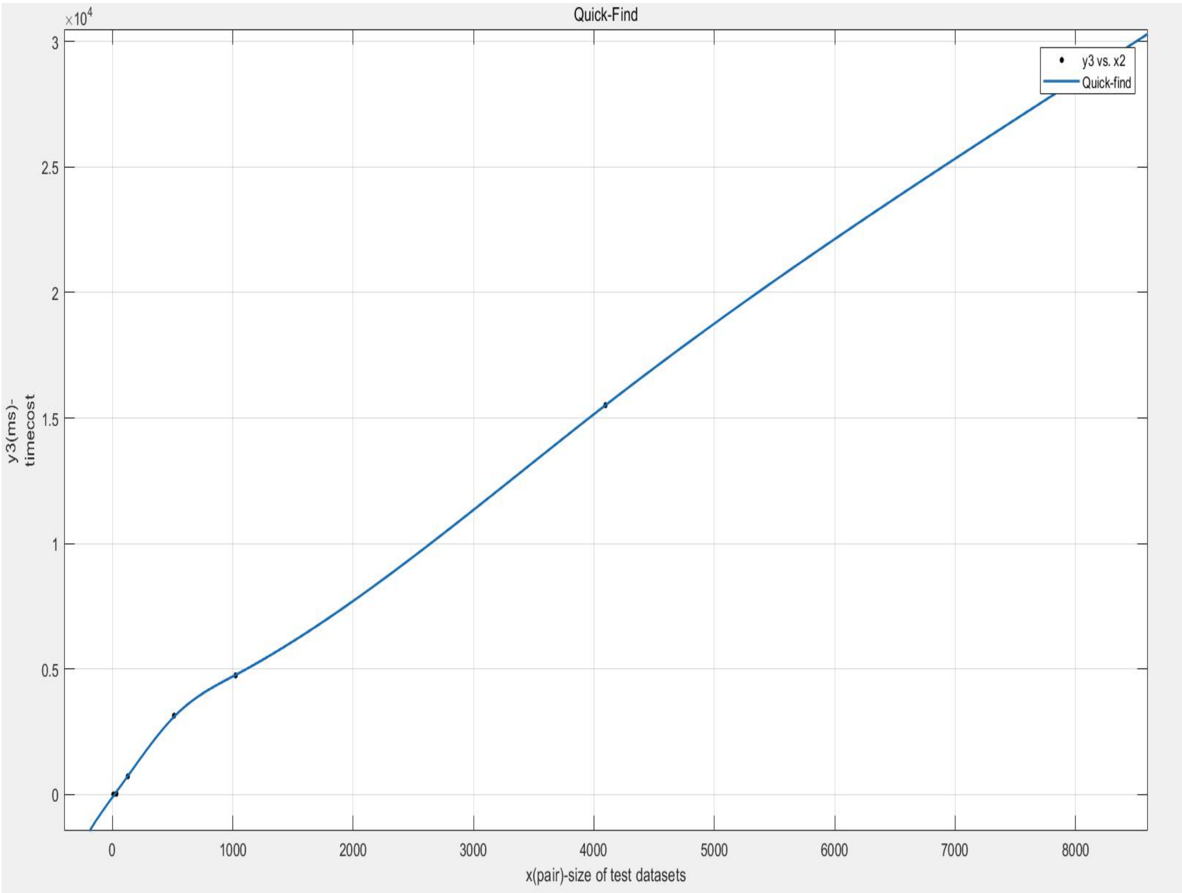


Figure.3

Every running time cost of each test point in the plot is obtained by running all the data in a single txt file. Y is the total running time for each txt file from the Quick-Find part.The runtime cost of Quick-Find implementation grows linearly with the increase of the size of test data from 8-pair to 8192-pair in hw1-2.data. The given data pairs are all random figures which means the id[i] for every figure is itself. The case is among the best case in which the two figures in every pair have already been connected Ω（1）and the worst case O(N) that N-1 figures share the same id[i]. However, the test case shares the same complexity with the worst case for you need to traverse the whole array not connected before. And the linear relationship implies that the complexity for Quick-find is linear with check if connected (N) and for the union(N) with traverse(N) and two finds(1) inside.

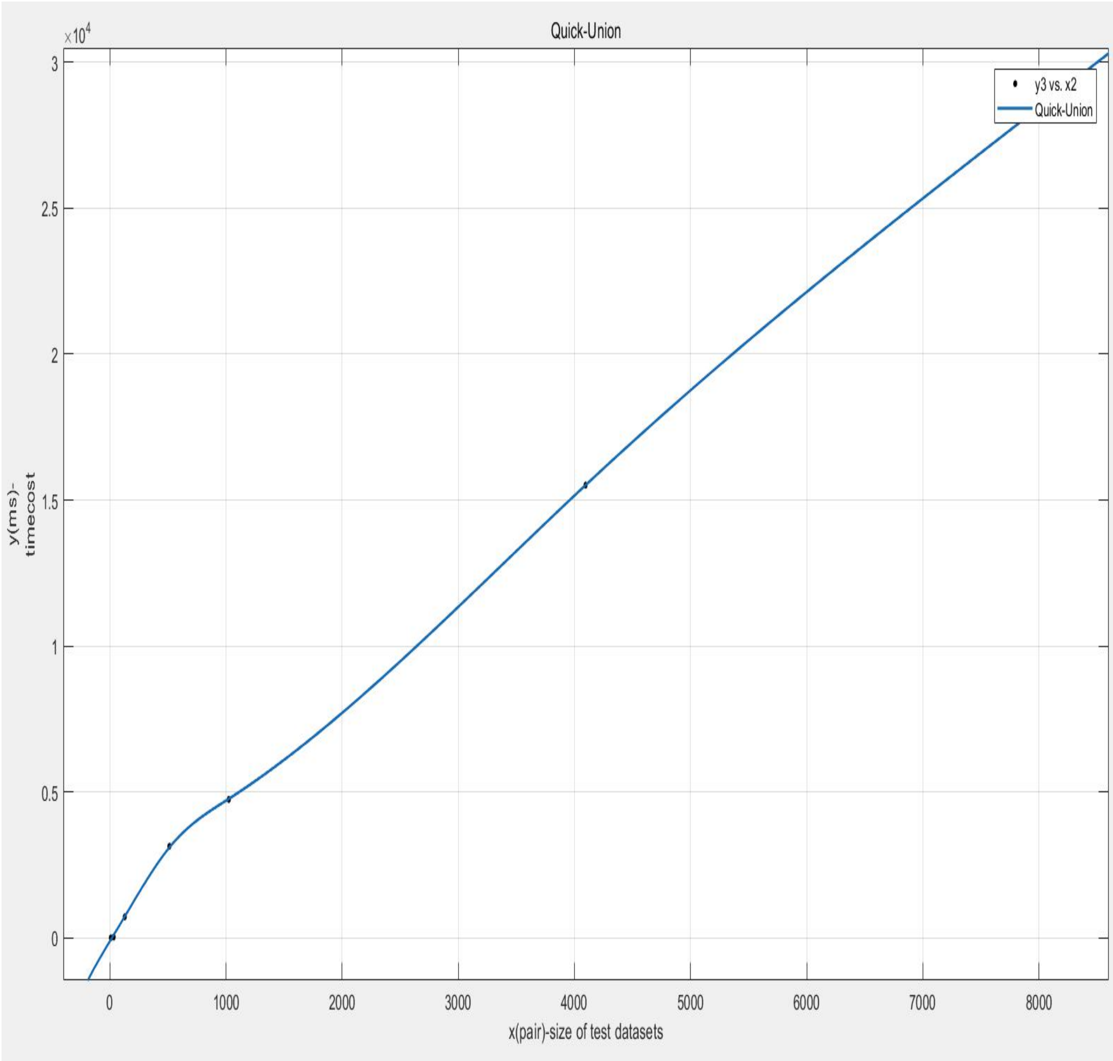


Figure.4

Every running time cost of each test point in the plot is obtained by running all the data in a single txt file. Y is the total running time for each txt file from the Quick-Union part.The runtime cost of Quick-Union implementation grows linearly with the increase of the size of test data from 8-pair to 8192-pair when using data in hw1-2.data. The given data pairs are all random figures which means the root of every figure is itself. The case is among the best case in which the two figures in every pair have already been connected with the same root as Ω（1）and the worst case O(N) (N is the depth of the tree) that N-2 points share the same root like a one-direction list when the last pair wants to make a connection. However, the test case shares the same complexity with the worst case for, N is the number of pairs, not the depth of the tree. Because in this case for Quick-Union, the times for the access of array is 5 with 2 from connected checking and 3 fron the union in each Quick-Union operation. With the constant N as 8192, the total running time increases linearly with the size of the test datasets.

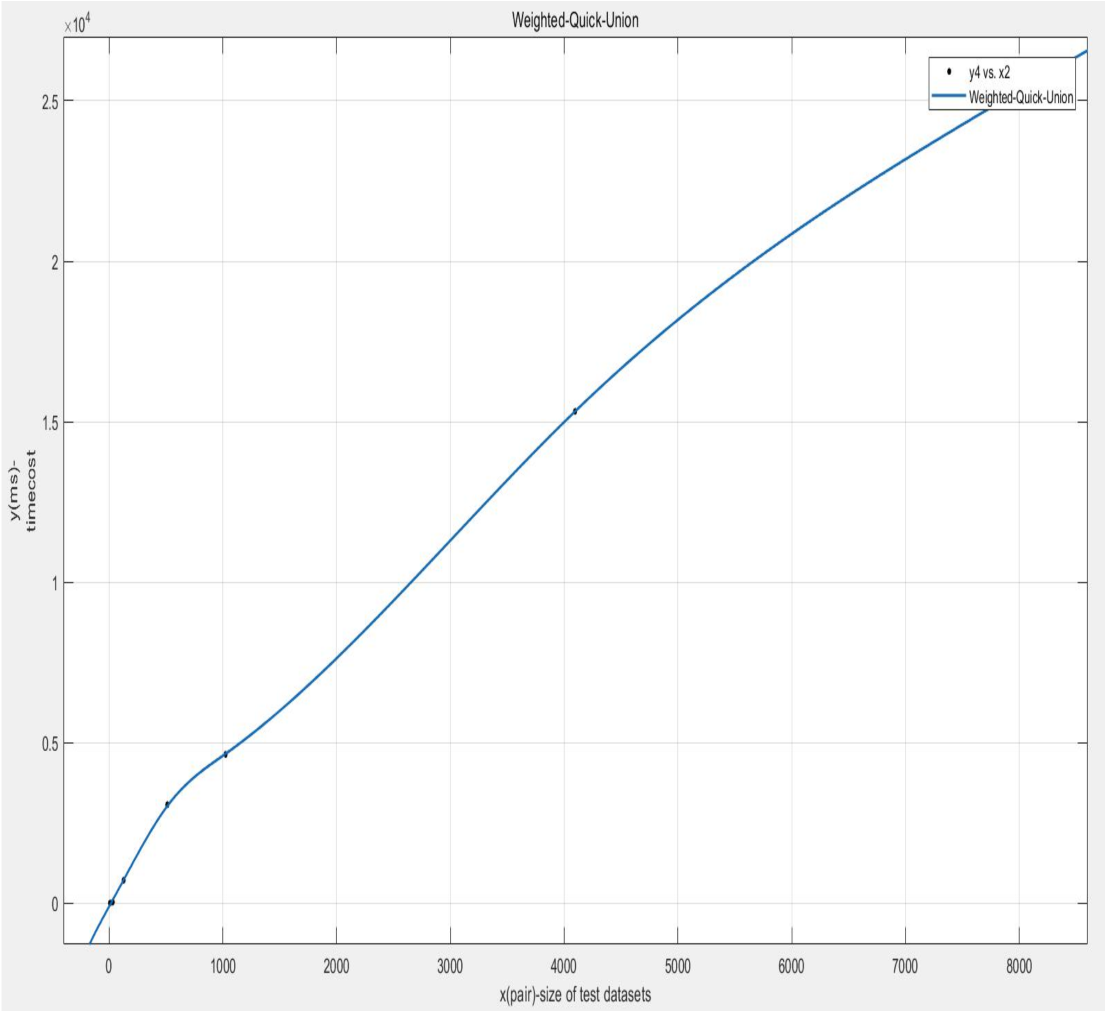


Figure.5

Every running time cost of each test point in the plot is obtained by running all the data in a single txt file. Y is the total running time for each txt file from the Weighted-Quick-Union part.The runtime cost of Weighted-Quick-Union implementation grows logarithm with the increase of the size of test data from 8-pair to 8192-pair when using data in hw1-2.data. The given data pairs are all random figures which means the root of every figure is itself. The case is among the best case in which the two figures in every pair have already been connected with the same root as Ω（1） and the worst case O(logN) that two trees with the same depth (logN-1) when the last pair, a child from each tree, wants to make a connection. However, the test case shares the same complexity with the worst case. Because in this case for Quick-Union, the times for the access of array is 4logN+5 with 4logN from find function and 5 from union function of size compare in each Weighted-Quick-Union operation.

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 your employ.

Q_3 ① naive-3-sum (worst-case)

$O(N^3)$ has analized the worst-case in Question 1

$$f(N) = \frac{N(N-1)(N-2)}{3!}$$ array access in 3-loops.

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

from $f(N) = O(g(N))$ we can get

$$\frac{N(N-1)(N-2)}{3!} = O(N^3)$$

$\exists N_c \ \forall N \geq N_c \ |f(N)| \leq c|g(N)|$

that is:

$$\frac{|N(N-1)(N-2)|}{3!} \leq c|N^3|$$

let $N_c = 1$

$$\frac{|N^3-3N^2+2N|}{6} \leq \frac{N^3+3N^2+2N}{6}$$

$$\leq \frac{N^3+3N^3+2N^3}{6}$$

$$= 1N^3$$

$c=1$, $N_c=1$

② sophisticated-3-sum (worst-case)

$O(N^2\log_2 N)$ has analized the worst-case in Q1

$$f(N) = \frac{N(N-1)}{2!}(1+\log_2 N) + \frac{3}{2}N^2$$

$\frac{N(N-1)}{2!}$: array access in two-loops

$\frac{N(N-1)}{2!}(\log_2 N)$: total array access in binary search in two-loops

$\frac{N^2}{2}$: $\frac{N^2}{2}$ the times for compare/replace in Insertionsort in each insertionsort, there are 3 times to access the array. tips: the second reference in while of the same element will not cost twice.

from $f(N) = O(g(N))$ we can get

$$\frac{N(N-1)}{2!}(1+\log_2 N) + \frac{3}{2}N^2 \leq c|N^2\log_2 N|$$

$\exists N_c \ \forall N \geq N_c \ |f(N)| \leq c|g(N)|$

$$\left|\left(\frac{3}{2}N^2 + \frac{N^2-N}{2}\right)c(1+\log_2 N)\right| \leq c|N^2\log_2 N|$$

let $N_c = 2$

$$\left|\frac{3}{2}N^2 + \left(\frac{N^2-N}{2}\right)(1+\log_2 N)\right| \leq \frac{N^2+N}{2}(1+\log_2 N) + \frac{3N^2}{2}$$

$$\leq 2N^2 + \frac{N}{2} + \frac{N^2+N}{2}\log_2 N$$

$$\leq \left(2N^2+\frac{N}{2}\right)\log_2 N + \left(\frac{N^2+N}{2}\right)\log_2 N$$

$$= \left(\frac{5N^2}{2}+N\right)\log_2 N$$

$$= \left(\frac{5N^2}{2}+\frac{N^2}{2}\right)\log_2 N$$

$$= \frac{7}{2}N^2\log_2 N$$

$c = \frac{7}{2}$, $N_c = 2$

③ Quick Find (worst-case)

$O(N)$ $f(N) = 2N+3$ worst-case has analyzed in Question 2.

check connected $= 2 * \text{find} = 2$

Union $= 2 * \text{find} + N$ (traverse array) $+ N-1$ (change id for elements)

$$f(N) = 2 + 2 + N + N - 1 = 2N+3$$

from $f(N) = O(g(N))$

$$O(N) = 2N+3$$

$\exists N_c \ \forall N \geq N_c \ |f(N)| \leq c|g(N)|$

$$|2N+3| \leq c|N|$$

let $N_c = 1$

$$\leq 2N+3N = 5N$$

$c=5$, $N_c=1$

④ Quick-Union (worst-case) ~~analyzed~~ analyze in Question 2 of worst-case.

$O(N)$ $f(N) = 2N+1$

$N \rightarrow$ number of elements

$f(N) = (N-1) + 1 + 1 + N-1 + 1$

$N-1$ = find(p) in union tips: in while the second reference of the same element will cost time

1: find(q)

1: replace the root

$N-1+1$ = two finds in check connected

$f(N) = 2N+1$

$O(N) = 2N+1$ $\exists N_c \ \forall N \geq N_0 \ |f(N)| \leq c\,|g(N)|$

$|2N+1| \leq c\,|N|$

let $N_c = 1$

$\leq 2N + N = 3N$

$c = 3,\ N_c = 1$

⑤ weight-quick-union (worst-case) analyze in Question 2 about the worst case.

$O(N)$ $f(N) = 4\log_2 N + 5 = 2\log_2 N + 2\log_2 N + 5$

$2\log_2 N$: 2* find root. in union function.

$2\log_2 N$: 2* find root in check if connected function

5: array access of tree size compare and modify in union function

$\exists N_c \ \forall N \geq N_c \ |f(N)| \leq c\,|g(N)|$

$|4\log_2 N + 5| \leq c\,|N|$

let $N_c = 2$

$\leq 4N + \frac{5}{2}N$.

$= \frac{13}{2}N$

$c = \frac{13}{2},\ N_c = 2$.

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.

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