CS 325 Winter 2019

HW 1 – 30 points

1) (6 pts) For each of the following pairs of functions, either f(n) is O(g(n)), f(n) is Ω(g(n)), or f(n) is Θ(g(n)) best describes the relationship. Select one and explain.

a. $f(n) = n^{0.75}$; $g(n) = n^{0.5}$

f(n) is Ω(g(n))

limit method:

the limit as n approaches infinity for the fraction $(n^{0.75})/(n^{0.5})$

$(n^{0.75})/(n^{0.5}) = n^{0.75-0.50} = n^{0.25}$ which will trend towards ∞

$\infty\ then\ f\ (n)\ is\ \Omega(g\ (n)\ )$

f(n)= Ω (g(n))

$n^{0.75}$ is always greater than $n''^{0.5}$ when n is a positive integer greater than 1

f(n) "≥" g(n) implies: f(n)= Ω (g(n))

b. f(n) = log n; g(n) = ln n

f(n) is Θ(g(n))

$$\Theta(g(n)) = \{f(n) : \text{ there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .$$

$C_1 = 0.2$ $C_2 = 1$ $n_0 = 2$ satisfies the above proposition

$0 \le C_1 \ln(n) \le \log(n) \le C_2 \ln(n)$

limit method:

the limit as n approaches infinity for the fraction (log n)/(ln n)

(log n)/(ln n)

This was my initial answer:

~~f(n)=O(g(n))~~

~~log(n) is always less than than ln(n) when n is a positive integer greater than 1~~

~~f(n) "≤" g(n)~~

c. f(n) = nlog n; g(n) =n(sqrt(n))

f(n)=O(g(n))

log(n) will always be less than sqrt(n)

multiplying by n is a linear change

so n log(n) will always be less than n sqrt(n)

f(n) "≤" g(n)

d. f(n) = e^n; g(n) = 3^N

f(n) is O(g(n))

limit method:

the limit as n approaches infinity for the fraction (e^n)/(3^n)

(e^n)/(3^n)= $(e/3)^n$ ~= $(2.718281828459045235360 28/3)^n$ which will trend towards 0

Same as $(2/3)^n$

0 $then\ f\ (n)\ is\ O(g\ (n)\ )$

e. $f(n) = 2^n$; $g(n) = 2^{n-1}$

f(n) is Θ(g(n))

Okay so, g(n) will always be half of f(n) right? Because what ever g(n) is f(n) is nothing more than g(n) times 2. If n is 9 then we have 2*2*2*2*2*2*2*2*2 for f and g(n) is the same but one less *2. That would mean that the difference is constant and not expanding. So I would guess f(n) is Θ(g(n)).

If we take the limit method:

as n approaches infinity for $(2^n)/(2^{n-1})$

$(2^n)/(2^{n-1}) = (2 * 2^{n-1})/(2^{n-1}) = 2$

thus we have a c constant greater than 0 so

$c > 0 \; then \; f \; n \; is \; \Theta(g \; n \;)$

f. $f(n) = 4^n$; $g(n) = n!$

f(n) is O(g(n))

Based on big Oh Classes

G(n) is factorial which is a much faster rate of growth compared exponential especially 4^n

So f is upper bounded by g

2) (4 pts) Let f_1 and f_2 be asymptotically positive non-decreasing functions. Prove or disprove each of the following conjectures. To disprove give a counter example.

a. If $f_1(n) = \Omega(g(n))$ and $f_2(n) = O(g(n))$ then $f_1(n) = \Theta(f_2(n))$.

False

Counter Example:

g(n) for the counter example will be n

f1(n) will be n^2

this allows f1(n) = Ω(g(n)) based on Big Oh class

f2(n) will be log n

this allows f2(n) = O(g(n))

f1(n) = theta (f2(n)) is impossible given all this because it at large n values log n will never be greater than n and n will never be greater than n^2

given the dramatic differences in functions and the basic premise it is clear that there are many counter examples to disprove conjecture 'a' above. And there are no examples to attempt to prove this case at all when n is larger than 1 or 3 or so.

b. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then $f_1(n)+f_2(n) = O(g_1(n) + g_2(n))$

True

Proof:

1- $f_1(n) = O(g_1(n))$ by definition of Big Oh this means that $f_1(n) \leq c_1*g(n)$ for all $n >= n_0$ for some n_0 and c_1 that is greater than 0

2- $f_2(n) = O(g_2(n))$ by definition of Big Oh this means that $f_2(n) \leq c_2*g(n)$ for all n >=n0 for some n_0 and c_2 that is greater than 0

3- This means that by simple rules of addition $f_1(n) + f_2(n) \leq c_1*g_1(n) + c_2*g_2(n)$

4- This is because if a is less than b and c is less than d. And we say x is b minus a, and we knows that because a is less than b x must be positive. The same is true for c and d and y, instead of x. Thus a + c will always be exactly x+y less than b and d. Because x and y must be positive always here we can say that the conjecture is certainly true.

4) (10 pts) Merge Sort vs Insertion Sort Running time analysis

a) Modify code- Now that you have verified that your code runs correctly using the data.txt input file, you can modify the code to collect running time data. Instead of reading arrays from the file data.txt and sorting, you will now generate arrays of size n containing random integer values from 0 to 10,000 to sort. Use the system clock to record the running times of each algorithm for ten different values of n for example: n = 5000, 10000, 15000, 20,000, …, 50,000. You may need to modify the values of n if an algorithm runs too fast or too slow to collect the running time data (do not collect times over a minute). Output the array size n and time to the terminal. Name these new programs insertTime and mergeTime.

Submit a copy of the timing programs to TEACH in the Zip file from problem 3, also include a "text" copy of the modified timing code in the written HW submitted in Canvas.

insertTime.cpp

```
#include <fstream>
#include <iostream>
#include <chrono>

// C program for insertion sort
#include <stdio.h>
#include <math.h>

using std::chrono::system_clock;

/* Adapted from www.geeksforgeeks.org/insertion-sort */
/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
	int i, key, j;
	for (i = 1; i < n; i++)
	{
		key = arr[i];
		j = i - 1;

		/* Move elements of arr[0..i-1], that are
		   greater than key, to one position ahead
		   of their current position */
		while (j >= 0 && arr[j] > key)
		{
			arr[j + 1] = arr[j];
			j = j - 1;
		}
		arr[j + 1] = key;
	}
}
```

```
// A utility function to print an array of size n
void printArray(std::ostream &stream, int arr[], int n)
{
	for (int i = 0; i < (n - 1); i++)
	{
		stream << arr[i] << " ";
	}
	stream << arr[n - 1];
	stream << std::endl;

}




int main()
{
	srand(time(NULL));


	for (size_t size = 5000;size <= 50000;size=size+5000)
	{
		int* array = new int[size];

		for (size_t j = 0; j < size; j++)
		{
			array[j] = rand() % 10001;
		}

		system_clock::time_point start = system_clock::now();
		insertionSort(array, size);
		system_clock::time_point end = system_clock::now();

		std::chrono::duration<double> duration = end - start;

		std::cout << size << " " << duration.count() <<std::endl;
	}
	std::cin.get();
}
```

Mergetime.cpp

```
#include <fstream>
#include <iostream>
#include <chrono>

// C program for insertion sort
#include <stdio.h>
#include <math.h>

using std::chrono::system_clock;

/* Adapted from www.geeksforgeeks.org/merge-sort/ */

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
	int i, j, k;
	int n1 = m - l + 1;
	int n2 = r - m;

	/* create temp arrays */
	int* L = new int[n1];
	int* R = new int[n2];

	/* Copy data to temp arrays L[] and R[] */
	for (i = 0; i < n1; i++)
		L[i] = arr[l + i];
	for (j = 0; j < n2; j++)
		R[j] = arr[m + 1 + j];

	/* Merge the temp arrays back into arr[l..r]*/
	i = 0; // Initial index of first subarray
	j = 0; // Initial index of second subarray
	k = l; // Initial index of merged subarray
	while (i < n1 && j < n2)
	{
		if (L[i] <= R[j])
		{
			arr[k] = L[i];
			i++;
		}
		else
		{
			arr[k] = R[j];
			j++;
		}
		k++;
	}

	/* Copy the remaining elements of L[], if there
	   are any */
	while (i < n1)
	{
		arr[k] = L[i];
		i++;
```

```
            k++;
        }
        /* Copy the remaining elements of R[], if there
           are any */
        while (j < n2)
        {
            arr[k] = R[j];
            j++;
            k++;
        }
}

/* l is for left index and r is right index of the
   sub-array of arr to be sorted */
void mergesort(int arr[], int l, int r)
{
        if (l < r)
        {
            // Same as (l+r)/2, but avoids overflow for
            // large l and h
            int m = l + (r - l) / 2;

            // Sort first and second halves
            mergesort(arr, l, m);
            mergesort(arr, m + 1, r);

            merge(arr, l, m, r);
        }
}

// A utility function to print an array of size n
void printArray(std::ostream &stream, int arr[], int n)
{
        for (int i = 0; i < (n - 1); i++)
        {
            stream << arr[i] << " ";
        }
        stream << arr[n - 1];
        stream << std::endl;

}



int main()
{
        srand(time(NULL));


        for (size_t size = 5000; size <= 50000; size = size + 5000)
        {
            int* array = new int[size];

            for (size_t j = 0; j < size; j++)
            {
                array[j] = rand() % 10001;
            }
```

```
        system_clock::time_point start = system_clock::now();
        mergesort(array, 0,size-1);
        system_clock::time_point end = system_clock::now();

        std::chrono::duration<double> duration = end - start;

        std::cout << size << " " << duration.count() << std::endl;
    }
    std::cin.get();
}
```

b) Collect running times - Collect your timing data on the engineering server. You will need at least eight values of t (time) greater than 0. If there is variability in the times between runs of the same algorithm you may want to take the average time of several runs for each value of n. Create a table of running times for each algorithm.

insertion sort

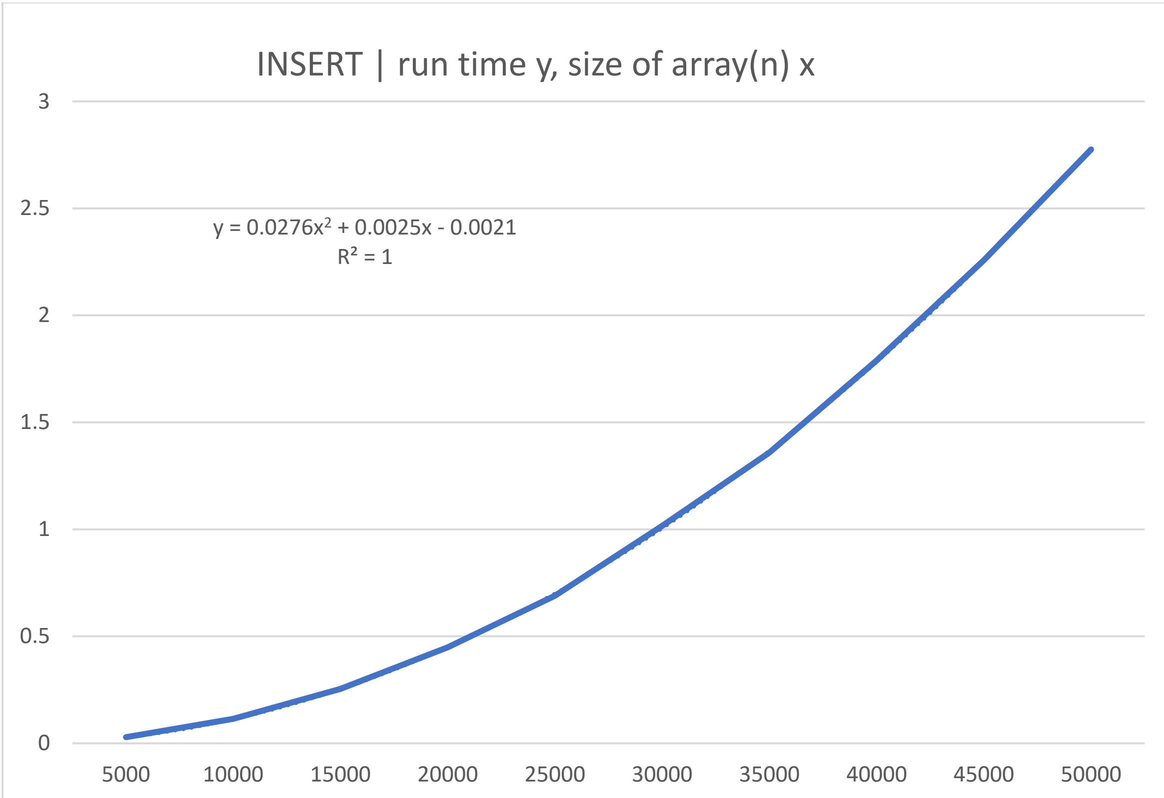
5000	0.0286228
10000	0.114225
15000	0.253971
20000	0.448621
25000	0.689177
30000	1.01623
35000	1.35893
40000	1.78979
45000	2.25759
50000	2.77542

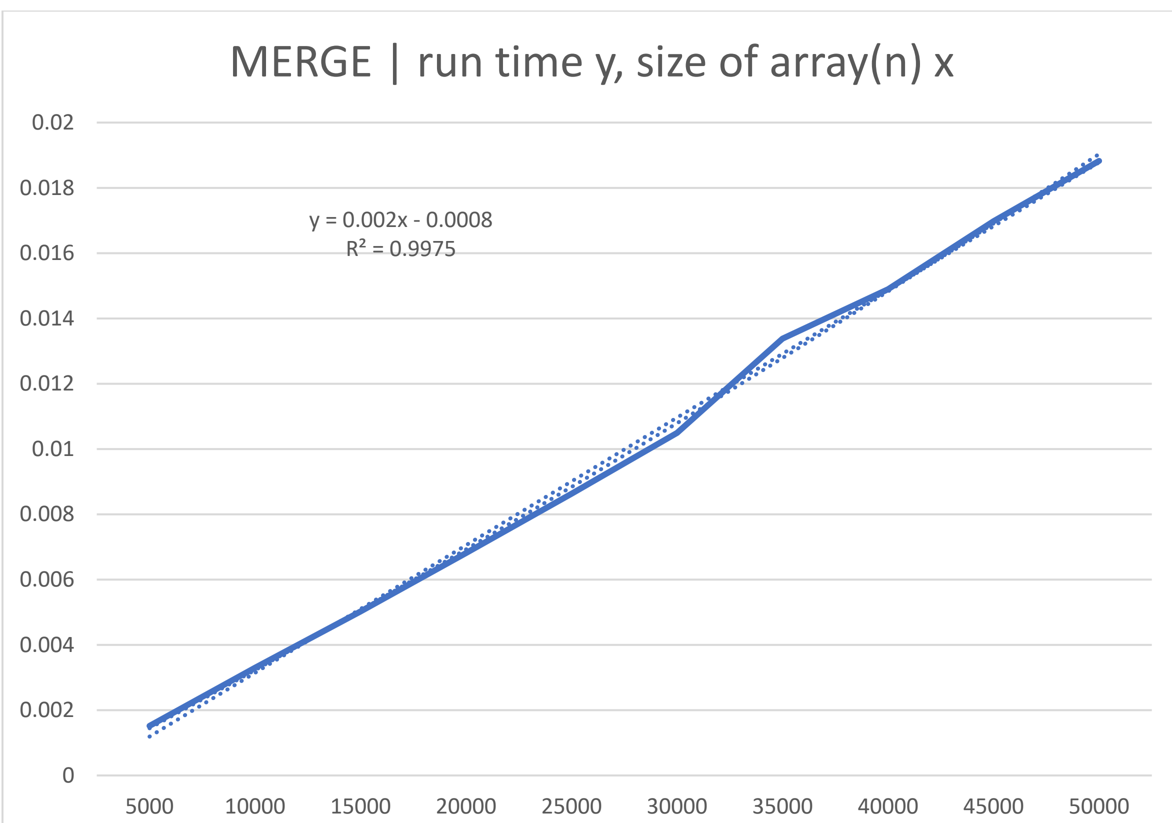
Merge sort

5000	0.00151828
10000	0.00329631
15000	0.00502149
20000	0.00681149
25000	0.00862973
30000	0.0104942
35000	0.0133825
40000	0.0148964
45000	0.0169792
50000	0.0188256

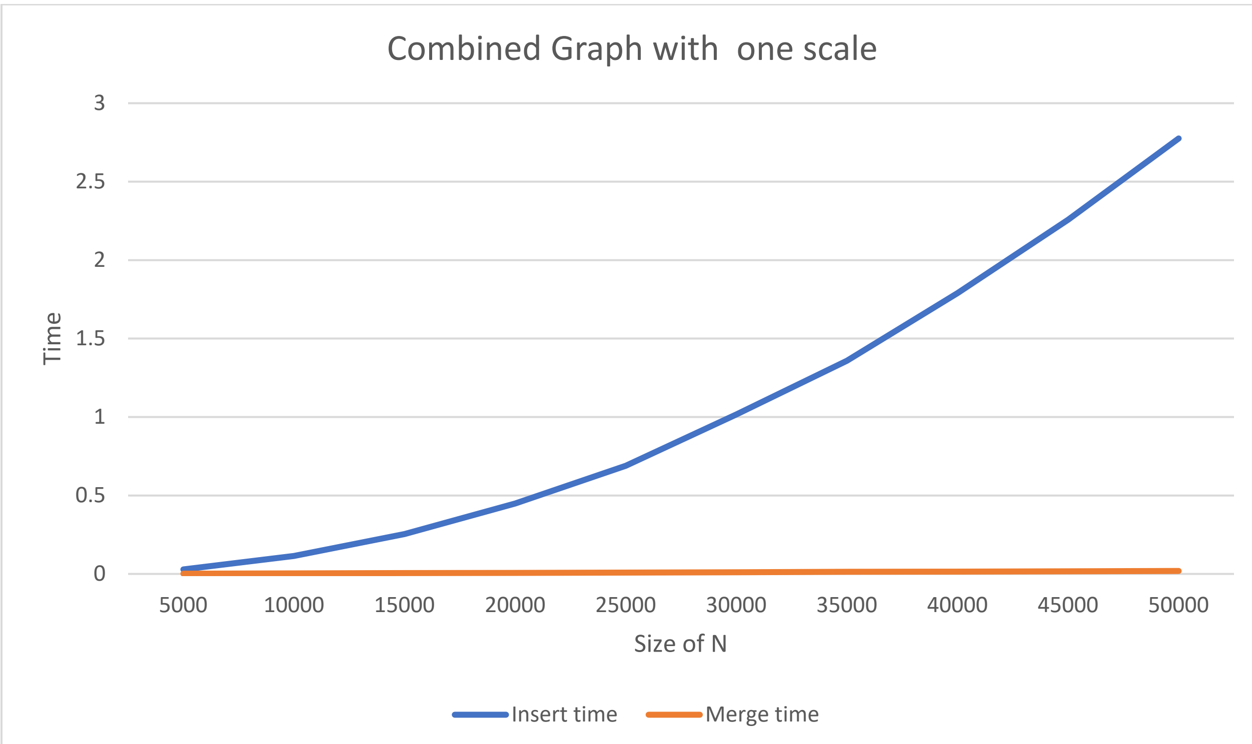
c) Plot data and fit a curve - For each algorithm plot the running time data you collected on an individual graph with n on the x-axis and time on the y-axis. You may use Excel, Matlab, R or any other software. What type of curve best fits each data set? Give the equation of the curves that best “fits” the data and draw that curves on the graphs.

BELOW


INSERT | run time y, size of array(n) x
y = 0.0276x² + 0.0025x - 0.0021
R² = 1
3
2.5
2
1.5
1
0.5
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
50000


MERGE | run time y, size of array(n) x
y = 0.002x - 0.0008
R² = 0.9975
0.02
0.018
0.016
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
50000

d) Combine - Plot the data from both algorithms together on a combined graph. If the scales are different you may want to use a log-log plot.



e) Comparison - Compare your experimental running times to the theoretical running times of the algorithms? Remember, the experimental running times were the "average case" since the input arrays contained random integers.

Given that the experimental data should be an average case scenario and the theoretical run times are worst case scenario the data seems to be what was expected.

For insert sort we have $f(n) = 0.0276x^2 + 0.0025x - 0.0021$ which is essentially $O(n^2)$

For merge sort we have $f(n) = 0.002x - 0.0008$ which is essentially $O(n)$

Thus the experimental data agrees with the theoretic.