Homework 1 (CS 325)

1). **Name: Hao (Jeff) Deng**

a). $\lim_{n\to\infty} \frac{f(n)}{g(n)} \rightarrow \lim_{n\to\infty} \frac{n^{0.75}}{n^{0.5}} = \infty$, therefore f(n) is Ω(g(n)) because when using limits, f(n) tends to grow faster than g(n). The asymptotic notations are O when f(n)/g(n) is not infinity; Θ is when f(n)/g(n) is not 0 nor infinity; Ω is when f(n)/g(n) is not 0.

b). $\lim_{n\to\infty} \frac{f(n)}{g(n)} \rightarrow \lim_{n\to\infty} \frac{\log n}{\ln n} = \frac{1}{\ln(10)}$, therefore f(n) is Θ(g(n)) because after taking the limit, the result is a constant. After the division, the ln takes in the base case of log, which is 10 by default, so the result is 1/ln(10).

c). $\lim_{n\to\infty} \frac{f(n)}{g(n)} \rightarrow \lim_{n\to\infty} \frac{\mathrm{n}\log n}{n\sqrt{n}} = 0$, therefore f(n) is O(g(n)) because f(n) grows slower than g(n) so the result of the limit is 0. In the limit equation, both the n from top and bottom will cancel each other out, that leaves with log n divided by √n. √n grows faster than log n.

d). $\lim_{n\to\infty} \frac{f(n)}{g(n)} \rightarrow \lim_{n\to\infty} \frac{e^n}{3^n} = 0$, therefore f(n) is O(g(n)) because f(n) grows slower than g(n). We can take the n out, that leaves with (e/3)^n, which is around 0.93^n, the constant is less than 1 so it converges to 0.

e). $\lim_{n\to\infty} \frac{f(n)}{g(n)} \rightarrow \lim_{n\to\infty} \frac{2^n}{2^{n-1}} = 2$, therefore f(n) is Θ(g(n)) because the result after taking the limit is a constant. The top part can be split into (2^{n-1}*2), then the 2^{n-1} can be cancelled out with the bottom one, that will leave the top with just 2, so the result of this limit is 2.

f). $\lim_{n\to\infty} \frac{f(n)}{g(n)} \rightarrow \lim_{n\to\infty} \frac{4^n}{n!} = 0$, , therefore f(n) is O(g(n)) because f(n) grows slower than g(n); n! grows faster than 4^n according to the complexity growth rate.

2).

a). I disagree with this statement of $f_1(n) = \Omega(g(n))$ and $f_2(n) = O(g(n))$ then $f_1(n) = \Theta(f_2(n))$. We know that f1(n) >= g(n), 0 <= cg(n) <= f1(n), and f2(n) <= g(n), 0 <= f2(n) <= cg(n), then 0 <= f2(n) <= cg(n) <= f1(n). An example would be plugging n^10 for f1(n); n^5 for f2(n); n^8 for g(n). f1(n^10) grows faster than f2(n^5), so they will not be the same. f1(n)!= f2(n).

b). Let us assume that f1(n) equals to g1(n) and f2(n) equals to g2(n), then we can state that there would be constants for the g(n). f1(n) <= c_1g(n) and f2(n) <= c_2g(n) where both c_1 and c_2 are greater than 0. To prove this, we need to find the third constant, c_3, for the claim of f1(n) + f2(n) <= c_3[g_1(n) + g_2(n)]. We will locate the biggest constant between c_1 and c_2, and we will let that constant be c_3. Therefore, c_3 will get the maximum value of c_1 and c_2, and this can satisfy the claim of this proof.

4.b).

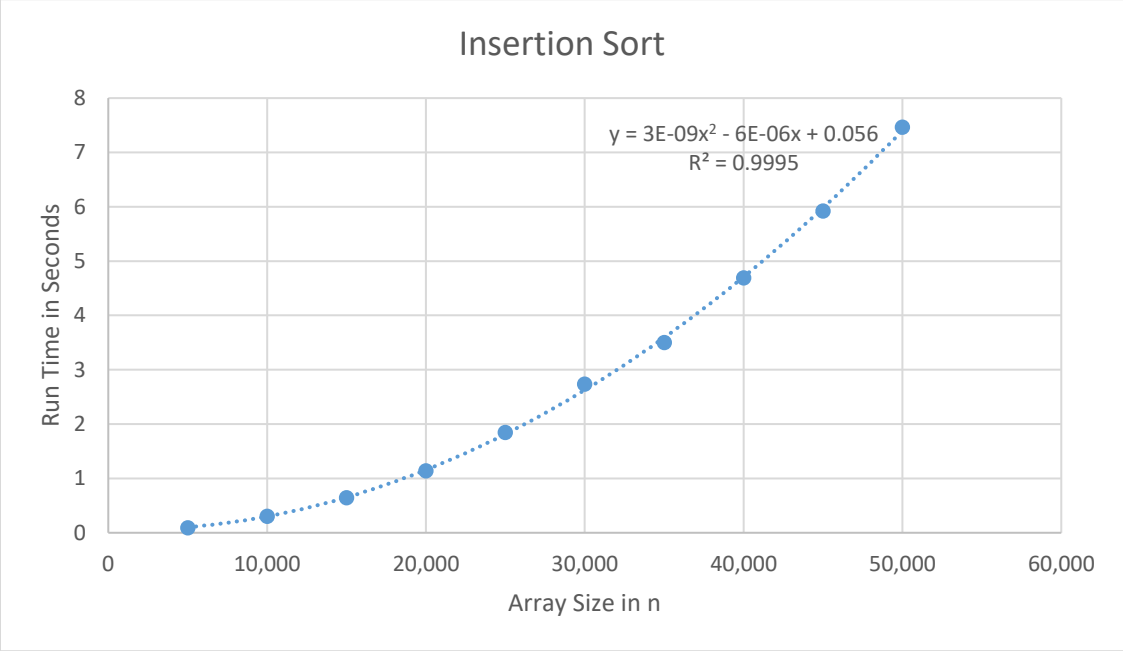
Insertion Sort

Array Size (n)	Run time 1 (Sec)	Run time 2 (Sec)	Run time 3 (Sec)
5,000	0.12	0.08	0.08
10,000	0.33	0.29	0.29
15,000	0.64	0.65	0.65
20,000	1.13	1.15	1.14
25,000	1.81	1.83	1.9
30,000	2.6	2.61	3
35,000	3.48	3.53	3.49
40,000	4.89	4.66	4.52
45,000	5.83	5.81	6.13
50,000	7.38	7.74	7.27

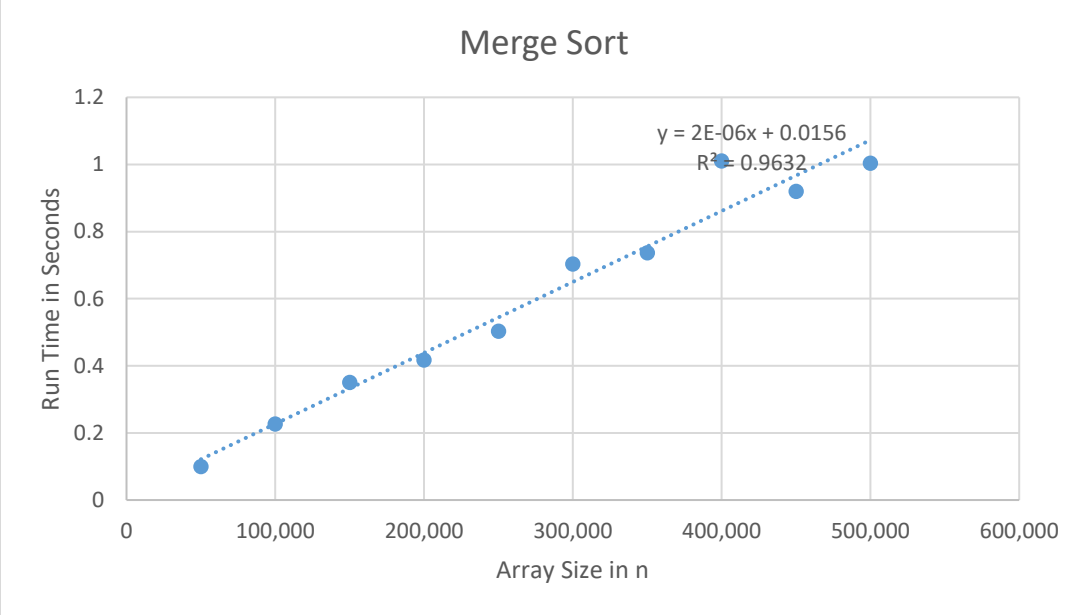
Merge Sort

Array Size (n)	Run time 1 (Sec)	Run time 2 (Sec)	Run time 3 (Sec)
50,000	0.1	0.1	0.1
100,000	0.21	0.21	0.26
150,000	0.32	0.35	0.38
200,000	0.42	0.42	0.41
250,000	0.51	0.5	0.5
300,000	0.84	0.64	0.63
350,000	0.73	0.74	0.74
400,000	0.83	1.24	0.96
450,000	0.92	0.93	0.91
500,000	1.01	1	1

4.c).

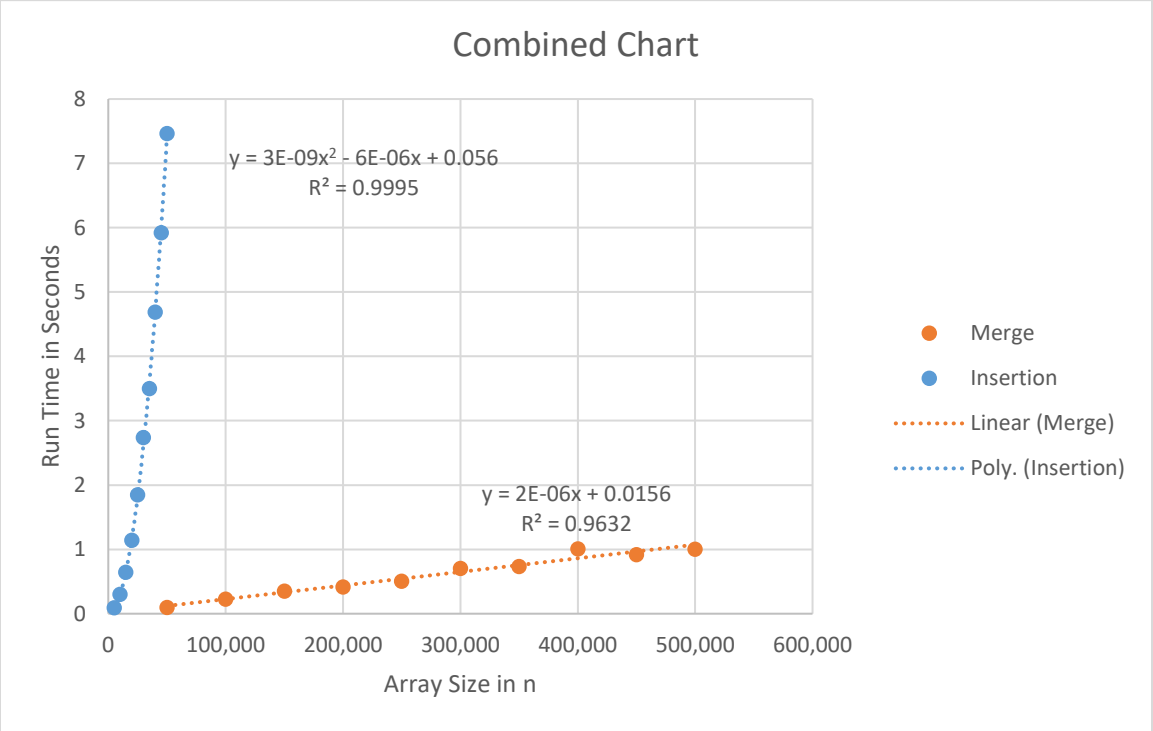


The type of curve best fits to this data set would be a quadratic curve because the graph corresponds with its complexity of $O(n^2)$.



The type of curve best fits to this data set would be a linear curve despite it's a O(n log n) time complexity, because in a large data set, the n will overcome the log n.

4.d).



4.e).

For the insertion sort, the theoretical running time should be $O(n^2)$ in both average and worst cases. The experimental running time for my insertion sort appears to be the complexity of $O(n^2)$ according to the quadratic curve on the graph. My conclusion for the insertion sort is both theoretically and experimentally identical in this case.
In the case of merge sort, the theoretical running time should be O(n log n) in both average and worst cases. The experimental running time, however, appears to be in the complexity of O(n) according to the linear curve on the graph. An explanation would be when the merge sort is sorting a very large array of values, the n in the complexity of O(n log n) will overpower the log n, which will leave a linear complexity result. My conclusion is that theoretical running time is O(n log n) but experimental running time with very large sample size will turn to linear complexity.

Below are the “text” of modified timing codes

Insertion Sort Timing

```
//This is the main function where data will be read from data.txt and sort them
int main(){

    srand (time(NULL));
    clock_t t;

    // int ten_arrays
    int n=0;
    int random_num;
    for(int i=0;i<10;i++){

        n = n + 5000; //size of the array
        vector <int> array;
       array.reserve(n);//making the space for the vector

        for(int j=0;j<n;j++){
            random_num = rand()%100001; //randomly generating the numbers
            array.push_back(random_num);
        }

        //Reference from:
        //http://www.cplusplus.com/reference/ctime/clock/
        t = clock();
        insertion(array,n);
        t = clock() - t;
        cout<<"Array Size: "<<n<<endl;
        cout<<"Time: "<<((float)t)/CLOCKS_PER_SEC << " seconds"<<endl;
        cout<<endl;
    }

    return 0;

}
```

Merge Sort Timing

```
//Merge sort reference:
//https://www.geeksforgeeks.org/merge-sort/
int main(){

    srand (time(NULL));
    clock_t t;

    // int ten_arrays
    int n=0;
    int random_num;
    for(int i=0;i<10;i++){

        n = n + 50000; //size of the array
        vector <int> array;
       array.reserve(n);//making the space for the vector

        for(int j=0;j<n;j++){
            random_num = rand()%100001; //randomly generating the numbers
            array.push_back(random_num);
        }

        //Reference from:
        //http://www.cplusplus.com/reference/ctime/clock/
        t = clock();
        mergeSort(array,0,n-1);
        t = clock() - t;
        cout<<"Array Size: "<<n<<endl;
        cout<<"Time: "<<((float)t)/CLOCKS_PER_SEC << " seconds"<<endl;
        cout<<endl;
    }


    return 0;
}
```