Brendan Jang

CS 325

Homework 1

1. A) f(n) = Ω (g(n)) because as n approaches infinity, f(n) is growing faster than g(n).

   B) f(n) = O(g(n)) because as n approaches infinity, g(n) is growing faster than f(n).

   C) f(n) = Ω(g(n)) because as n approaches infinity, f(n) is growing faster than g(n).

   D) f(n) = O(g(n)) because as n approaches infinity, g(n) is growing faster than f(n).

   E) f(n) = Θ(g(n)) because as n approaches infinity, g(n) is growing equally.

   F) f(n) = Ω(g(n)) because as n approaches infinity, f(n) is growing faster than g(n).

2. A) If f1(n) = Θ(g(n)) and f2(n) = Θ(g(n)) then f1(n) + f2(n) = Θ(g(n))

   f(n) = Θ(g(n)) if there exist positive constants c1, c2 and n0, such that 0 <= c1g(n) <= f(n) <= c2g(n) for all n >= n0.

   f1(n) = Θ(g(n)) so there must exist c1 and c2 such that c1g(n) <= f1(n) <= c2g(n)
   f2(n) = Θ(g(n)) so there must exist c3 and c4 such that c3g(n) <= f2(n) <= c4g(n)
   f1(n) + f2(n) = c1g(n) + c3g(n) <= f1(n) + f2(n) <= c2g(n) + c4g(n)
   We can simplify the equation above into the following:
   (c1 + c3)*g(n) <= f1(n) + f2(n) <= (c2 + c4)*g(n)
   c5g(n) <= f1(n) + f2(n) <= c6g(n) which satisfies the definition of theta.
   Therefore, f1(n) + f2(n) = Θ(g(n)) hence TRUE.

   B) If f1(n) = O(g(n)) and f2(n) = O(g(n)) then f1(n) = O(f2(n))
   This statement is clearly false.
   A counter example would be:
   Suppose f1(n) = n and f2(n) = log n and g(n) = n^2.
   f1(n) <= g(n) and f2(n) <= g(n) so f1(n) = O(g(n) and f2(n) = O(g(n)) are true.
   If f1(n) is equal to O(f2(n)) then f1(n) <= cf2(n) must be true.
   However, n <= c log n will never be true for any value of c > 0.
   Therefore, f1(n) = O(f2(n)) is FALSE.

3. mergeSort.cpp and insertSort.cpp

4. Modified mergeSort2.cpp and insertSort2.cpp

A) Modified insertSort2 source code and mergeSort2 source code.

```
#include <iostream>
#include <string>
#include <ctime>
#include <cstdlib>

using namespace std;

// modified insertion sort method
void insertionSort(int list[], int size) {
	for (int x = 1; x < size; x++) {
		int current = x;
		int index = x - 1;

		// compare current value with our analyzed section
		while (index >= 0 && list[current] < list[index]) {
			int temp = list[index];
			list[index] = list[current];
			list[current] = temp;

			// update index
			current = index;
			index--;
		}
	}
}

int main() {
	for (int i = 0; i < 10; i++) {
		srand(time(0));
		int size;
		int* list;
		cout << "Enter a number between 10000 and 1000000: " << endl;
		cin >> size;

		// allocate memory for our dynamic array
		list = new int[size];

		// get elements for  dynamic array
		for (int x = 0; x < size; x++) {
			int temp = rand() % 10001;
			list[x] = temp;
		}

		// Sort values and output the time
		clock_t start;
		start = clock();
```

```
                insertionSort(list, size);
                start = clock() - start;
                cout << endl << "It took " << double(start) / CLOCKS_PER_SEC << " seconds to sort" <<
endl << endl;

                // deallocate
                delete[] list;
        }
        return 0;
}
```

```
#include <iostream>
#include <string>
#include <ctime>

using namespace std;

// helper function
void merge(int list[], int start, int mid, int end) {
        // temp list to hold values in sorted order
        int * tempList = new int[end];
        int index = 0;
        int x = start;
        int y = mid;

        // iterate through sub arrays
        // will compare the current index and pass the appropriate value to the temp list
        while (x < mid && y < end) {
                if (list[x] <= list[y]) {
                        tempList[index] = list[x];

                        // update index
                        index++;
                        x++;
                }
                else if (list[y] <= list[x]) {
                        tempList[index] = list[y];
                        index++;
                        y++;
                }
        }

        // loops to make sure the subarrays were fully traversed
        // pass the remaining values
        while (x < mid) {
                tempList[index] = list[x];
                index++;
                x++;
```

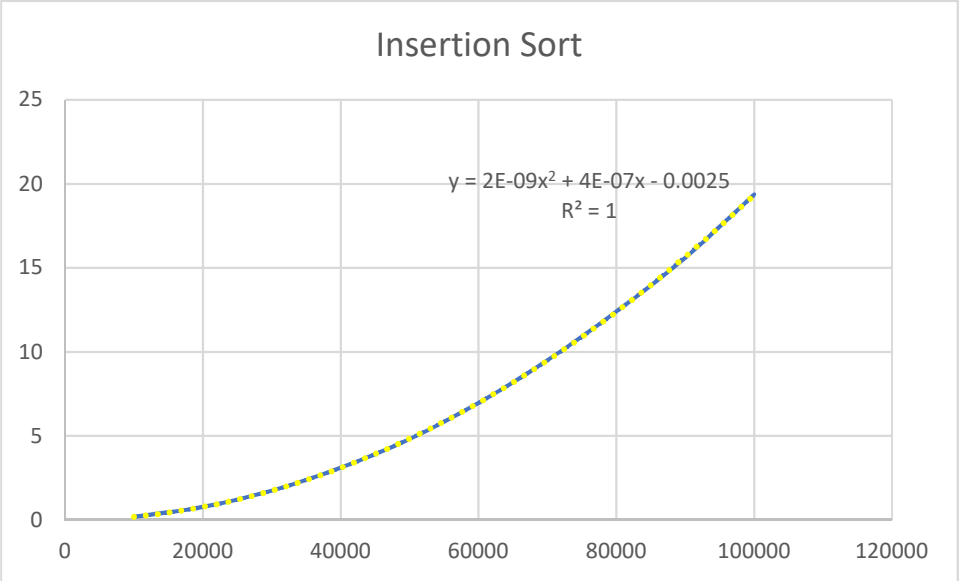
```
	}

	while (y < end) {
		tempList[index] = list[y];
		index++;
		y++;
	}

	// copy values from temp dynamic array back to input lis
	for (x = start, y = 0; y < index; x++, y++) {
		list[x] = tempList[y];
	}

	// deallocate
	delete[] tempList;
}

// merge sort function
void mergeSort(int list[], int start, int end) {
	if (start + 1 == end) {
		return;
	}

	int mid = (start + end) / 2;
	// recursive call for the first subarray
	mergeSort(list, start, mid);
	// recursive call for the second subarray
	mergeSort(list, mid, end);
	// call helper function to merge in proper order.
	merge(list, start, mid, end);
}

int main() {
	for (int i = 0; i < 10; i++) {
		srand(time(0));
		int size;
		int* list;
		cout << "Enter a number between 10000 and 1000000: " << endl;
		cin >> size;
		// allocate memory for dynamic array
		list = new int[size];
		// get elements for dynamic array
		for (int x = 0; x < size; x++) {
			int temp = rand() % 10001;
			list[x] = temp;
		}
		// sort values and output time cost
		clock_t start;
		start = clock();
```

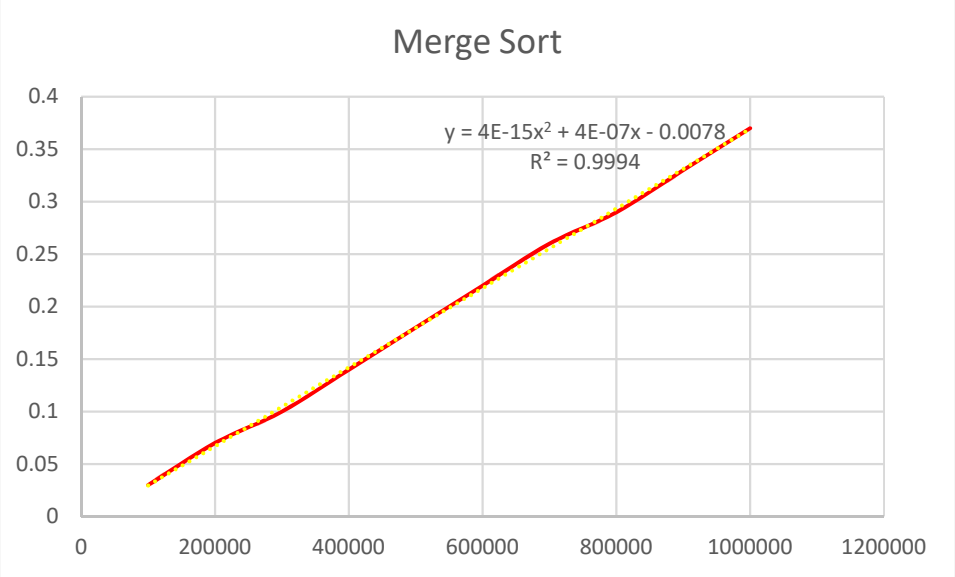
```
                mergeSort(list, 0, size);
                start = clock() - start;
                cout << endl << "It took " << double(start) / CLOCKS_PER_SEC << " seconds to sort" <<
endl << endl;

                // deallocate
                delete[] list;
        }
        return 0;
}
```

B)

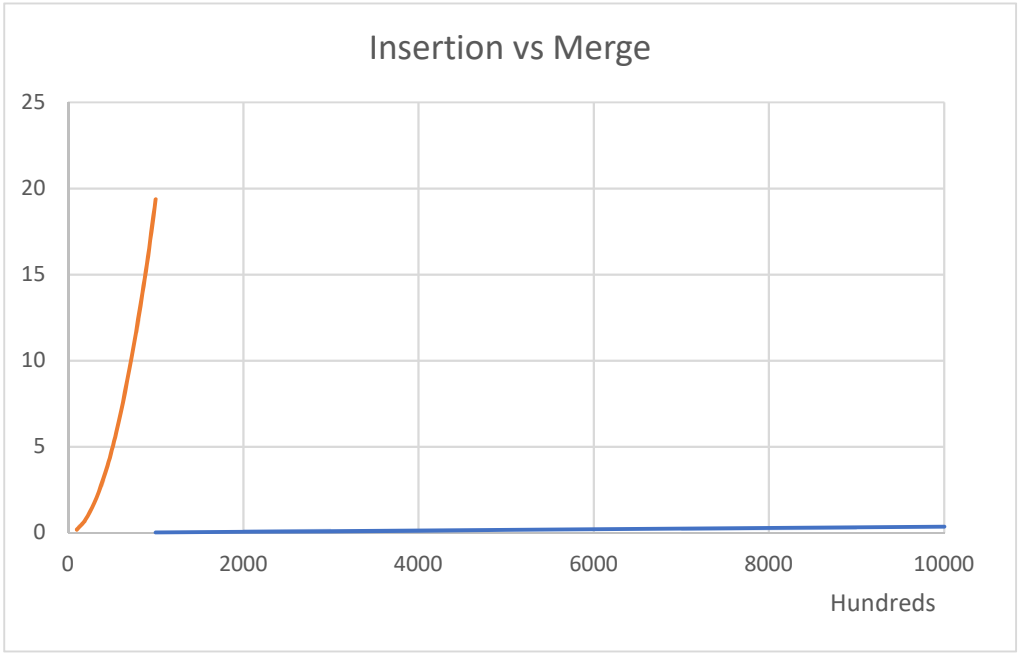
insert sort			merge sort	
n	time		n	time
10000	0.19		100000	0.03
20000	0.78		200000	0.07
30000	1.75		300000	0.1
40000	3.11		400000	0.14
50000	4.82		500000	0.18
60000	6.97		600000	0.22
70000	9.51		700000	0.26
80000	12.4		800000	0.29
90000	15.61		900000	0.33
100000	19.37		1000000	0.37

C)


Insertion Sort
y = 2E-09x² + 4E-07x - 0.0025
R² = 1
25
20
15
10
5
0
0
20000
40000
60000
80000
100000
120000


Merge Sort
y = 4E-15x² + 4E-07x - 0.0078
R² = 0.9994
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0
200000
400000
600000
800000
1000000
1200000

D)



E) Theoretical runtime of insertion sort is O(n^2) and merge sort is O(n log n). The experimental running times are on par with the theoretical run times. Merge sort runs faster and takes less time than the insertion sort.