CSCI 3412 Algorithms
Homework 1
Matt Sullivan

Part 1: Question
For Homework 1, I have selected bubble sort as my naive sort, merge sort as my normal sort, and I am using a hash table for the O(n) sort.
The sorting problem is defined as:
Input: A sequence of *n* numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.
From this definition, we can derive a sorting algorithm that tests successive permutations of the input sequence until it finds one that is sorted. Note that as long as the elements in the sequence are comparable using ≤ then a sorted permutation must exist.
I will give you several input files that will allow you to analyze various aspects of the code. All of the data are integers between 0 and 99 inclusive. The data sets are:
1. shuffled.txt – the integers 0-99 in random order – i.e., shuffled
2. sorted.txt – the integers 0-99 in sorted order (ascending order)
3. nearly-sorted.txt – the integers 0-99 in nearly sorted order
4. unsorted.txt – the integers 0-99 in unsorted order (descending order)
5. nearly-unsorted.txt – the integers 0-99 in nearly unsorted order
6. duplicate.txt – 100 integers 0, 10, 20, …, 90 – i.e., many duplicates – in random order.

All of the files have the following format with one entry per line:
Description of the data set
<n>
A[1]
A[2]
… A[n]
Note that because of the order of growth of the expected running time for the permutation sort, we will limit ourselves to using the first ten (10) elements of each data set.

Part 2: Pseudo-Code
I found the Bubblesort pseudo-code from page 40 of our textbook.

```
BUBBLESORT(A)
for I = 1 to A.length - 1
    for j = 1 to A.length-1
        if A[j] < A[j+1]
            exchange A[j] with A[j+1]
```

Likewise, I found Pseudo-code for merge-sort is from in our lecture notes. I'musing the merge function, because it's actually the recursively called part of the code

```
MERGE(A, p, q, r)
n1 = q-p+1
n2 = r-q
let L[1...n1 +1] and R[1...n2+1] be new arrays
for I =1 to n1
    L[i] = A[p+i-1]
for j =1 to n2
```

```
7        R[i] = A[q+j]
8 L[n1+1] =∞
9 R[n2 +1]=∞
10 i =1
11 j=1
12 for k = p to r
13       if L[i] <= R[j]
14             A[k]=L[i]
15             i=i+1
16       else A[k]=R[j]
17             j=j+1
```

Because the count sort has special circumstances (all members are from 0-99), it is not a common algorithm, so I wrote my own pseudo-code.

```
1While i < length of the list
2        Increment the ith index of the array of zeros
3        Increment I
4        Increment computation tracker
5 While k < length of the file list
6        While  l< length of the array of zeros
7              Append an instance of k to the sorted list
8              Increment l
9              Increment computation tracker
10       Increment k
11       Reset value l
```

Part 3: Static Analysis

Bubble-sort:

At the start of each iteration of the for loop, the values of A[length-j...length] are greater than A [1...length-j] and are sorted.

This is true prior to the first iteration, because there is no value for subarray[length-j... length] is empty, and hence sorted.

With every iteration, the higher value swaps higher (bubbles) to the way to the right of the array, making it true.

At the end of the loop, the highest value is at the rightmost position, and all of the values in [length-j...length] will be sorted.

At the start of each iteration of the for loop in lines 12-17, L[i] and R[j] are the lowest values of their lists, and A is in sorted order.

This is true prior to the first iteration, as L[i] and R[j] are both single element lists, and therefore are the lowest value, and A is sorted because it is empty.

With every iteration, L[i] and R[j] are the lowest values in their respective lists because they are iterated, and whichever is smaller is added to the end of list A, making it sorted.

At the end of the loop, L[i] and R[j] will both iterate to the end of their respective lists, and A will be completely sorted.

At the start of each iteration of the while loop in lines 5-11, the values of list[1...k] is in sorted order.

This is true prior to the first iteration, because list[1...0] is an empty list, which is sorted.

With every iteration, we step through the hash table and add the value to the list. Since the values are in sorted order, The list will be in sorted order.

At the end of the loop, the list[1..k] traverses the entire hash table, and the list is now in sorted order.

Part 4: Code
I initially wrote all 3 algorithms in Python. But I couldn't find a way to optimize the compiler fast enough for the one million randoms file size (rough estimates put runtime at about 10 days), so I re-wrote bubble-sort using C++.

Bubble-sort:
My bubble-sort is written by stepping through the array and comparing i and i+1, and swapping the value if i <i+1.

```
// This program uses a bubble sort to arrange an array of integers in
// ascending order

#include<iostream>
#include<fstream>
#include<vector>
#include<cstdint>
using namespace std;

// Matthew Sullivan

void bubbleSortArray(vector<int>&);
void displayArray(vector<int>);

int main()
{
	fstream readFile;
	readFile.open("unsorted.txt");
	readFile.ignore(256, '\n');
	readFile.ignore(256, '\n');
	vector<int> values;
	int value = 0;
	while (readFile.peek() != EOF)
	{
		readFile >> value;
		values.push_back(value);
	}
	values.pop_back();

	bubbleSortArray(values);

	return 0;
}

void displayArray(vector<int> array)    // function heading
{                                                 // displays the array
for (int i = 0; i < array.size(); i++)
{
	cout << array[i] << "  ";
}
cout << endl;
}


void bubbleSortArray(vector<int> &array)
{
	uint64_t compareNum = 0;
```

```
        uint64_t swapNum = 0;
        int temp;
        for (int i = 0; i <= array.size()-1; i++) {
                for (int j = 0; j <= array.size()-1; j++) {
                        if (j == array.size() - 1)
                        {
                                break;
                        }
                        else if (array[j] > array[j + 1]) {
                                // Consider std::swap here.
                                temp = array[j + 1];
                                array[j + 1] = array[j];
                                array[j] = temp;
                                swapNum++;
                        }
                        compareNum++;
                }
        }
        cout << "Comparisons: " << compareNum << endl;
        cout << "Swaps: " << swapNum <<endl;
}
```

Merge-sort:

Merge-sort is written in python. It calls itself recursively to split left and right halves of the array. The arrays are then merged together in sorted order.

```
"""
Created on Sat Feb  3 23:11:57 2018

@author: oneey_000
"""
a=[]
b = open("one-million-randoms.txt", "r")
b.readline()
b.readline()
size =0
for line in b:
    a.append(int (line))
    size+=1
#     if (size == 100000):          used to modify size for length comparison
#         break
def mergeSort(thisList):
    numberComp = 0
    if len(thisList) >1:
        middle = int(len(thisList)/2)
        left=thisList[:middle]
        right=thisList[middle:]
        leftResult =mergeSort (left)
        rightResult =mergeSort (right)
        numberComp +=1
        numberComp += leftResult[1]+rightResult[1]
        i=0
        j=0
        k=0
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                thisList[k] = left[i]
```

```
                i+=1
                numberComp +=1
            else:
                thisList[k] = right[j]
                j+=1
                numberComp +=1
            k+=1
        while i < len(left):
            thisList[k] = left[i]
            k+=1
            i+=1
            numberComp += 1
        while j < len(right):
            thisList[k] = right[j]
            k+=1
            j+=1
            numberComp += 1
    return thisList, numberComp
totalComp =mergeSort(a)[1]
print ("Total operations: " + str(totalComp))
```

Hash Table:
My hash table sort is also written in python. It takes advantage of the fact that we know that all of the numbers are between 0-99 to create a lists of zeros (representing integers 0-99) that are incremented as we step through the array.

```
"""
Created on Wed Feb  7 21:52:07 2018

@author: oneey_000
"""

index=[0]*100
#while i <=100:
#    index[i]=0
#    i=i+1
a=[]
b = open("shuffled.txt", "r")
b.readline()
b.readline()
for line in b:
    a.append(int (line))
i=0
j=0
k=0
l=0
sortedList=[]
numOps=0
while i < len(a):
    index[a[i]]=index[a[i]]+1
    i=i+1
    numOps=numOps+1
while j < len(index):
    print("Index: " +str(j) + " Count: " + str(index[j]))
    j=j+1
while k<len(index):
    while l< index[k]:
```

```
        sortedList.append(k)
        l=l+1
        numOps+=1
    k=k+1
    l=0
#print (sortedList)
print ("Operations: " +str(numOps))
```

Part 5: Analysis

Analysis of Growth:

As you can see from the data, the bubble-sort grows as expected at a rate of n^2. The comparisons are not full n^2, because the last index breaks the loop instead of making a comparison.

Merge-sort: As you can see, the merge-sort grows at roughly 4n(log(n)), which simplifies to n(log(n)).

Hash/Count Sort: As you can see, the Hash/Count sort grows at a rate of 2n. This is what we expected.

Comparison Across Data Sets:

Bubble-sort: As we can see, Bubble-sort is much more efficient in terms of swaps when the data is more sorted. The fewest number of swaps is a completely sorted array, followed by nearly sorted, duplicates, shuffled, nearly unsorted, then unsorted. Of course, the number of swaps is already dwarfed by the number of comparisons.

Merge-sort: As we can see, all of the different data sets have the same number of operations, which is what we expected for merge-sort, which is bound by Θ n(log(n)).

Hash/Count Sort: All of the data sets grow at a flat rate of 2n.

BubbleSort	Comparisons	Swaps	Total	
	1	0	0	0
	10	90	17	107
	100	9900	2319	12219
	1000	999000	245606	1244606
	10000	99990000	24694055	124684055
	100000	9999900000	2485926493	12485826493
	1000000	999999000000	247470673111	1247469673111

BubbleSort	Comparisons	Swaps	Total
duplicates.txt	9900	2237	12137
nearly-sorted.txt	9900	15	9915
nearly-unsorted.txt	9900	4941	14841
shuffled.txt	9900	2524	12424
sorted.txt	9900	0	9900
unsorted.txt	9900	4950	14850
one-million-randoms.txt	999999000000	247470673111	1247469673111

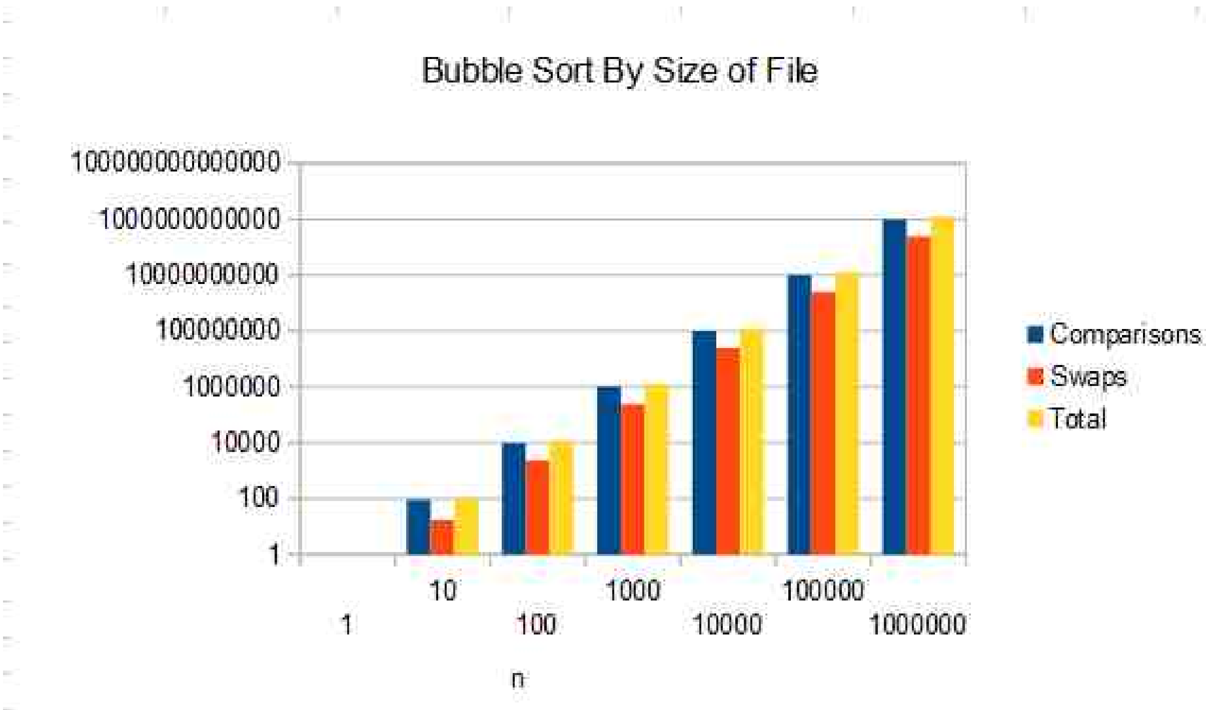
MergeSort	Operations
1	0
10	43
100	771
1000	10975
10000	143615
100000	1768927
1000000	20951423

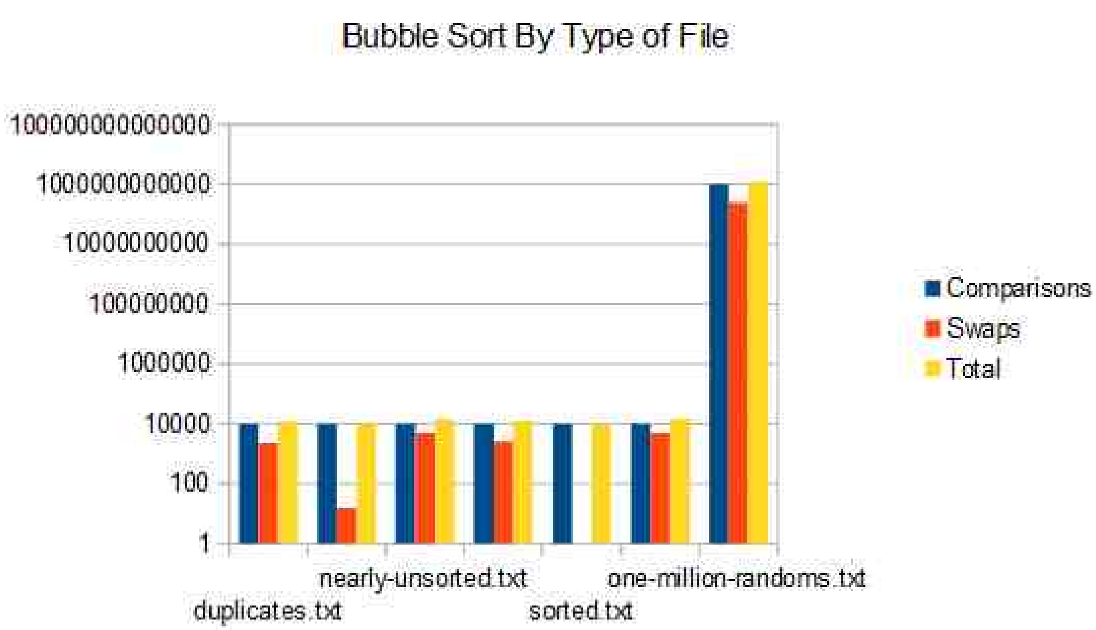
MergeSort	Operations
duplicates.txt	771
nearly-sorted.txt	771
nearly-unsorted.txt	771
shuffled.txt	771
sorted.txt	771
unsorted.txt	771
one-million-randoms.txt	20951423

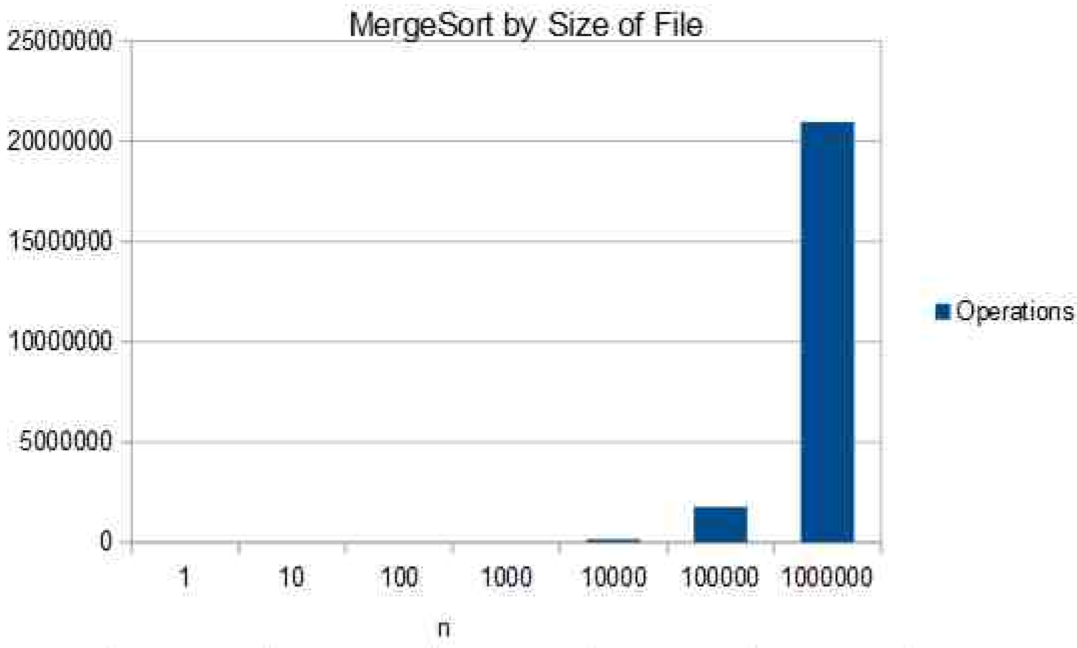
HashTable	Operations
1	0
10	20
100	200
1000	2000
10000	20000
100000	200000
1000000	2000000

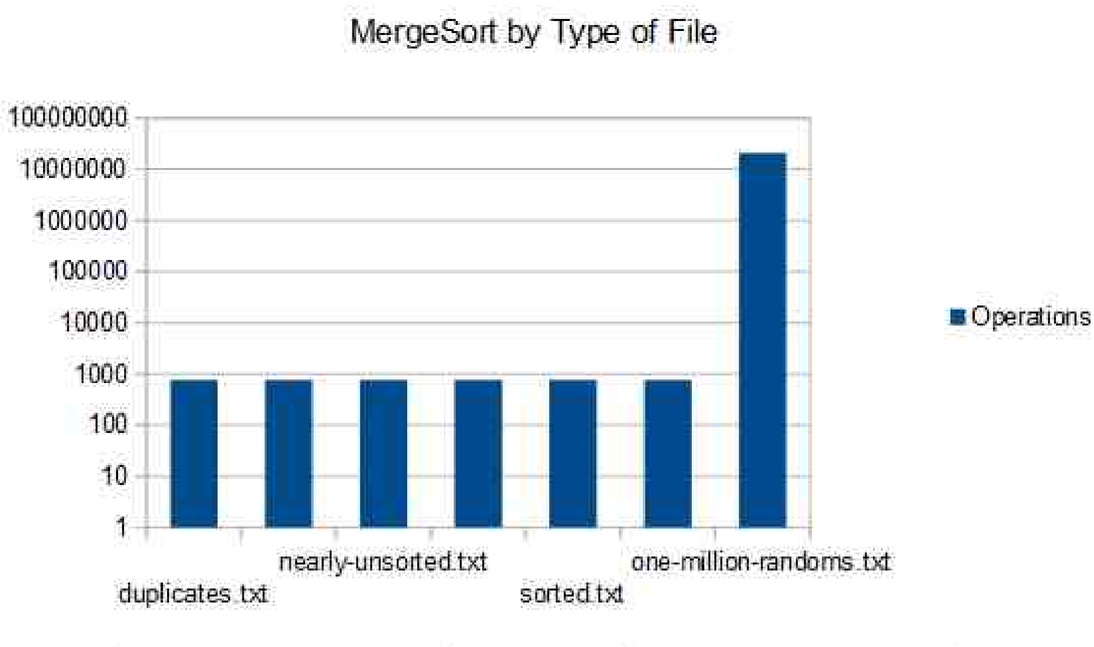
HashTable	Operations
duplicates.txt	200
nearly-sorted.txt	200
nearly-unsorted.txt	200
shuffled.txt	200
sorted.txt	200
unsorted.txt	200
one-million-randoms.txt	2000000

Graphs are on the following pages.

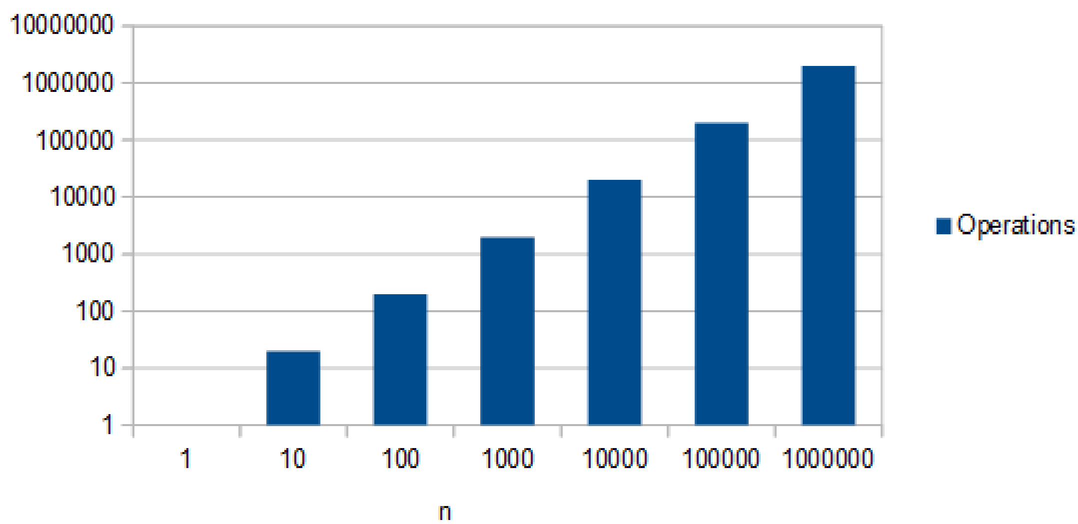

Bubble Sort By Size of File
100000000000000
1000000000000
10000000000
100000000
1000000
10000
100
1
1
10
100
1000
10000
100000
1000000
n
Comparisons
Swaps
Total


Bubble Sort By Type of File
100000000000000
1000000000000
10000000000
100000000
1000000
10000
100
1
duplicates.txt
nearly-unsorted.txt
sorted.txt
one-million-randoms.txt
Comparisons
Swaps
Total


MergeSort by Size of File
25000000
20000000
15000000
10000000
5000000
0
1
10
100
1000
10000
100000
1000000
n
Operations


MergeSort by Type of File
100000000
10000000
1000000
100000
10000
1000
100
10
1
duplicates.txt
nearly-unsorted.txt
sorted.txt
one-million-randoms.txt
Operations

Hash Table by Size of File



Hash Table by Type of File