Problem Set 1 – Sorting Algorithms

Jordan Stein

Introduction

I have implemented three different sorting algorithm's to simulate runtime for sorting many sets of integers ranging from 0 to 99. The three implemented programs are the Bubble Sort, The Merge Sort, and the Counting Sort. The first algorithm, BUBBLE-SORT, compares each array element against each other. The expected order of growth of the bubble sort is O(N^2). The second algorithm, the MERGE-SORT, splits the array elements in halves, and rebuilds the array by comparing elements. The expected order of growth for the merge sort is O(NlogN). The third algorithm, COUNTING-SORT, increments each element of the array by summing them and reassigning the indices of the elements in a new array. The Counting Sort is not caparisoned based. The expected order of growth for the counting sort is O(N). I have implemented each sort in Java and conducted a static and dynamic analysis of the efficiency of each algorithm by comparing the amount of comparisons and assignment work of each algorithm against each other.

Problem Definition

The sorting problem is defined as:

Input:

A sequence of different files containing integers sorted in many different ways:

- shuffled.txt :: integers between 0-99 in random order, i.e., shuffled
- sorted.txt :: integers between 0-99 already sorted in ascending order
- nearly-sorted.txt :: integers between 0-99 in a nearly sorted order
- unsorted.txt :: integers between 0-99 in a unsorted descending order
- nearly-unsorted.txt :: integers between 0-99 in a nearly unsorted order
- duplicate.txt :: 100 integers, 0…10…20…………90, with many duplicates in a random order
- one-million-randoms.txt ::1,000,000 integers between 0-99 in a random order

Output:

An array of integers from each file in a sorted order along with a count of the total number of comparison's each algorithm has made during run-time.

I have analyzed the output of each algorithm, (bubble sort vs merge sort vs counting sort), by sorting integers from every file to compare the efficiency of the algorithms against each other.

Part 1 – Select Sorting Algorithms

I implemented and compared the outputs of the three algorithms. The first one, the BUBBLE-SORT, compares every element against each other and is O(N^2). The second algorithm, MERGE SORT, splits the indices of the array in halves and rebuilds it by comparing the elements, O(NlogN). The third algorithm, COUNTING-SORT, increments each element of the array by summing the elements and reassigning the indices of the elements in a new array, O(N).

Part 2 - Pseudo-Code

Bubble-Sort

The bubble-sort algorithm is a simple algorithm that compares every element of the array against each other, and swaps the values if the lesser integer is in a higher element of the array as it increments through the array.

BUBBLE-SORT (arr[])

```
1     for i=0 to arr.length – 1
2         for (j=1 to arr.length)
3.            If (arr[j] < arr[j-1])
4.                swap arr[j and arr[j-1]
```

The bubble sort function returns the array with every integer in a sorted ascending order in O(N^2) time.

Merge-Sort

The merge-sort algorithm is more complicated. It consists of two functions, merge-sort, and merge. The merge-sort function recursively calls itself, splitting the array in halves for each recursive call. The merge function merges the halves together as it rebuilds the array in order.

MERGE-SORT(A, p, r)

```
1. If(p < r)
2.     q = floor((P+R)/2)
3.     MERGE-SORT(A,p,q)
4.     MERGE-SORT(A,q+1,r)
5.     MERGE(A,p,q,r)
```

MERGE(A[],p,q,r)

```
1. i = k = p
2. j=q+1
3. create tempArr from A
4. while ( i <= q && j <=r )
5.     if (tempArr[i] <= tempArr[j]) arr[k] = tempArr[i], i++
6.     else arr[k] = tempArr[j] j++
7.     k++
8. while ( i <= q )
9.     arr[k] = tempArr[i] k++ i++
```

The merge sort function returns the sorted array of integers in O(NlogN) time.

Counting-Sort

The counting sort is a simpler algorithm that sums the array elements as it increments through the array, and reassigns the array elements in order by their indices in a new array.

COUNTING-SORT(A[], min, max)

```
1.  int[] count = new int[max-min +1]
2.  int range = max – min + 1
3.  for ( i to A.length )
4.      count[ i  - min] = count[ i - min] + 1
5.  for ( i to range )
6.      count[ i ] = count[ i - 1]
7.  int j
8.  for ( i  to range )
9.      while ( j < count[i])
10.         A[z] = i
11.         j++
12.         count[ i – min ] = count[ i – min ] – 1
```

The counting sort function returns the sorted array of integers in O(N) time.

<u>Part 3 – Static Analysis</u>

We expect a different amount of total comparisons between each sorting algorithm and the different file inputs. Merge Sort should provide the least comparisons and best order of growth at O(NlogN). Counting Sort should follow after with O(N), without making any comparisons. Finally, Bubble Sort should provide the most comparisons and the worst order of growth at O(N^2)

BUBBLE-SORT

The loop invariant for the loops of the bubble sort is:

> At iteration i, A[i – 1] is sorted and every element in A[i+1….A.length] is greater or equal to every element in A[1…i]

<u>Initialization:</u>

The subarray A[j+1…A.length] is empty and has no values that exceed A[j]

<u>Maintenance:</u>

Assume the loop invariant is true. We need to show that the loop invariant is true for the following iteration. The loop will terminate when all elements have been compared against each other.

There exists two cases per iteration, either: A[j] < A[j-1] *or* A[j] >= A[j-1].

If A[j] < A[j-1], we will swap A[j] and A[j-1] so that A[j] >= A[j-1].

Else, the case A[j] >= A[j-1] is already in order and the loop invariant remains true for the next iteration.

Termination:

When the loop terminates, j = i and all of the elements in the array are in order. The best and worst case of bubble-sort is $O(N^2)$ because every element will always be compared against each other, regardless if the data set is already in order.

MERGE-SORT

The Merge Sort uses recursion to implement a divide-and-conquer approach at sorting the array.

Let's refer to the MERGE-SORT algorithm:

```
MERGE-SORT( A, p, r)

1.  If(p < r)
2.        q = floor((P+R)/2)
3.        MERGE-SORT(A,p,q)
4.        MERGE-SORT(A,q+1,r)
5.        MERGE(A,p,q,r)
```

Initialization:

MERGE-SORT is called with A being the unsorted array, p being 1, and r being the array length-1. Each recursive call of the array will subdivide it in half, and merge will combine the divided portions together in order.

Maintenance:

The Merge-Sort function divides the array in half with each recursive call, conquers each side per call, and merges the sides together in order.

DIVIDE: On line 2, we divide the size in half by computing the average of p and r.

CONQUER: On lines 3 and 4, we recursively call the function, which will divide the left and right sides of the array in half.

On line 5, we merge the array back together in a sorted fashion.

Termination:

The recursive calls terminate when p < r and the array has been sorted from A[p…r].

COUNTING-SORT

The Counting Sort is a non-comparison based sort. Comparisons are only required to find the minimum and maximum values in the array, which is solves in O(N) time. Once we know the minimum and the maximum values in the array, we can apply the Counting Sort Algorithm. The Counting Sort sorts the array by adding up every element as it increments through the array, and creates a new array by reassigning the indices with the counted sums.

Let's refer to the pseudo-code for Counting Sort:

COUNTING-SORT(A[], min, max)

```
1.  int[] count = new int[max-min +1]
2.  int range = max – min + 1
3.  for ( i to A.length )
4.      count[ i  - min] = count[ i - min] + 1
5.  for ( i to range )
6.      count[ i ] = count[ i - 1]
7.  int j
8.  for ( i  to range )
9.       while ( j < count[i])
10.           A[z] = i
11.           j++
12.           count[ i – min ] = count[ i – min ] – 1
```

The loop invariant's are as follows:

Loop on lines 3-4: after each iteration, each element of the counting array is summed from the previous element.

Loop on lines 5-6: after each iteration, the elements on count are reassigned back a value to account for the range of the sort.

Loop on lines 8-9: after each iteration, the next element in the initial array has been sorted.

Initialization:

The subarray A[0…A.length] is unsorted and the minimum and maximum values in the array are known.

Maintenance:

Assume the loop invariants are true. That is, the loops counts up the array elements and reassign them in order in the initial array.

The algorithm terminates when all elements have been sorted in the array.

In the first loop, we will sum every element with the previous one as we iterate through the loop. When the loop is complete, every element will have an incremented total up to that point. In the second loop, we are shifting all of the elements back a slot. In the third loop, we are counting up the total amount of each summed number and assigning it back into the initial array.

Termination:

The algorithm terminates after all elements have been summed up and reassigned into the initial array in sorted order.

Part 4 – Implementation

I have implemented all three algorithms in Java. You are able to choose which file of random numbers to use and which sorting algorithm to run. The output shows the total number of comparisons and assignments made for the chosen algorithm.

SortingAlgorithms.java

```
//Jordan Stein 101390302

// CSCI 3412 - FALL 2015

// Algorithms - Dr. Williams

// Problem Set 1 - Sorting Algorithms

import java.util.*;

import java.io.*;

public class SortingAlgorithms

{

        static long comparison = 0; // will keep a count of how many comparisons are done per sort.

        static long assignment = 0; // will keep a count of how many assignments are done per sort.


        public static void main(String[] args)

        {

                Scanner sc2 = new Scanner(System.in); // scanner for user input

                int fchoice = 0; // will represent file choice

                String file = "";

                do
                {
                        System.out.println("Which file would you like to use?");
                        System.out.println("1. shuffled.txt – the integers 0-99 in random order – i.e., shuffled\n"
                                        + "2. sorted.txt – the integers 0-99 in sorted order (ascending order)"
                                        + "\n3. nearly-sorted.txt – the integers 0-99 in nearly sorted order"
                        +       "\n4. unsorted.txt – the integers 0-99 in unsorted order (descending order);
                                +       "\n5. nearly-unsorted.txt – the integers 0-99 in nearly unsorted order"
        +       "\n6. duplicate.txt – 100 integers 0, 10, 20, …, 90 – i.e., many duplicates – in random order"
                        +       "\n7. one-million-randoms – 1,000,000 integers 0-99 in random order");

                        fchoice = sc2.nextInt();
                } while (fchoice > 7 || fchoice < 1); // error checks if user choses an invalid number.

                int arrlength = 0;
                if (fchoice < 7) // used if user choses any file that is not one-million-randoms
                        arrlength = 100;
                else // used if user choses one-million-randoms for the file
                {
                        System.out.print("Please input the amount of integers from one-million-randoms you
                would like to sort: ");
                        arrlength = sc2.nextInt();
                }
                switch(fchoice) // assigns file based on user choice
                {
                        case 1: file = "shuffled.txt";
                           break;
                        case 2: file = "sorted.txt";
                                break;
                        case 3: file = "nearly-sorted.txt";
                                break;
                case 4: file = "unsorted.txt";
                        break;
                case 5: file = "nearly-unsorted.txt";
                        break;
                case 6: file = "duplicate.txt";
                        break;
                case 7: file = "one-million-randoms.txt";
                        break;
        }

        int[] arr = new int[arrlength]; // array we will store integer data in

        readData(arr, fchoice, arrlength, file, sc2); // reads data from fileinto array.

        System.out.println("Which sort would you like to use?");
        System.out.println("1. Bubble Sort\n2. Merge Sort\n3. Counting Sort");

        int choice = sc2.nextInt();

        while (choice < 1 || choice > 3) // error checks if user doesnt input 1, 2, or 3
        {
                System.out.println("Which sort would you like to use?");
                System.out.println("1. Bubble Sort\n2. Merge Sort\n3. Counting Sort");
                choice = sc2.nextInt();
        }

        sc2.close(); // close file scanner

        System.out.println("Before sorting:");
```

```
for (int i=0; i < arr.length; i++)          System.out.print(arr[i] + " ");


switch(choice) // chooses which algorithm to call based on user input

{

        case 1: bubblesort(arr);

                        break;

        case 2: mergesort(arr,0,arr.length-1);

                        break;

        case 3: int max, min;

                        max = min = arr[0]; // initialize max and min to first element

                        assignment +=2;


                        for (int i = 1; i < arr.length; ++i) // finds min and max values of array.

                        {

                                if (arr[i] > max) // finds max value

                                {

                                        max = arr[i];

                                }

                                else if (arr[i] < min) // finds min value

                                {

                                        min = arr[i];

                                }

                                comparison++; // comparison for every iteration

                                assignment++; // assignment for every iteration

                        }

                        countingsort(arr, min, max);

                        break;

}
```

```
        System.out.println( "\nAfter sorting: ");
        for (int i=0; i < arr.length; i++)      System.out.print(arr[i] + " ");


        System.out.println("\nNumber of comparisons: " + comparison);
        System.out.println("Number of assignments: " + assignment);
} // end main


public static int[] bubblesort(int arr[])
{
        int temp = 0; // will hold temporary array value for swapping
        assignment++;
        for (int i=0; i < arr.length-1; ++i)
        {
                for (int j = 1; j < arr.length; ++j)
                {
                        if (arr[j] < arr[j-1])
                        {
                                temp = arr[j];                  // temporarly stores arr[j] value
                                arr[j] = arr[j-1];  // swaps arr[j] and arr[j]'s value
                                arr[j-1] = temp;    // restores arr[j-1]'s value with arr[j]s initial value
                                assignment += 3; // 3 assignments for this sort
                        }
                        comparison++; // comparison done every iteration of this loop
                }
        }
        return arr;
}
```

```
public static void mergesort(int[] arr, int p, int r) // p-> low, q-> mid, r-> high
{
  if (p < r)
  {
      comparison++;
    int q = p + ((r - p)/2); // calculates middle of array
    assignment++;
    mergesort(arr, p, q); // sorts left side of array
    mergesort(arr, q + 1, r); // sorts right side of array
    merge(arr, p, q, r); // merges left and right side together
  }
}

public static void merge(int arr[], int p, int q, int r)
{
  int tempArr[] = arr.clone(); // copies arr into tempArr
  int i, j, k;
  i = k = p; // initializes i and k to low
  j = q+1;  // initializes j to mid+1
  assignment +=3;

  while (i <= q && j <= r)  // merges right side
  {
    if (tempArr[i] <= tempArr[j])
    {
      arr[k] = tempArr[i];
      i++;
    }
    else
```

```
        {
            arr[k] = tempArr[j];
            j++;
        }
        k++;
        comparison++; // tempArr comparison for every iteration of this loop
        assignment++; // assignment for every iteration of this loop
    }
    while (i <= q) // merges left side
    {
        arr[k] = tempArr[i];
        assignment++;
        k++;
        i++;
    }
}

public static int[] countingsort(int[] arr, int min, int max)
            {
                  int range = max - min + 1; // calculates the range of values

                  int[] count = new int[range]; // arr2 will be the counting array

                  for (int i = 0; i < arr.length; i++)
                  {
                        count[arr[i] - min]++;
                  }
                  for (int i = 1; i < range; i++)
                  {
```

```
                count[i] += count[i - 1];

                assignment++;

            }


            int j = 0; // index in while loop

            for (int i = 0; i < range; i++)

            {

                while (j < count[i]) // restores initial array

                {

                    arr[j] = i + min;

                    j++;

                    assignment++;

                }

            }

            return arr;

        }
```

```
        public static void readData(int[] arr, int fchoice, int arrlength, String file, Scanner sc2) // reads data from
file into array.

        {

                try{

                        String js = ""; // will hold temporary inputs from scanner for error handling in reading
into array.

                        Scanner sc = new Scanner(new File(file)); // throws file into scanner


                        if (fchoice == 1 || fchoice == 2 || fchoice == 4 || fchoice == 6) // formats input stream
for reading numbers per file

                                for (int i=0; i < 4; i++) js = sc.next();

                        else if (fchoice == 3 || fchoice == 5)
                for (int i=0; i < 5; i++) js = sc.next();
        else if (fchoice == 7)
                for (int i=0; i < 6; i++) js = sc.next();
        if (fchoice < 7) // used if user picks file that is not one-million-randoms
        {       int i, j;
                i = j = 0;
                while (sc.hasNext())
                {
                        arr[j] = sc.nextInt();
                        if (j < arrlength)
                                j++;
                        i++;
                }
        }
        else // used if user picks file that is one-million-randoms
        {
                for (int i=0; i < arrlength; i++)
                {
                        arr[i] = sc.nextInt();
                }
        }
    }
    catch(FileNotFoundException e)
    {
        System.out.println("File not found");
    }
}
```

Part 5 - Analysis

Analysis of Growth

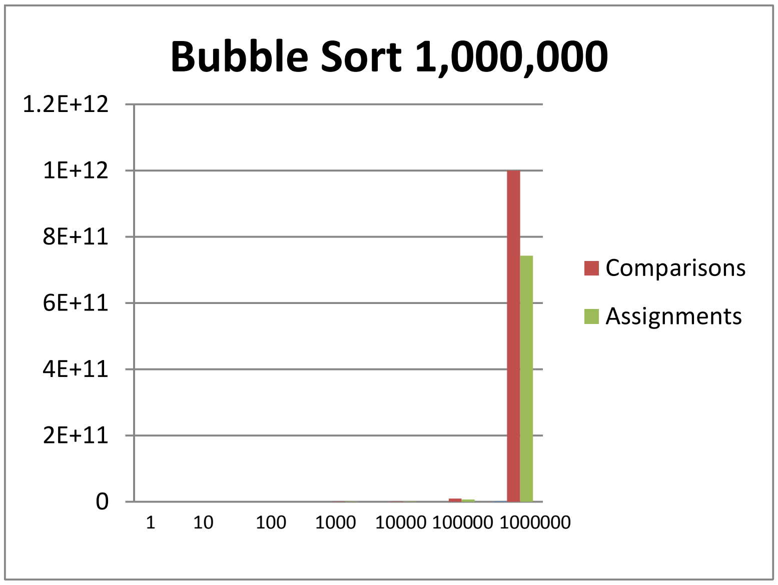
Using the one-million-randoms.txt file, I have collected the total amount of comparisons and assignments for all three sorts for n = 1,10,100,1000,10000,100000,1000000. We expect the bubble sort to output according to O(N^2), the merge sort to output according to O(NlogN), and the comparison sort to output according to O(N). I have also included the run-times for sorting 1,000,000 numbers using the Eclipse IDE. I ran this program with an Intel i7 4790k @ 4ghz and 16GB ddr3-2311 ram.

The following pages include graphed and charted data for sorting up to 1,000,000 integers with the bubble sort, merge sort, and counting sort algorithms.

Bubble Sort

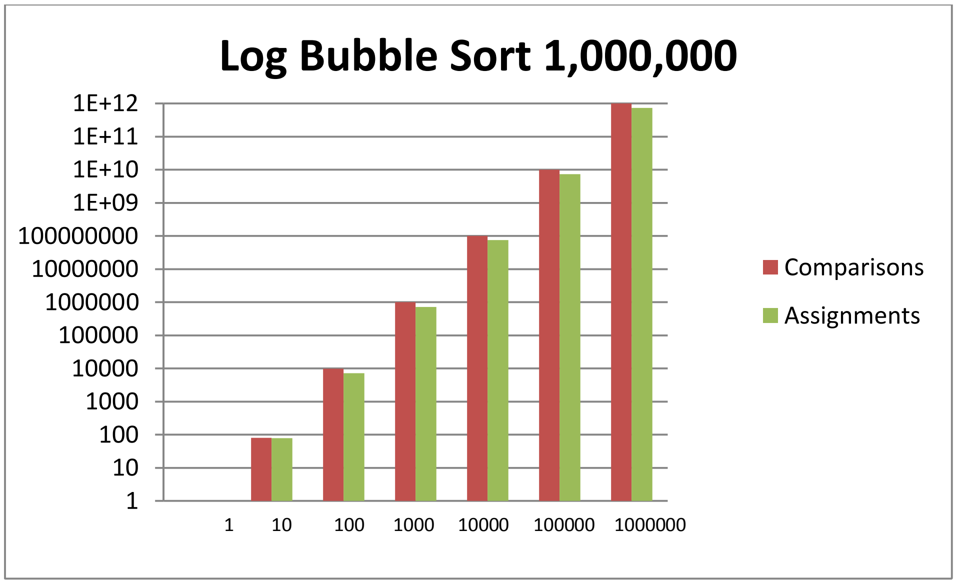
Here are the results of running the bubble sort algorithm for up to n = 1,000,000.

N	Comparisons	Assignments
1	0	1
10	81	79
100	9801	7213
1000	998001	727186
10000	99980001	75192154
100000	1E+10	7.41E+09
1000000	1E+12	7.43E+11



Note that the difference in comparisons and assignments is massive between 100,000 and 1,000,000.

Below I have provided a logarithmic scale in base 10 to see a more readable version of the graph.

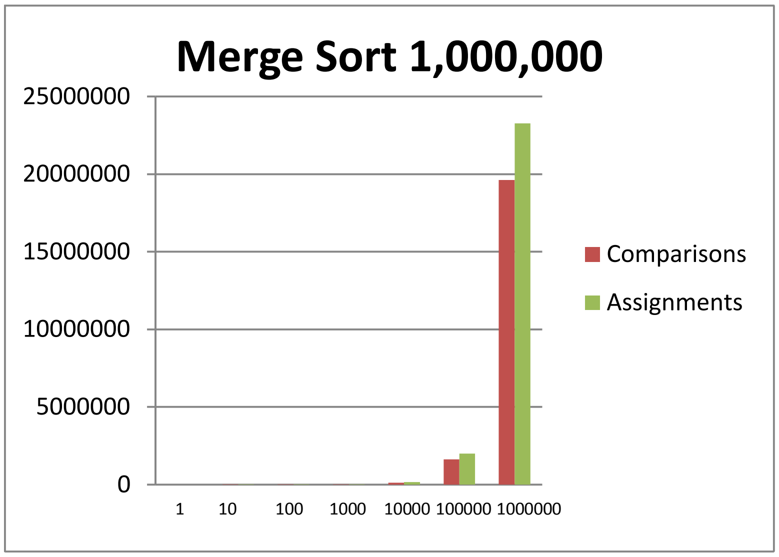


As expected, the comparisons and assignments for the bubble sort is outputting in correlation to N^2. I have at least one assignment done in the first case to initialize the temporary value that will be used to place-hold when swapping values. The program took 21 minutes to sort 1,000,000 integers on my computer.

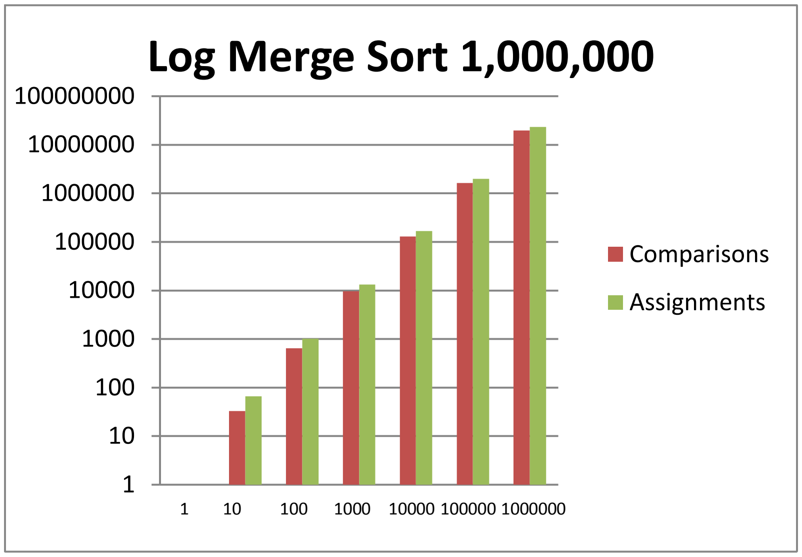
Merge Sort

Here are the results of running the merge sort algorithm for up to n = 1,000,000.

N	Comparisons	Assignments
1	0	0
10	33	66
100	644	1018
1000	9705	13316
10000	130301	167711
100000	1632627	2006933
1000000	19619827	23268089



Again, the comparison/assignment difference is massive so a base 10 log graph is below:



The output of the merge sort grows in correlation to the expected order of NlogN. The runtime was much faster using this sort, taking only 6 minutes to sort 1,000,000 integers; outperforming the bubble sort's runtime by 350%.

Counting Sort

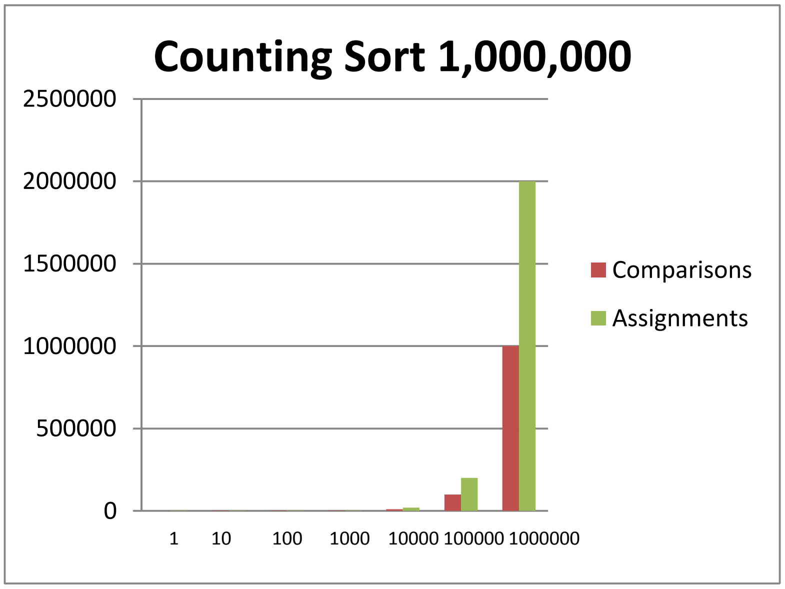
Here are the results of running the counting sort for up to n = 1,000,000.

Counting Sort O(N)

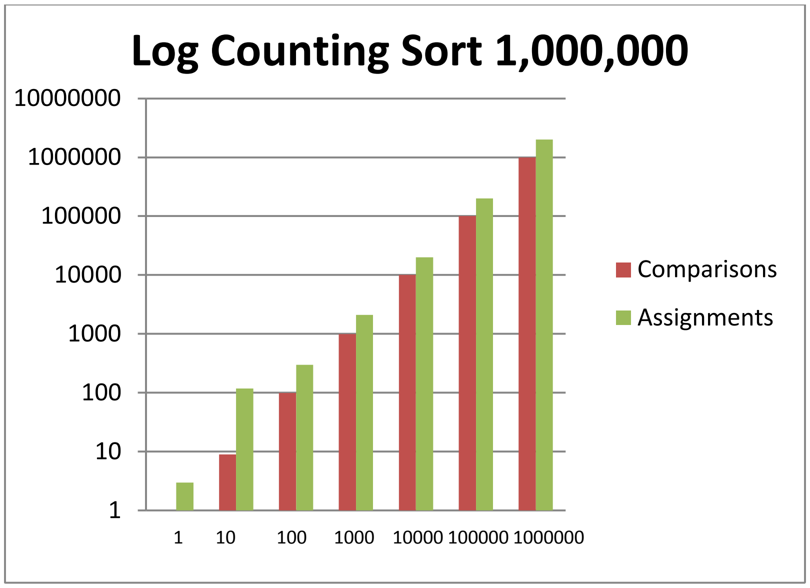
N	Comparisons	Assignments
1	0	3
10	9	119
100	99	300
1000	999	2100
10000	9999	20100
100000	99999	200100
1000000	999999	2000100

*comparisons only made to calculate

minimum and maximum values



Base 10 log graph:



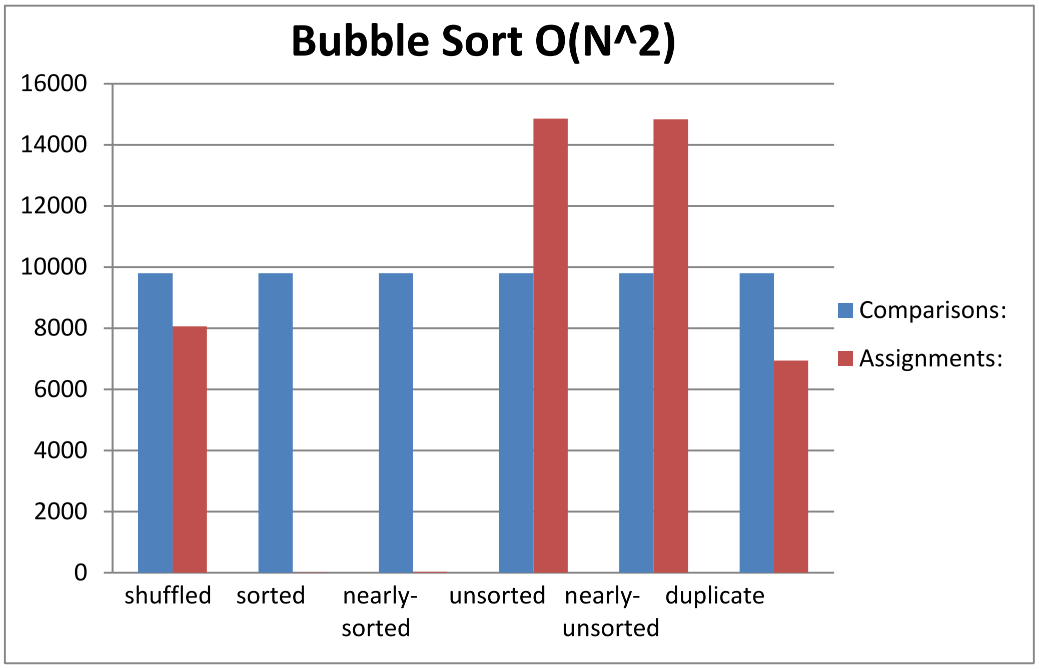
The comparisons and assignments of the counting sort algorithm also grow as expected in O(N). Although the counting sort is non-comparison based, I did keep track of the total amount of comparisons because I calculated my minimum and maximum values in a for loop rather than hard coding in the minimum of 0 and maximum of 99 into my program so it would work with any integer data set. This algorithm provided the fastest runtime for sorting 1,000,000 integers at ONLY 1 MINUTE; outperforming the bubble sort by 2100% and the merge sort by 6000%.

Comparison Across Data Sets

I have provided a spread and bar graphs of the differences between comparisons have assignments for all three sorts using shuffled.txt, sorted.txt, nearly-sorted.txt, unsorted.txt, nearly-unsorted.txt, and duplicate.txt. Note that each data file had 100 integers ranging from 0-99 inclusive.

Bubble Sort O(N^2)

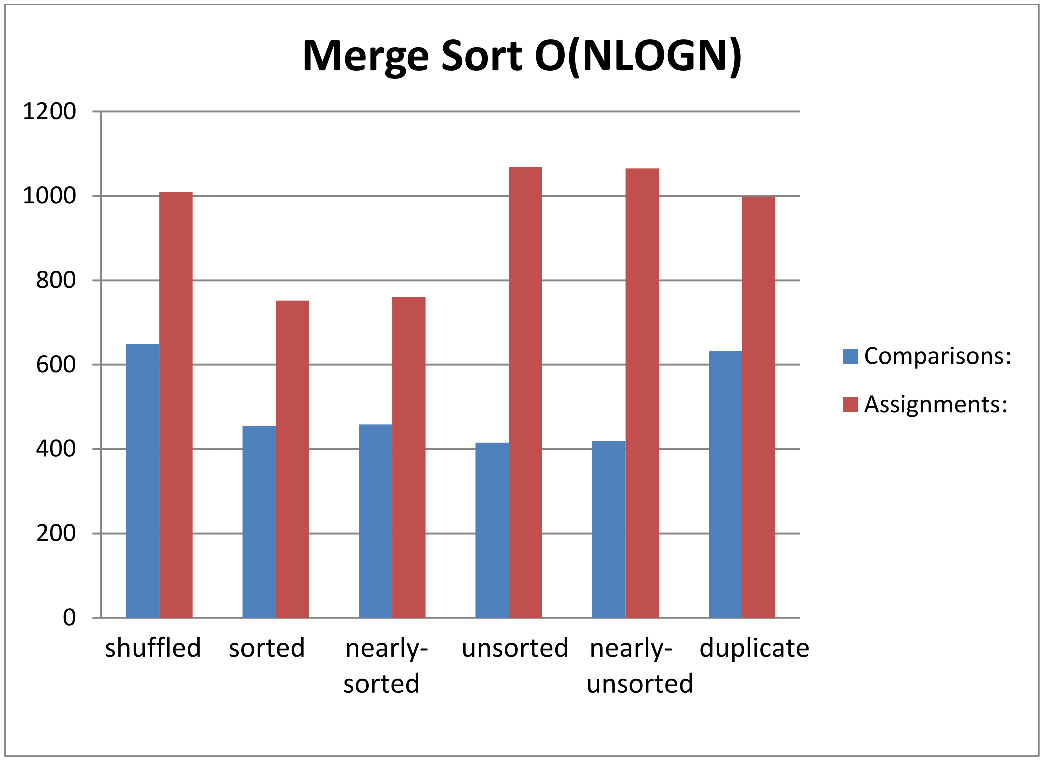
Algorithm: File	Bubble Sort O(N^2) Comparisons:	Assignments:
shuffled.txt	9801	8062
sorted.txt	9801	1
nearly-sorted.txt	9801	31
unsorted.txt	9801	14851
nearlyunsorted.txt	9801	14831
duplicate.txt	9801	6934



Notice that the amount of comparisons is constant regardless of the input size using the bubble sort algorithm. This is because every element will always be compared with each other due to the algorithm being O(N^2). However, the amount of assignments or “work" done by this algorithm is completely dependent on the data set being sorted. When the data set was shuffled or had duplicates, the amount of assignments needed was slightly less than the amount of comparisons made. When the data set was sorted, very little assignments were required to bring the list in order. The most amount of assignment work was required when the graph was unsorted or nearly-unsorted. This is because every single value had to be switched to bring the graph into ascending order.

Merge Sort O(NlogN)

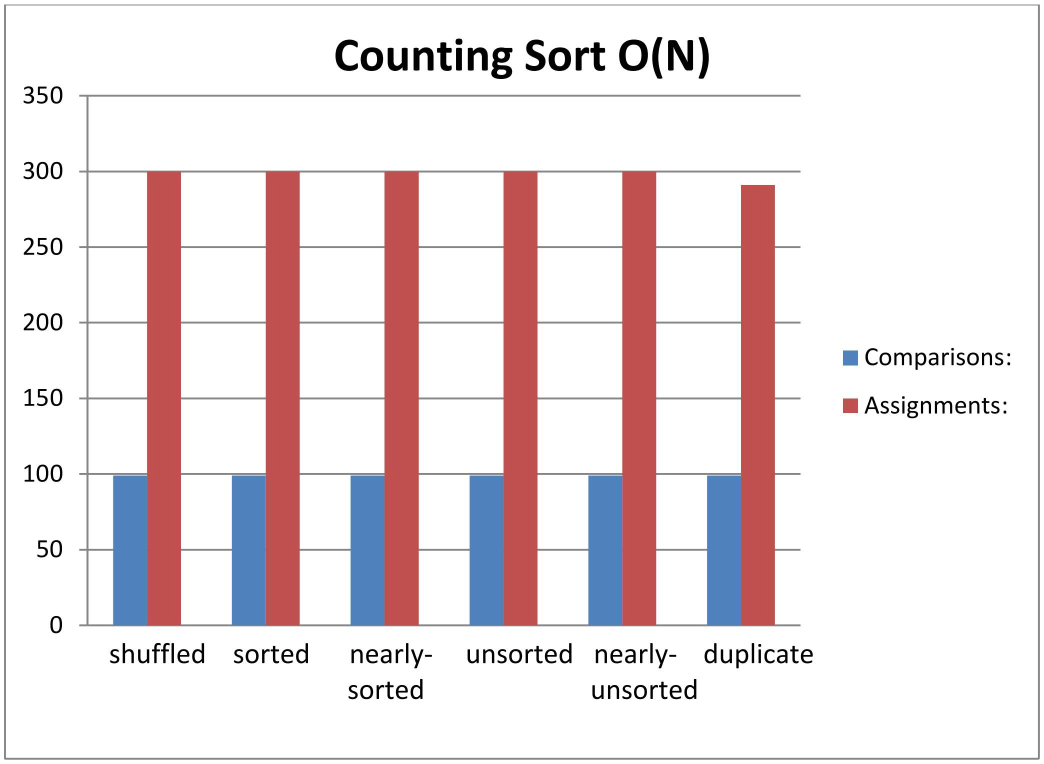
Algorithm:	Merge Sort O(NlogN)	
File	Comparisons:	Assignments:
shuffled.txt	649	1010
sorted.txt	455	752
nearly-sorted.txt	458	761
unsorted.txt	415	1068
nearlyunsorted.txt	419	1065
duplicate.txt	633	998



What is interesting about the merge sort is that the amount of comparisons only slightly varies between 400 and 650 and the amount of assignments only slightly varies between 750 and 1070 regardless of the data set given. This is because the merge sort algorithm will break the data set apart many times, and rebuild it by comparing values disregarding if the data set is already sorted or nearly sorted. The output of this algorithm correlates with O(NlogN) as expected.

Counting Sort O(N)

Algorithm: File	Counting Sort O(N) Comparisons:	Assignments:
shuffled.txt	99	300
sorted.txt	99	300
nearly-sorted.txt	99	300
unsorted.txt	99	300
nearlyunsorted.txt	99	300
duplicate.txt	99	291



This was my favorite sort to study and implement. As previously mentioned, note that although the counting sort is non-comparison based, I did use comparisons to calculate the min and max values rather than hard-coding 0 and 99 so that my function would work with any program. What surprised me about this algorithm is that the amount of assignments made was constant for every data set given. (Only slightly fewer assignments with duplicate values). This concludes that the runtime is linear based on the number of inputs, O(N) as expected.

Conclusions

I specified, implemented, and analyzed the bubble sort, merge sort, and counting sort algorithms using many different integer data sets. The outputs of my program had successfully matched the expected order of growth of each algorithm for every input given.