Problem Set 1 – Sorting Algorithms

Sean G Meyer

Introduction

I implemented three different algorithms for sorting: INSERTION-SORT, MERGE-SORT, and COUNTING-SORT. All three algorithms are specified in psuedo-code and implemented in the C# programming language. Static analysis was done to show correctness and the expected order of growth, and dynamic analysis of the actual number of comparisons and assignments was done against seven different data sets in order to determine the effect of data ordering on the expected order of growth.

From these analysis, we show that the expected order of growth of the algorithms is as follows, INSERTION-SORT: $\mathcal{O}(n^2)$, MERGE-SORT: $\mathcal{O}(n \log n)$, COUNTING-SORT: $\mathcal{O}(n)$.

Problem Definition

The sorting problem is defined as:

Input: A sequence of n numbers $< a_1, a_2, \ldots, a_n >$

Output: A permutation (reordering) $< a'_1, a'_2, \ldots, a'_n >$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Using this definition, the problem is to select three sorting algorithms to analyze and implement. One must be a naive sorting algorithm with $\mathcal{O}(n^2)$ expected running time, another must have an expected run time of $\mathcal{O}(n \log n)$, and the third must have an expected running time of $\mathcal{O}(n)$ (possible by using the fact that our sample data is made up only of integers between 0 and 99 inclusive).

Discussion

I implemented three different algorithms: INSERTION-SORT, MERGE-SORT, and COUNTING-SORT. Details and analysis of these follow.

Part 1 - Psuedo-Code

INSERTION-SORT

The insertion sort algorithm is a relatively simple comparison based sorting algorithm which selects items one by one and inserts them into the correct position (within the items that have been sorted so far).

INSERTION-SORT(A) [1]

```
for j = 2 to A.length
        key = A[j]
        // Insert A[j] into the sorted sequence A[1 .. j -1]
        i = j -1
        while i > 0 and A[i] > key
                A[i + 1] = A[i]
                i = i - 1
        A[i + 1] = key
```

The INSERTION-SORT procedure sorts the given array.

MERGE-SORT

This merge sort algorithm works by dividing an array into two halves, sorting the two halves (recursively), and then merging the results so that we are left with a single ordered array.

MERGE-SORT(A, low, hi) [3]

```
if hi <= low
        return;
mid = low + (hi - low) / 2
MERGE-SORT(A, low, mid)        // Sort left half.
MERGE-SORT(A, mid+1, hi)       // Sort right half
MERGE(A, low, mid, hi)         // Merge results
```

The MERGE-SORT procedure is the main procedure used to perform the sort. It divides the array in half (recursively, down to 1-item arrays) and then merges all of those subarrays to form the final, completely sorted, array. The MERGE procedure referenced in this procedure is shown on the next page.

MERGE(A, low, mid, hi) [3]

```
i = low
j = mid + 1

let AUX[low .. hi] be a new array

for k = low to hi        // Copy A[lo .. hi] to AUX[lo .. hi]
        AUX[k] = A[k]

for k = low to hi        // Merge back to a[lo .. hi]
        if i > mid
                A[k] = AUX[j]
                j = j + 1
        else if j > hi
                A[k] = AUX[i]
                i = i + 1
        else if AUX[j] < AUX[i]
                A[k] = AUX[j]
                j = j + 1
        else
                A[k] = AUX[i]
                i = i + 1
```

The MERGE procedure implements the merge portion of MERGE-SORT by first copying everything into an auxiliary array (AUX) and then merging back to the original array (A).

COUNTING-SORT

COUNTING-SORT(A, min, max)

```
let C[min .. max] be a new array
for i = 1 to A.length
        C[A[i]] = C[A[i]] + 1

pos = 0
for i = 1 to A.length
        for j = 1 to C[i]
                 A[pos] = j
                 pos = pos + 1
```

The COUNTING-SORT procedure performs a sort without doing comparisons by utilizing a min and max value of the numbers to be sorted. It creates a new array with that many positions, and increments the value stored at the appropriate position each time a number is seen. These counts are then written back out to the original array, producing a sorted array.

Part 2 - Static Analysis

INSERTION-SORT [1]

The loop invariant for the main insertion sort loop is:

> At the start of each iteration of the for loop of lines 1-8, the subarray $A[1 \dots j-1]$ consists of the elements originally in $A[1 \dots j-1]$, but in sorted order.

Initialization: The subarray $A[1 \dots j-1]$ consists of a single element, A[1], and this is the original element in $A[1]$. Therefore the subarray $A[1 \dots j-1]$ consists of the original elements and is sorted.

Maintenance: The body of the loop works by moving $A[j-1], A[j-2], A[j-3]$, and so on by one position to the right until it finds the proper position for $A[j]$. It then inserts the value of A[j]. The subarray $A[1 \dots j]$ then consists of the elements originally in $A[1 \dots j]$, but in sorted order. After incrementing j for the next iteration, the loop invariant is preserved.

Termination: The loop terminates when j > A.length = n. At the end of the loop j must equal n + 1 (since j is incremented by 1 each iteration). Therefore at termination, the subarray $A[1 \dots n]$ consists of the elements originally in $A[1 \dots n]$, but in sorted order (by the loop invariant). Since $A[1 \dots n]$ is the entire array, we conclude that the entire array is sorted.

Order of Growth: In the worst case (if the array is in reverse sorted order), each element A[j] must be compared with the entire sorted subarray $A[1 \dots j-1]$. This results in a worst case running time which can be expressed as an^2+bn+c, thus the order of growth is $\mathcal{O}(n^2)$.

MERGE-SORT [2]

The loop invariant for the MERGE function loop is:

> At the start of each iteration of the for loop of lines 9-21, the subarray $A[low \dots k-1]$ contains the $k - low$ smallest elements of $AUX[1 \dots hi]$, in sorted order.

Initialization: Prior to the first iteration of the loop we have $k = low$, so the subarray $A[low \dots k-1]$ is empty. Since there are no items, the loop invariant holds.

Maintenance: If the left half of AUX has no items, then the next element is in the right half. That element is copied to A[k] (line 11) and the subarray $A[low \dots k]$ will contain the $k - p + 1$ smallest elements. Then the right half counter is incremented (12) and k is incremented, reestablishing the loop invariant.

If the right half has no items, the next element is in the left half. That element is copied into A[k] (14), and the subarray $A[low \ldots k]$ will contain the $k - p + 1$ smallest elements. Then the left half counter is incremented (15) and k is incremeented, reestablishing the loop invariant.

If neither half is empty, there are two cases:

If $AUX[i] \leq AUX[j]$, then AUX[i] is the smallest element not yet copied back into A. After it is copied into A[k], the subarray A[low .. k] will contain the $k - p + 1$ smallest elements. Increment k (for loop increment) and i (line 21) reestablishes the loop invariant.

Conversely, if AUX[j] $\leq$ AUX[i], then lines 17-18 perform the corresponding actions to once again maintain the loop invariant.

Termination: At termination, $k = hi + 1$. By the loop invariant, the subarray $A[low \ldots hi]$ contains the $hi - low$ smallest elements in sorted order. Since low was provided as the smallest element of the array, and hi was provided as the largest element of the array, the entire array is now sorted.

Order of Growth [3]: Let $C(N)$ be the number of compares needed to sort an array of length N. We have $C(0) = C(1) = 0$ and for $N > 0$ we can write a recurrence relationship that directly mirrors the recursive MERGE-SORT method to establish an upper bound:

$$C(N) \leq C(\lfloor N/2 \rfloor) + C(\lfloor N/2 \rfloor) + N$$

An exact solution to the recurrence can be derived when equality holds and N is a power of 2. The resulting solution is:

$$C(N) = C(2^n) = n2^n = NlgN$$

This shows that merge sort has an order of growth of $\mathcal{O}(n \log n)$

COUNTING-SORT

The loop invariant for the for loop on lines 3-4 is:

> At the start of each iteration of the loop, the elements of the subarray $A[1 \ldots i-1]$ have been counted in array $C[min \ldots max]$, where $C[A[i]]$ holds the value of the count of each number i within A.

Initialization: The subarray $A[1 \ldots i-1]$ has no elements since i is 1. Therefore the subarray has been counted.

Maintenance: Each iteration the item stored at $A[i]$ is counted in array $C[min \ldots max]$, so that $A[1 \ldots i]$ elements have been counted. i is then incremented by 1 so that $A[1 \ldots i-1]$ have been counted, which is the loop invariant.

Termination: The loop terminates when $i = A.length + 1$. By the loop invariant, this means the subarray $A[1 \ldots A.length]$ has been counted, so the entire array has been counted.

Order of Growth: The counting loop goes through array A once, giving a running time of N. Another loop is required to rewrite the array, and it also goes to each element once for N operations. This results in $2N$ operations, which is an order of growth of $\mathcal{O}(n)$.

Part 3 - Implementation

All algorithms were coded in C# . The file Sort.cs implements the algorithms as a class, with some helper functions and comparison/assignment fields added. The Program.cs file contains the main loop to sort the sample data and output the results.

Sort.cs

```
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Text;
6  using System.Threading.Tasks;
7  using System.Diagnostics.Contracts;
8
9  namespace SortingAlgorithms
10 {
11     class Sorter<T> where T : struct, IComparable     // value type and comparable
12     {
13         private long comparisons;
14         private long assignments;
15         private List<T> list;
16         private IList<T> originalList;
17
18         public Sorter(IList<T> inputList)
19         {
20             this.originalList = inputList;
21             this.list = CopyList(inputList);
22         }
23
24         public long Comparisons
25         {
26             get { return comparisons; }
27             set { comparisons = value; }
28         }
```

```
29
30          public long Assignments
31          {
32              get { return assignments; }
33              set { assignments = value; }
34          }
35
36          public bool IsSorted()
37          {
38              for (int i = 1;  i < list.Count; i++)
39                  if (list[i].CompareTo(list[i-1]) < 0)
40                      return false;
41              return true;
42          }
43
44          public List<T> InsertionSort()
45          {
46              Contract.Ensures(IsSorted());
47              comparisons = 0;
48              assignments = 0;
49              list = CopyList(originalList);
50
51              for (int j = 1; j < list.Count; j++)
52              {
53                  var key = list[j];
54                  // Insert list[i] into the sorted sequence list[1...i-1]
55                  int i = j - 1;
56
57                  while (i >= 0 && less(key, list[i]))
58                  {
59                      list[i + 1] = list[i];
60                      assignments++;
61                      i--;
62                  }
63                  list[i + 1] = key;
64                  assignments++;
65              }
66              return list;
67          }
68
69          public List<T> MergeSort()
70          {
71              Contract.Ensures(IsSorted());
72              comparisons = 0;
73              assignments = 0;
74              list = CopyList(originalList);
```

```
75
76              List<T> aux = new T[list.Count].ToList();
77              MergeSort(aux, 0, list.Count - 1);
78              return list;
79          }
80
81          private void MergeSort(List<T> aux, int lo, int hi)
82          {
83
84              if (hi <= lo) return;
85              int mid = lo + (hi - lo) / 2;
86              MergeSort(aux, lo, mid);
87              MergeSort(aux, mid + 1, hi);
88              merge(aux, lo, mid, hi);
89          }
90
91          public void merge(List<T> aux, int lo, int mid, int hi)
92          {
93              int i = lo, j = mid+1;
94
95              for (int k = lo; k <= hi; k++)
96              {
97                  aux[k] = list[k];
98              }
99
100             for (int k = lo; k <= hi; k++)
101             {
102                 if (i > mid)
103                     list[k] = aux[j++];
104                 else if (j > hi)
105                     list[k] = aux[i++];
106                 else if (less(aux[j], aux[i]))
107                     list[k] = aux[j++];
108                 else
109                     list[k] = aux[i++];
110                 assignments++;
111             }
112         }
113
114         public List<T> CountingSort()
115         {
116             Contract.Requires(typeof(T) == typeof(int));
117             Contract.Ensures(IsSorted());
118             comparisons = 0;
119             assignments = 0;
120
```

```
121             int[] countArray = new int[100];
122             foreach (T num in originalList)
123             {
124                 countArray[(int)(object) num]++;
125             }
126 
127             list = new List<T>();
128 
129             for (int i = 0; i < 100; i++)
130                 for (int j = countArray[i]; j > 0; j--)
131                 {
132                     list.Add((T)(object) i);
133                     assignments++;
134                 }
135 
136             return list;
137         }
138 
139         public void PrintSortedList()
140         {
141             foreach (T item in list)
142             {
143                 Console.WriteLine(item);
144             }
145         }
146 
147         // Used so that we can return a sorted list without modifying original
148         private List<T> CopyList(IList<T> list)
149         {
150             List<T> listCopy = new List<T>(list);
151             return listCopy;
152         }
153 
154         // return true if a is less than b
155         private bool less(T a, T b)
156         {
157             comparisons++;
158             return a.CompareTo(b) < 0;
159         }
160 
161         // exchange a with b
162         private void exchange(int a, int b)
163         {
164             T temp = list[a];
165             list[a] = list[b];
166             list[b] = temp;
```

```
            assignments += 2;   // 2 list assignments done
        }
    }
}
```

Program.cs

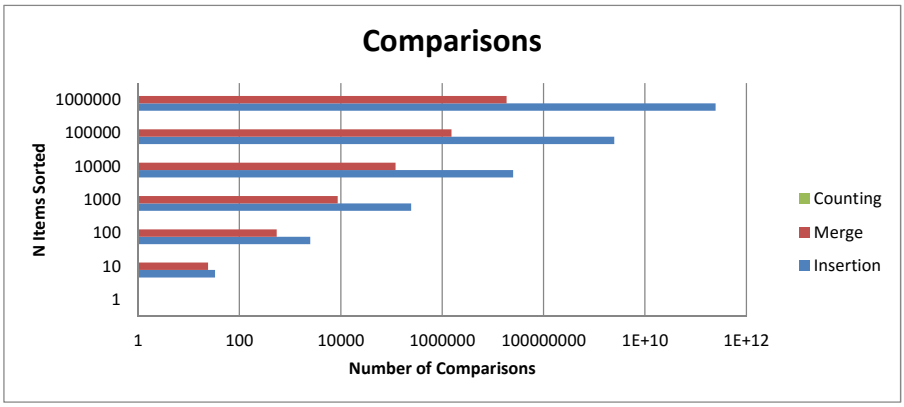
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;
using System.Diagnostics;

namespace SortingAlgorithms
{
    class Program
    {
        static void Main(string[] args)
        {
            PerformSorts("sorted.txt");
            PerformSorts("nearly-sorted.txt");
            PerformSorts("duplicate.txt");
            PerformSorts("shuffled.txt");
            PerformSorts("nearly-unsorted.txt");
            PerformSorts("unsorted.txt");
            Console.ReadKey(true);
        }

        static List<int> ReadFile(string fileName)
        {
            var numberList = new List<int>();
            string line;

            TextReader reader = File.OpenText("data/" + fileName);
            reader.ReadLine();  // skip first line
            int total = int.Parse(reader.ReadLine());   // second line gives total

            // read the file
            while ((line = reader.ReadLine()) != null)
            {
                numberList.Add(int.Parse(line));
            }

            return numberList;
        }
```

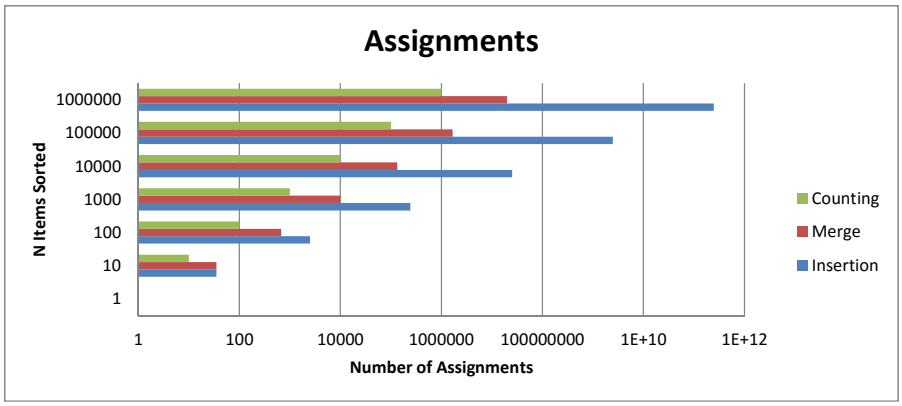
```
41
42          static void PerformSorts(string fileName)
43          {
44              var Sort = new Sorter<int>(ReadFile(fileName));
45              long sum;
46
47              Console.WriteLine("---------------- " + fileName + " ----------------");
48              List<int> insertionSort = Sort.InsertionSort();
49              Console.WriteLine("Insertion Sort");
50              Console.WriteLine("Comparisons: " + Sort.Comparisons);
51              Console.WriteLine("Assignments: " + Sort.Assignments);
52              sum = Sort.Comparisons + Sort.Assignments;
53              Console.WriteLine("Total Operations: " + sum);
54              Console.WriteLine();
55              List<int> mergeSort = Sort.MergeSort();
56              Console.WriteLine("Merge Sort");
57              Console.WriteLine("Comparisons: " + Sort.Comparisons);
58              Console.WriteLine("Assignments: " + Sort.Assignments);
59              sum = Sort.Comparisons + Sort.Assignments;
60              Console.WriteLine("Total Operations: " + sum);
61              Console.WriteLine();
62              List<int> countingSort = Sort.CountingSort();
63              Console.WriteLine("Counting Sort");
64              Console.WriteLine("Comparisons: " + Sort.Comparisons);
65              Console.WriteLine("Assignments: " + Sort.Assignments);
66              sum = Sort.Comparisons + Sort.Assignments;
67              Console.WriteLine("Total Operations: " + sum);
68              Console.WriteLine();
69              List<int> shellSort = Sort.ShellSort();
70              Console.WriteLine("Shell Sort");
71              Console.WriteLine("Comparisons: " + Sort.Comparisons);
72              Console.WriteLine("Assignments: " + Sort.Assignments);
73              sum = Sort.Comparisons + Sort.Assignments;
74              Console.WriteLine("Total Operations: " + sum);
75              Console.WriteLine();
76          }
77      }
78  }
```

Part 4 - Analysis

Analysis of Growth

Using a file with one million random numbers, I performed the three search algorithms and counted comparisons and array assignments.





These results meet the expectations I had in terms of growth. As we can see, the insertion and merge sort algorithms start fairly close in number of comparisons and assignments, but as the items sorted grows insertion sort begins to have vastly more operations done. With one million items, the difference is extremely large as would be expected of these two algorithms.

Comparison Across Data Sets

I ran each algorithm on each data set and added together the number of comparisons and assignments. That total number is used to compare the algorithms.

Here are my results:

Operations	Insertion Sort	Merge Sort	Counting Sort
sorted.txt	198	1028	100
nearly-sorted.txt	217	1031	100
duplicate.txt	4819	1206	100
shuffled.txt	5567	1222	100
nearly-unsorted.txt	9991	992	100
unsorted.txt	9999	988	100

As we would expect, insertion sort performs very well when the input is already sorted or mostly sorted. Conversely, it performs quite poorly on unsorted numbers.

Merge sort on the other hand is much more reliable. The number of comparisons + assignments performed does not vary much, regardless of the layout of the numbers that are input.

Counting sort always takes only N assignments, as expected.

Conclusions

From the data gathered about Insertion, Merge, and Counting sorts, I've come to a few conclusions.

It's obvious that a non-comparison based sort, like the counting sort I've implemented here, is the ideal way to sort numbers if the necessary characteristics (like a known, relatively small, range) are met. Many default sorting algorithms are comparison based sorts, and it's easy to see how knowledge of different algorithms can yield much better results.

If a comparison based sort needs to be used, then merge sort is a good option. The stability seen in the number of comparisons and assignments, regardless of input, means that the results will be consistent and good.

Insertion sort is not a good choice when the input is unknown, but it may be a good choice in some situations, despite being a naive algorithm. When numbers are nearly sorted, or when the total number of items is very small, insertion sort actually outperforms merge sort.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* 2009. p. 18

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* 2009. p. 32

[3] Sedgewick, Robert, and Kevin Daniel Wayne. *Algorithms, Fourth Edition.* 2011. p. 271