Programming Assignment 1
Sorting Algorithms

CSCI 3412 – Algorithms
Fall 20185
Kulprawee Prayoonsuk

Introduction

The purpose of this analysis is to study the number of comparison and assignment operations of sorting algorithm with different time complexity in order to study how operation can impact runtime and most importantly growth rate of an algorithm. I have selected three different sorting algorithm min order to make this analysis. The first algorithm, BUBBLE-SORT is a naïve sort that repetitively make comparison with adjacent item. Second the MERGE-SORT, a more advance sorting algorithm that utilize the divide and conquer and recursion technique. Lastly, the RADIX-SORT which sort array by comparing digit by digit. All algorithm was implemented in C++. Static analysis of each algorithm was done to show its correctness and expected order of growth. Dynamic analysis of the run-time performance of each algorithm was done against data sets with different ordering characteristics to determine the effect or data ordering of the expected order of growth.

Problem Definition

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, the algorithm we can derive a sorting algorithm that test successive comparison and assignment until it reaches assorted state. The problem is to select multiple sorting algorithm with different big O complexity, analyze it statically to show its correctness and estimate its expected rate of growth, implement it, and analyze its run time performance.

<u>Part 1 – Pseudo-Code</u>

BUBBLE-SORT

Bubble sort is a simple algorithm simply run through the array repeatedly and swap the adjacent element if it is in the order.

```
BUBBLESORT(A)
1  for i = 1 to A.length − 1
2      for j = A.length downto i + 1
3          if A[j] < A[j − 1]
4              exchange A[j] with A[j − 1]
```

MERGE-SORT

Merge sort utilize the Divide and Conquer and recursion technique in order to divide itself into two half and then call itself recursively again for each half then it uses the “merge” function to merge back the pieces into the sorted array.

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1 . . n1 + 1] and R[1 . . n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = 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
```

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

RADIX-SORT

Radix sort is a sort that compare the elements digit by digit sort starting rom the least significant digit to the most significant. first step involved find the highest numbers so that we can properly append zero to the number with less digit. Second part involved reaaranging order of elements by comparing each digit.

MAX-NUM(A)

```
1  max = A[0]
2  for i = 1 to A.length
3      if A[i] > max
4          exchange max with A[i]
```

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

```
RADIX-SORT(A, d)
1  for i = 1 to d
2      use a stable sort to sort array A on digit i
```

Part 2 - Static Analysis

BUBBLE-SORT

Initialization: i = 1 < A.length – 1, if this is true if mean that there are still element that need to be iterated through.

Maintenance: Assuming that not all A[j – 1] is < A[j] then the loop will continue.

Termination: Loop terminate when a sorted array is seen meaning all the A[j – 1] is < A[j]

MERGE-SORT

Initialization: Prior to the first iteration of the loop, we have k = p, so that the subarray A[p.. k-1] is empty. This empty subarray contains the k - p = 0 smallest elements of L and R, and since i = j = 1, both L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.

Maintenance: If L [i] $\leq R[j]$. Then L[ii] is the smallest element not yet copied back into A. Increasing k and i reestablishes the loop invariant for the next iteration.

Termination: The loop terminates when a sorted array is seen.

RADIX-SORT

Initialization: Before the start of i^{th} iteration, A is sorted according to the 1….$(i – 1)^{th}$ digits

Maintenance: Assuming that not all A is sorted according to the 1….d^{th} digits then the loop will continue.

then the loop will continue.

Termination: The loop terminates when A is sorted according to the 1….d^{th} digits.

Part 3 – Implementation

All algorithms were coded in C++. Since I used multiple .cpp and .h file, below is the relevant code, showing the sorting algorithm.

Bubble-Sort

```
void Sort::swap(int &x, int &y)
{
   int temp = x;
   x = y;
   y = temp;
}
void Sort::bubbleSort(vector<int>&vec)
{
   for (int i = 0; i < vec.size() - 1; i++)
   {
            for (int j = 0; j < vec.size() - i - 1; j++)
            {
                     compare++;
                     if (vec[j] > vec[j + 1])
                     {
                              swap(vec[j], vec[j + 1]);
                              assign++;
                     }
            }
   }
}
```

Merge-Sort

```
void Sort::merge(vector<int>&vec, int low, int high, int mid)
{
   int i, j, k;
   int size = high - low + 1;
   vector<int> c(size);
   i = low;
```

```
    k = 0; //the index holder
    j = mid + 1;

    while (i <= mid && j <= high)
    {
            compare++;
            assign++;
            if (vec[i] < vec[j])
            {
                    c[k] = vec[i];
                    k++;
                    i++;
            }
            else
            {
                    c[k] = vec[j];
                    k++;
                    j++;
            }
    }
    while (i <= mid)
    {
            c[k] = vec[i];
            k++;
            i++;
            assign++;
    }

    while (j <= high)
    {
            c[k] = vec[j];
            k++;
            j++;
            assign++;
    }
    assign += k;
    for (i = 0; i < k; i++)
    {
            vec[low + i] = c[i];
    }
}

void Sort::mergeSort(vector<int>&vec, int low, int high)
{
```

```
    if (low < high)
    {
            int mid;
            mid = (low + high) / 2;

            mergeSort(vec, low, mid);
            mergeSort(vec, mid + 1, high);

            merge(vec, low, high, mid);
    }
}
```

Radix-Sort

```
int Sort::MaxNum(vector<int>& vec, int vecLen)
{
    int max = vec[0];

    for (int i = 1; i < vecLen; i++)
    {
            compare++;
            if (vec[i] > max)
            {
                    max = vec[i];
                    assign++;
            }
    }
    return max;
}

void Sort::countSort(vector<int>& vec, int vecLen, int exp)
{
    vector<int>output(vecLen);
    int i;
    int count[10] = { 0 };

    for (i = 0; i < vecLen; i++)
    {
            compare++;
            count[(vec[i] / exp) % 10]++;
    }

    for (i = 1; i < 10; i++)
    {
            compare++;
```

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

    for (i = vecLen - 1; i >= 0; i--)
    {
        compare++;
        output[count[(vec[i] / exp) % 10] - 1] = vec[i];
        assign++;
        count[(vec[i] / exp) % 10]--;
    }

    for (i = 0; i < vecLen; i++)
    {
        compare++;
        vec[i] = output[i];
        assign++;
    }
}

void Sort::radixSort(vector<int>& vec, int vecLen)
{
    int max = MaxNum(vec, vecLen);
    for (int exp = 1; max / exp > 0; exp *= 10)
    {
        compare++;
        countSort(vec, vecLen, exp);
    }

}
```

Part 4 – Analysis

Analysis of Growth

Please note that I include all comparison and assignment that the sorting algorithm produce. Meaning comparison is everything that involved comparison operator. Please refer to the implementation to see the placement of the comparison and assignment counter.

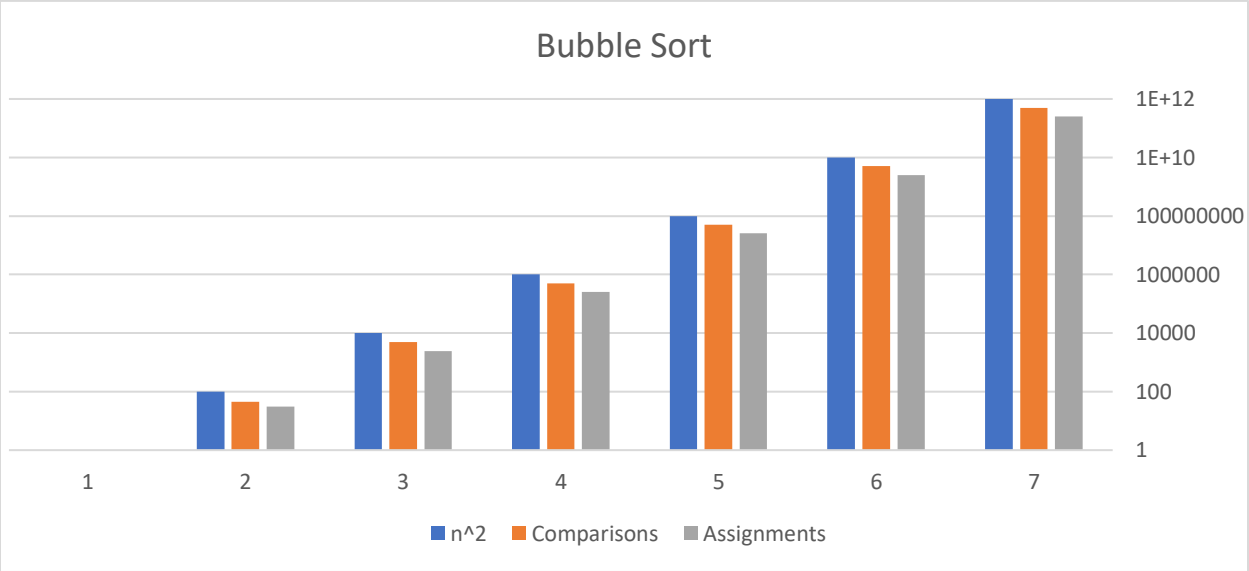
BUBBLE-SORT

Here are the results of running bubble sort for up to n = 1000000.

n:	Comparisons	Assignments
1	0	0
10	45	31
100	4851	2374

1000	498501	253402
10000	49995000	25277175
100000	4999950000	2501389533
1000000	499999500000	249966851045

Below are plot showing the growth of both the number of comparison and assignment compare to the growth of n^2. As you can see it is growing as roughly the rate of n^2 therefore establish bubble sort to have complexity of $O(n^2)$.

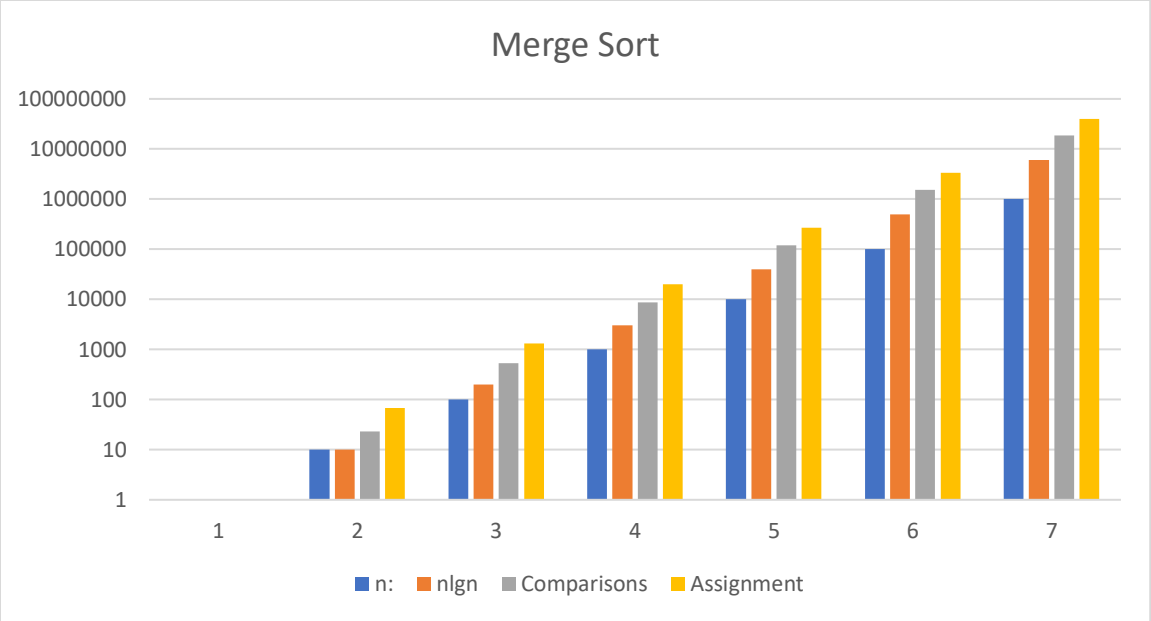


MERGE-SORT

Here are the results of running merge sort for up to n = 1000000.

n:	Comparisons	Assignment
1	0	0
10	23	68
100	532	1328
1000	8682	19930
10000	120473	267232
100000	1536328	3337856
1000000	18668784	39902848

Below are plot showing the growth of both the number of comparison and assignment compare to the growth of n and nlgn. As you can see it is growing as roughly the rate of nlgn therefore establish merge sort to have complexity of O(nlgn).

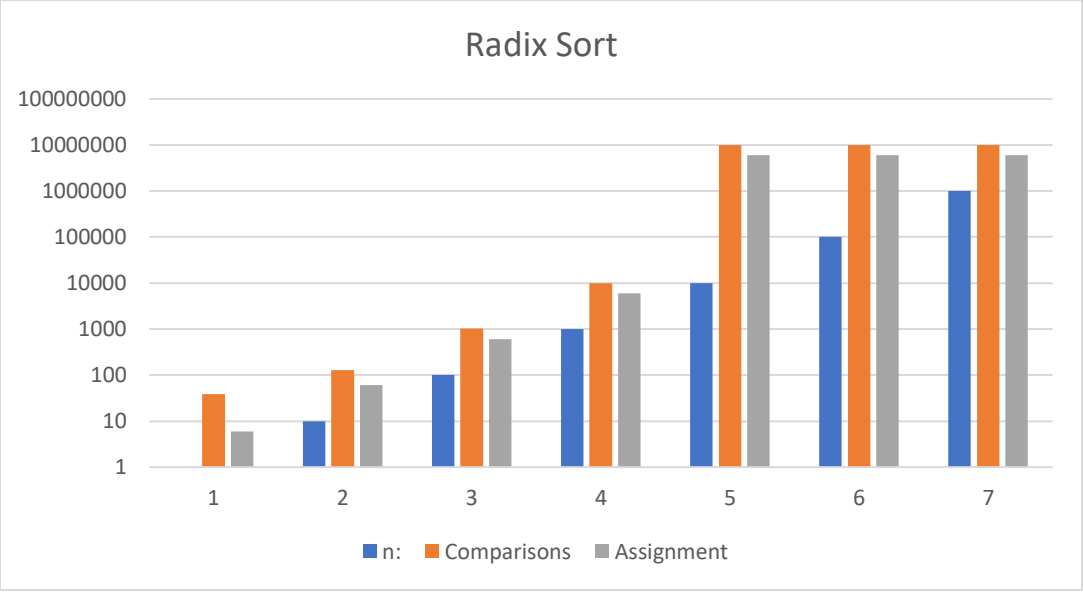


RADIX-SORT

Here are the results of running radix sort for up to n = 1000000.

n:	Comparisons	Assignment
1	39	6
10	129	61
100	1019	596
1000	10019	5999
10000	10000029	6000006
100000	10000029	6000006
1000000	10000029	6000006

Below are plot showing the growth of both the number of comparison and assignment compare to the growth of n. We know that Radix sort have complexity of O(n) although my graph does not reflect that. Although the implementation of the Radix sort result in correct result. I suspect that the counter was simply place in the wrong place thus showing in correct result because at n = 1, comparison and assignment should have both been 0.



Comparison Across Data Sets

Bubble-Sort		
File	Comparisons	Assignments
shuffled.txt	499500	253081
sorted.txt	499500	0
nearly-sorted.txt 1	499500	105
unsorted.txt	499500	499500
nearly-unsorted.txt	499500	499381
duplicate.txt	499500	244865

Since bubble sort compares each element to each other element, the amount of comparisons grows exponentially as the number of elements increases. This is not an efficient sort due to the fact that it need to run through all the comparison even if the array was sorted.

Merge-Sort		
File	Comparisons	Assignments
shuffled.txt	8718	19952
sorted.txt	5044	19952
nearly-sorted.txt 1	5095	19952
unsorted.txt	4932	19952
nearly-unsorted.txt	4987	19952
duplicate.txt	8702	19952

Because merge more assignment into a new array and move element into the existing array regardless of the outcome of comparison therefore regardless of the data the amount of comparison will be consistent.

Radix-Sort		
File	Comparisons	Assignments
shuffled.txt	10029	6001
sorted.txt	10029	6999
nearly-sorted.txt 1	10029	6902
unsorted.txt	10029	6000
nearly-unsorted.txt	10029	6000
duplicate.txt	10029	6003

For each digit only one pass the data and since in the beginning all number are appended with leading zeros to have the same amount of digit as the highest number, thus the same number of comparison is perform each time if the input is of the sane size n.

Conclusions

More complex sorting algorithm have condition that allow them to = require less comparisons and assignment operator which time is consuming to calculate therefore the least number of comparison and assignment that an algorithm need to order to sort the better it time complexity. Although bubble sort is simple it become useless as the number of elements grew, thus it is always better to select a more complex sort such as merged sort even though it may cost more space.

References:

https://www.geeksforgeeks.org/merge-sort/

https://www.geeksforgeeks.org/radix-sort/

Introduction to Algorithms (3rd Edition)