CS590 – Algorithms Assignment 2

Name: Sushant Bhat

CWID: 10474365

Recurrences: Solve the following recurrences using the substitution method. Subtract off a lower-order term to make the substitution proof work or adjust the guess in case the initial substitution fails.

1. $T(n) = T(n-3) + 3\lg n$. Our guess: $T(n) = O(n \lg n)$.
   Show $T(n) \leq cn \lg n$ for some constant $c > 0$.
   (Note: $\lg n$ is monotonically increasing for $n > 0$)
2. $T(n) = 4T\left(\frac{n}{3}\right) + n$. Our guess: $T(n) = O(n^{\log_3 4})$.
   Show $T(n) \leq cn^{\log_3 4}$ for some constant $c > 0$.
3. $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + T\left(\frac{n}{8}\right) + n$. Our guess: $T(n) = O(n)$.
   Show $T(n) \leq cn$ for some constant $c > 0$.
4. $T(n) = 4T(\frac{n}{2}) + n^2$. Our guess: $T(n) = O(n^2)$.
   Show $T(n) \leq cn^2$ for some constant $c > 0$.

Solution:

1. T(n) = T(n-3) + 3log n.

   Our guess: T(n) = O(n log n).

   Therefore, T(n) = T(n-3) + 3log n

   = c(n-3) log(n-3) + 3log n

   = c.n log(n-3) – 3.c log(n-3) + 3 log n

   Therefore, T(n) is not <= c.log(n-3) – 3.c log(n-3) + 3 log

   Now subtracting lower order,

   T(n) <= c.n log n – d.log n

   T(n) = (c(n - 3) log(n-3) – d.log(n-3))

   = c.log n – d.log n

   <= c.n log n – dn

   If d >= 2

2. $T(n) = 4T(n/3) + n$.

   Our guess: $T(n) = O(n^{\log_3 4})$

   $= 4T(n/3) + n$

   $= 4.c(n/3)^{\log_3 4} + n$

   $= (4/3^{\log_3 4})\, c(n)^{\log_3 4} + n$

   Let, $b = (4/3^{\log_3 4})\, c$ (constant)

   Therefore, $T(n) = bn^{\log_3 4} + n$

   Thus, $T(n)$ not $<= cn^{\log_3 4}$

   Now subtracting lower order,

   $T(n) <= O(n^{\log_3 4} - n)$

   $T(n) = 4(c(n/3)^{\log_3 4} - (n/3)) + n$

   $= cn^{\log_3 4} - (4/3)\, cn + n$

   $<= cn^{\log_3 4}$ for all $c > 1$

3. $T(n) = T(n/2) + T(n/4) + T(n/8) + n$

   Our Guess: $T(n) = O(n)$

   $= c(n/2) + c(n/4) + c(n/8) + n$

   $= c((n/2) + (n/4) + (n/8)) + n$

   $= c(7/8)n + n$

   $= ((7/8)c + 1)\, n$

   $<= c.n$ for all $c >= 8$

4. $T(n) = 4T(n/2) + n^2$.

   Our Guess: $T(n) = O(n^2)$

   $<= c.n^2$ belongs to $c > 0$

   Assuming it's true for all $k < n$

   $T(n) = 4T(n/2) + n^2$

   $= 4.\ c(n/2)^2 + n^2$

   $= 4.c(n/4)^2 + n^2$

   $= c.n^2 + n^2$

   $= (c + 1)\ n^2$ which is a failed case

   Now our new guess is:

   $T(n) = O(n^2 \log n)$

   $T(n) = 4T(n/2) + n^2$

   $<= 4.c.(n/2)^2 \log(n/2) + n^2$

   $= c.n^2 \log(n/2) + n^2$

   $= c.n^2 \log n – c.n^2 \log 2 + n^2$

   $<= c.n^2 \log n$ for all $c > 1$

Report

Abstract:

This report discusses and provides findings on 3 algorithms that have been used to sort an array of strings namely, Insertion Sort, Counting Sort and Radix Sort.

The problem statement is as follows: We need to Implement Insertion Sort as well as Counting Sort to sort an array of strings and use these sorting functions to implement Radix Sort.

Algorithms:

Three algorithms have been implemented for this assignment:

1. **Insertion Sort:**
   Insertion Sort is an efficient algorithm at sorting a small dataset. While it is not the best or the most efficient sorting algorithm, it is relatively easier to implement than many other sorting algorithms and is best when the data is almost or partially sorted.

```
InsertionSort(array)
  mark first element as sorted
  for each unsorted element X
    'extract' the element X
    for j <- lastSortedIndex down to 0
      if currentElement j > X
        move sorted element to the right by 1
    break loop and insert X here
end InsertionSort
```

2. **Counting Sort:**
   Counting Sort algorithm is a sorting algorithm that sorts the elements based on their number of occurrences in the array and allotting to their respective indexes in a temporary array. This algorithm is kind of a hashing.

```
CountingSort(array, n)
  Find largest element among nth place elements
  Create a COUNT array and initialize it with all
zeros
  for j <- 0 to size
    Find the total count of each unique digit in
nth place of elements and
    Store the count at jth index in count array
  for i <- 1 to maxLength
    Find the cumulative sum and store it in COUNT
array itself
  for j <- size down to 1
    Restore the elements to array
    Decrease count of each element restored by 1
```

3. **Radix Sort:**

   Radix Sort, also known as Bucket Sort or Digital Sort is a unique non-comparative sorting algorithm. When elements are in the range from 1 to n^2, the performance of counting sort is $O(n^2)$ which is worse than comparison-based sorting algorithms and this is when Radix sort comes into the picture. Radix Sort avoids comparisons altogether by sorting the elements digit by digit, be it integers, words, or character. Radix Sort still uses Counting Sort as a subroutine to perform digit by digit sorting. Radix sort creates buckets of length 0 – 9 only. This becomes more clearer in the algorithm below:

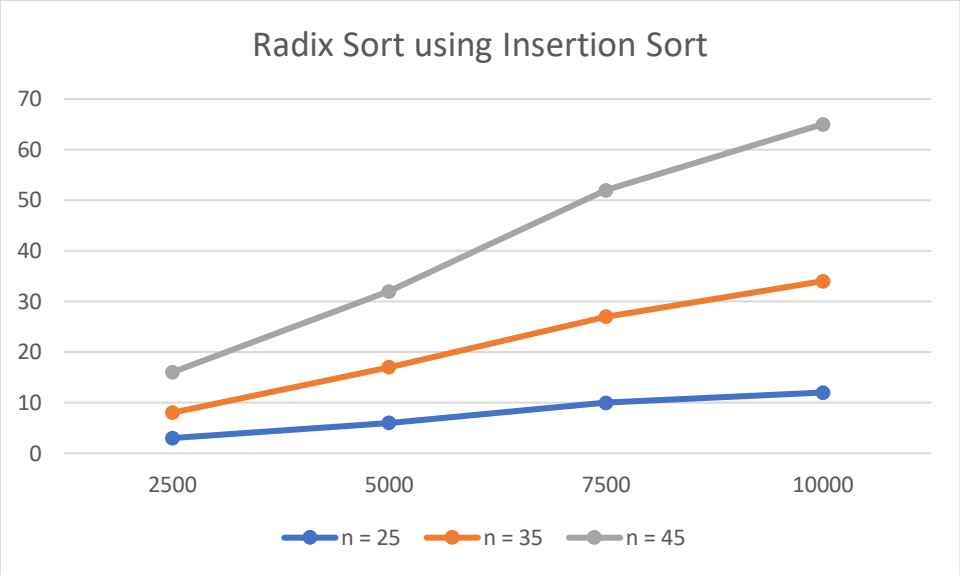
```
RadixSort(array)
  Find max number of digits in the largest
element
  Create d buckets of size 0-9
  for i <- 0 to d
    Sort the elements according to ith place
digits using countingSort

CountingSort(array, n)
  Find largest element among nth place elements
  Create a COUNT array and initialize it with all
zeros
  for j <- 0 to size
    Find the total count of each unique digit in
nth place of elements and
    Store the count at jth index in count array
  for i <- 1 to maxLength
    Find the cumulative sum and store it in COUNT
array itself
  for j <- size down to 1
    Restore the elements to array
    Decrease count of each element restored by 1
```

Following are the running times and graph plots of Radix Sort using Insertion Sort as well as Counting Sort algorithms:

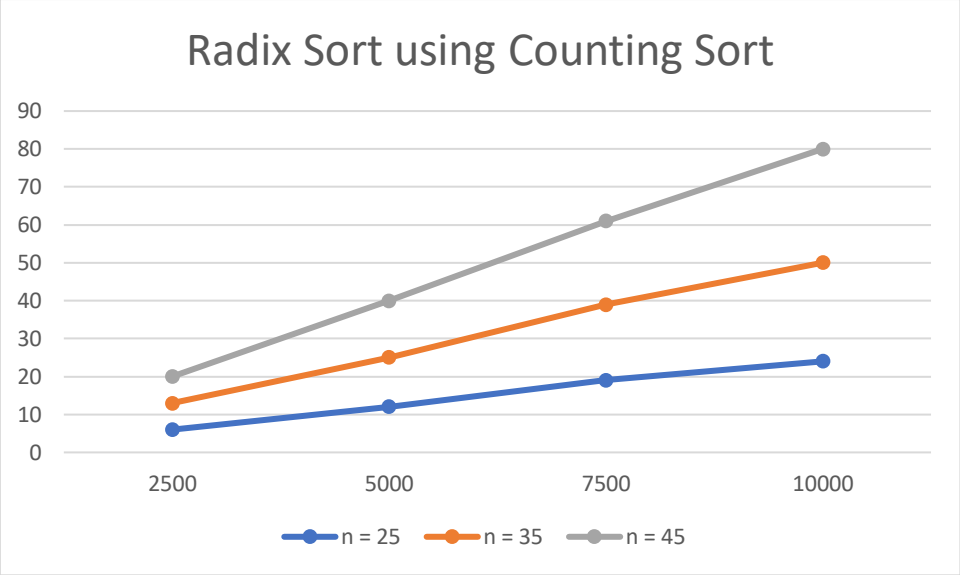
Radix Sort using Insertion Sort :

Radix Sort using Insertion Sort			
m	n = 25	n = 35	n = 45
2500	3	5	8
5000	6	11	15
7500	10	17	25
10000	12	22	31



Radix Sort using Insertion Sort :

Radix Sort using Counting Sort			
m	n = 25	n = 35	n = 45
2500	6	7	7
5000	12	13	15
7500	19	20	22
10000	24	26	30



Conclusion:

Insertion Sort is a very basic sorting algorithm which cannot be used for larger data sets. Counting sort is efficient if the range of input data is not significantly greater than the number of objects to be sorted. Comparatively, Radix Sort is far more efficient in terms of performance when it comes to memory storage as well as time complexity.