CS590 homework 2 – Recurrences and sorting

Recurrences: Solve the following recurrences using the substitution method. Subtract off a lower-order term to make the substitution

1) $T(n) = T(n-3) + 3\log n$. Our guess: $T(n) = O(n \log n)$
Show $T(n) <= cn \log n$ for some constant $c > 0$
(Note: $\log n$ is monotonically increasing for $n > 0$)

Solution:

$T(n) = T(n-3) + 3\log n$ .... (i)

Assuming that it is true for all $m < n$
We must prove $T(n) <= cn \log n$ assuming $T(m) <= cm \log m$ for all $m < n$
Let $m = (n-3) < n$

$= c(n-3) \log(n-3) + 3\log n$
$= cn \log(n-3) – 3c \log(n-3) + 3\log n$

Therefore $T(n)$ is not equal to $cn \log(n-3) – 3c \log(n-3) + 3\log n$
$T(n) <= cn \log n – d\log n$
$= c(n-3)\log(n-3) – d\log(n-3)$
$= cn \log n – d\log n$
$= cn \log n – dn$
$<= cn \log n – dn$
If $d >= 2$

----

2) $T(n) = 4T(n/3) + n$. Our guess : $T(n) = O(n^{\log_3 4})$.
Show $T(n) <= cn^{\log_3 4}$ for some constant $c > 0$.

Solution:

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

Assuming that it is true for all $m < n$
We must prove $T(n) <= cn^{\log_3 4}$ assuming $T(m) <= cm^{\log_3 4}$ for all $m < n$

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

Let us guess that $T(n) = O(n^{\log_3 4} – n)$
$T(n) = 4T(n/3) + n$
$<= 4c((n/3)^{\log_3 4} – n/3) + n$

$T(n) <= cn^{\log_3 4} – (4/3)cn + n$
$T(n) <= cn^{\log_3 4}$
for all $c > 2$

---

3) $T(n) = T(n/2) + T(n/4) + T(n/8) + n$. Our guess: $T(n) = O(n)$.
Show $T(n) <= cn$ for some constant $c > 0$.

Solution:

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

Assuming that it is true for all $m < n$
We must prove $T(n) <= cn$ assuming $T(m) <= cm$ for all $m < n$

$T(n) = c(n/2) + c(n/4) + c(n/8) + n$
$T(n) = ((4cn + 2cn + cn)/8) + 8n$
$T(n) = ((7/8)c+1)n$
$T(n) <= cn$ for all $c >= 8$
Therefore $T(n) <= c(n)$

---

4) $T(n) = 4T(n/2) + n^2$. Our guess: $T(n) = O(n^2)$.
Show $T(n) <= cn^2$ for some constant $c > 0$.

Solution:

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

Assuming that it is true for all $m < n$
We must prove $T(n) <= cn^2$ assuming $T(m) <= cm^2$ for all $m < n$

$T(n) = 4c(n/2)^2 + n^2$
$T(n) = 4(cn^2/4) + n^2$
$T(n) = cn^2 + n^2$
$T(n) = n^2(c+1)$ which fails

Let us assume new guess as $T(n) = O(n^2 \log n)$
$T(n) = 4T(n/2) + n^2$
$\quad = 4c(n/2)^2\log(n/2) + n^2$
$\quad = cn^2\log(n/2) + n^2$
$\quad = cn^2\log n – cn^2\log 2 + n^2$
$\quad <= cn^2\log n$
for all $c > 1$

Report - Recurrences and sorting

1. Abstract:

We are given a string C which is represented by an array of characters (e.g. char *st). Each character present in the string C is an ASCII representation of a specific integer value. The character "a" is a representation of the value 97 according to the ASCII table. This gives us a way to view any array or string as a digit. We are implementing sort algorithms for strings which will be sorting the strings according to the character present at position d.

The first given algorithm is Insertion Sort. We must improve and implement the Radix Sort using the modified insertion sort to measure the runtime performance for specific values arrays. We must also implement a Count sort and use it in the Radix Sort to sort the given array of strings with the given length.

2. Result:

Testing sorting algorithms:

Three different types of sorting algorithms were implemented and the time complexity of each one of them was measured to complete the given experiment. The three algorithms are as follows:

- Insertion Sort
- Counting Sort
- Radix Sort

Insertion Sort:

In this type of sorting algorithm, we start from the second element and compare it with first element. Then we perform this operation for all the remaining elements. The rows are present in a multidimensional way. For each row, we are checking the length of the corresponding array element with the character present at the d position. We are comparing the length of array at a given position d with the value present at the position [i][d] of the string array each time when the loop is being traversed. Comparing of length and sorting of the string array takes place at the same time. Because of this, time taken by this sorting algorithm is more and is not efficient to perform sorting.

General Algorithm:

- Step 1: Declare an integer variable key_length to store length of key and two character variables temp_id and key_id to store A[i][d] value and key[d] value of the string array.
- Step 2: Declare a For loop where j starts from 1 till the length of string array to compare and store length of each string array at position d into temp_id.
- Step 3: Set the value of key as A[j] and value of key_length as A_len[j].

- Step 4: Set the value of i as j - 1.
- Step 5: In the while loop, check whether i >= 1 && temp_id > key_id.
- Step 6: Till the while condition satisfies, A[i+1] = A[i], A_len[i+1] = A_len[i] and i is decrement by 1.
- Step 7: After while condition fails, key is assigned to A[i+1], key_length is assigned to A_len[i+1] and j is incremented for next iteration.
- Step 8: Exit.

Counting Sort:

This is a type of sorting algorithm which depends on data present between a specific range. It depends on the range of input data. Firstly, the max value of the input array is calculated. After calculating the max value, occurrences of each element present in the array is counted from the first element to the last element of the array. The count of the occurrences is stored in a temporary array. The temporary array of occurrences is used to sort the given input data.

General Algorithm:

- Step 1: Declare an array countArray of size 256.
- Step 2: Declare a For loop to initialize each element of countArray to 0.
- Step 3: Declare a For loop from 0 to size of array to count of occurrences of array elements and store them in the countArray.
- Step 4: Declare a For loop to find the cumulative sum values of the count of occurrences by performing countArray[i] += countArray[i-1].
- Step 5: Declare a For loop to copy the element and the A_len or B_len back into the input array and decrement the count of each element by 1.
- Step 6: Exit.

Radix Sort:

Radix sort is used to compare input data based on the single character present in the string. For strings with more than one character, the process is repeated for each character while keeping the order of characters intact. The process of putting the characters in a bucket of index stops when each character is taken into consideration. Insertion Sort and Counting Sort algorithm is used to sort the array of strings using Radix Sort.

General Algorithm:

- Step 1: Find the max value in the given input array.
- Step 2: Define the buckets based on the ASCII value of max value present in the input array.
- Step 3: Consider the character present at the unit’s place of each string present in the input array to be sorted.

- Step 4: Insert the taken character into the bucket according to the index equal to the ASCII value.
- Step 5: Take out all the characters present in the bucket from first index to last index in the order.
- Step 6: Repeat the process from step 3 by just changing the character which is taken into consideration. Take the character present to the left of the previous character.
- Step 7: Repeat all the process until the array of strings is sorted.
- Step 8: Stop.

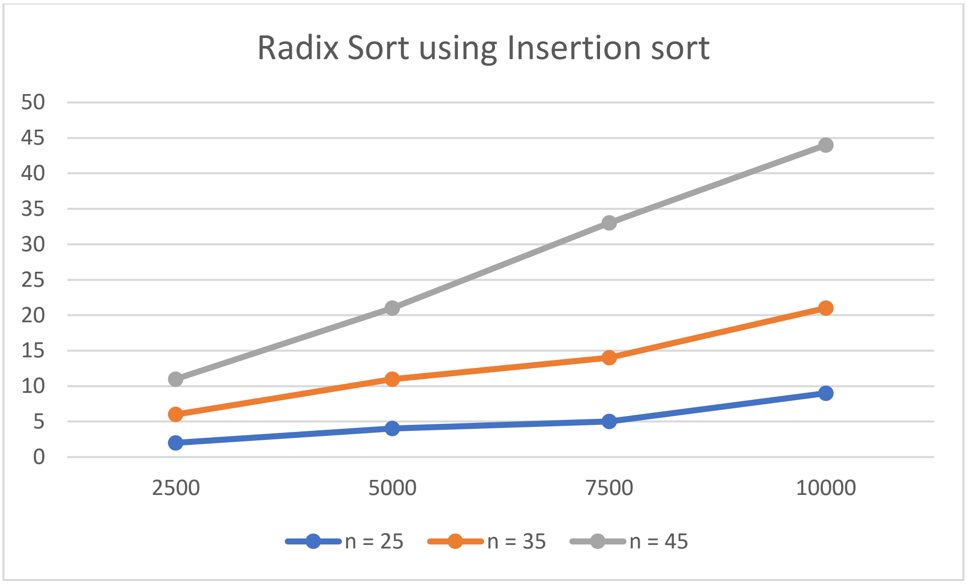
Testing Values:

1. **Radix Sort using Insertion Sort:**

- The runtime in ms for the Radix sort using Insertion Sort is given below:

	n = 25	**n = 35**	**n = 45**
Size (m)	**Runtime Performance (ms)**	**Runtime Performance (ms)**	**Runtime Performance (ms)**
2500	2	4	5
5000	4	7	10
7500	5	9	19
10000	9	12	23

- The runtime taken by the Insertion Sort which is called through the Radix sort is less when the length of the string (m) and the size of the array (n) is small.
- As the size of the array and the length of the string increases, the time taken to sort the input string also increases.
- The length of the string is also compared to sort the given input array.
- Insertion sort is used to compare the length of the string with the next element according to the character present at position d.
- Insertion sort works best and is efficient when the size of the given input data is small
- The Input size vs Runtime for the Radix Sort using Insertion Sort is given below:

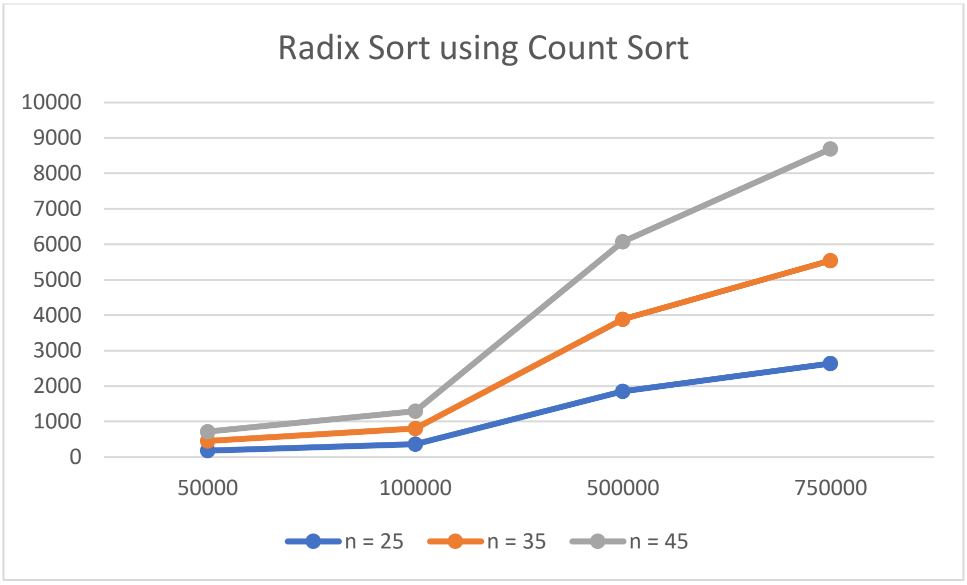


2. **<u>Radix Sort using Counting Sort:</u>**

- The runtime in ms for the Radix sort using Counting Sort is given below:

	n = 25	**n = 35**	**n = 45**
Size (m)	**Runtime Performance (ms)**	**Runtime Performance (ms)**	**Runtime Performance (ms)**
50000	180	272	265
100000	361	441	489
500000	1849	2035	2185
750000	2637	2904	3146

- The runtime taken by the Counting Sort which is implemented through the Radix sort is less even when the length of the string (m) and the size of the array (n) is large as compared to insertion sort.
- As the size of the array and the length of the string increases, the time taken to sort the input string also increases.
- The length of the string is also compared to sort the given input array.
- Counting sort is used to count the occurrence of the character of the string and is stored in the temporary array.
- Counting sort works best and is efficient when the size of the given input data is large.
- The Input size vs Runtime for the Radix Sort using Counting Sort is given below:



3. Discussion

Running Time:

- Insertion sort is considered as an efficient algorithm when the array size is small. As it uses the while loop instead of for loop, the technique of sort does not perform extra iterations while sorting the array of strings. Even if the passed string array is sorted, insertion sort does not perform any extra iterations. The running time in case of modified insertion sort increases as the length of the string in the array and the array size increases.
- The time complexity of Counting Sort is O(n+k) where n is the number of elements present in the array of strings and k is the range of the elements.
- Time taken by Counting Sort increases when the range of the given data is larger than the other elements. As the range increases, time taken by the Counting Sort also increases. When all the array elements are of the same range, Counting Sort takes constant time to sort the given array, which makes its time complexity to O(n) which is also called as linear.
- The runtime of Radix Sort is O(nk) where n is the length of the array and k is the maximum number of digits.
- When all the elements have the same number of characters present in the string and if there is one element which has significantly large number of characters present, then time taken by Radix Sort is more.
- On the other hand, if all the elements have same number of characters then, it takes less amount of time to sort the data.

Limitations:

- As the insertion sort takes n-squared steps for every element which needs to be sorted, there is a problem when the size of array of strings which is passed is large.

- Hence Radix Sort using Insertion Sort and can be used and can be efficient when it comes to sorting array of strings up to a limited length.
- Counting sort on the other hand only works when the range of the data elements present in the array is known.
- If the range of the elements is very high, then Counting sort requires lot of space to perform the sorting and lot of memory gets utilized.
- Counting Sort is not recommended to sort string values or large data set values.
- Radix Sort cannot be used for different types of data as it uses the letters and digits to perform sorting.
- Space required by Radix sort is more as compared to other sorting techniques.

Improvements:

- The implementation of counting sort can be improved when the range of the data is small and compact.
- The running time of the Radix Sort can be improved if the number of characters processed at each pass can be increased. If the number of characters to be processed is doubled then there is a possibility that the number of passes gets reduced by half.

4. Conclusion:

Hence, we can say that Counting Sort work best if the largest element is not significantly larger than the number of elements. Also, we can say that larger the range of the elements in the given array is, more the space is required by the Counting Sort. On the other hand, Radix sort is a good option for fast sorting. Radix sort also can handle larger data sets more efficiently as compared to the Counting Sort.