HOMEWORK 03

INTRODUCTION:

A non-comparative sorting algorithm called radix sort uses digit sorting to handle integers. Radix sort takes advantage of the internal structure of the keys being sorted, as opposed to comparison-based sorting algorithms like quicksort or merge sort, which evaluate digits to obtain their relative order. Sorting digit by digit, from right to left, or vice versa, is the basic principle behind radix sorting. Until every digit has been taken into consideration, each is analyzed separately, and the sorting procedure continues for every digit. Although radix sort can be used with any base (radix), it is most frequently applied to binary, decimal, or hexadecimal representations of numbers.

RADIX-SORT(A,d)
1 For i = 1 to d
2 Use a stable sort to sort array A on digit I [1]

This algorithm sorts these numbers in $\Theta(d(n+k))$ time if the included stable sorting algorithm has a time complexity of $\Theta(n+k)$ (i.e. counting sort). The analysis of the time depends on the stable sorting algorithms that are included. When every digit ranges from 0 to k-1, the counting sort is the best choice to make. Each pass over n d-digit number takes time of (n+k) which is comparatively less than any other sorting algorithm's average case running time.

In the given assignment, the random generator function generates some random array of strings as an input for sorting. A string is represented by an array of characters where each of the alphabets has a corresponding ASCII value. Every single character of the array is then accessed digit by digit. The function randomly chooses a length m between 1 to maximum. The length of the random string is stored for manipulation.

1.1 INSERTION SORT:

Insertion sort is a simple algorithm for sorting that uses comparisons to create the final array of sorted items by comparing a single item at a time. It gets the final sorted array by comparing one item at a time.
The second element in the array (or list) is thought to be the key at the beginning of the algorithm. Next, it proceeds from left to right, comparing this key with the elements to its left, until it determines the proper location for the key. It moves an element one position to the right if it discovers an element that is greater than the key while comparing it to elements to its left. The key is inserted into the designated spot after it has determined where it should go. The algorithm builds a sorted sub-array progressively from left to right by repeating this procedure for each element in the array. The method by which insertion sort operates is by keeping a sorted subarray and adding elements to it one at a time.

Here, we are supposed to use insertion sort for sorting the generated strings based on the position d. Here is the code that is implemented for digit insertion sort:

```
void insertion_sort_digit(char** A, int* A_len, int l, int r, int d)
{
    for (int i = l + 1; i <= r; ++i) {
        char* key = A[i];
        int key_len = A_len[i];
        int j = i - 1;
        while (j >= l && (A[j][d] > key[d] || (A[j][d] == key[d] && A_len[d] > key_len))) {
            A[j + 1] = A[j];
            A_len[j + 1] = A_len[j];
            j = j - 1;
        }
        A[j + 1] = key;
        A_len[j + 1] = key_len;
    }
}
```

Fig 1. Implemented code of Insertion sort based on digits.

EXPLANATION:

We are supposed to add 2 more parameters, namely the digit d and the length of each string A_len. In the loop of insertion sort, a conditional check to determine if the specified position d is present in both the current and key element. If d is more than the length of string then consider 0 to make sure the string is properly sorted. To improve the time and reduce the calculations, it is better to only compare the character at position d instead of the entire string.

Time Complexity:

The time complexity is based on 2 steps:

- During the comparison: Each iteration of the loop has access to the character at from two strings. Hence it takes O(n) comparisons.
- Swap and Assigning: Based on comparisons there are equal swapping and assigning the values. In the worst case, it executes for O(n) time.

The overall *time complexity* of this *insertion sort* is $O(n^2)$.

Now, this stable sorting algorithm is used in radix sort to sort the randomly generated arrays of strings.

1.2 RADIX SORT WITH INSERTION SORT:

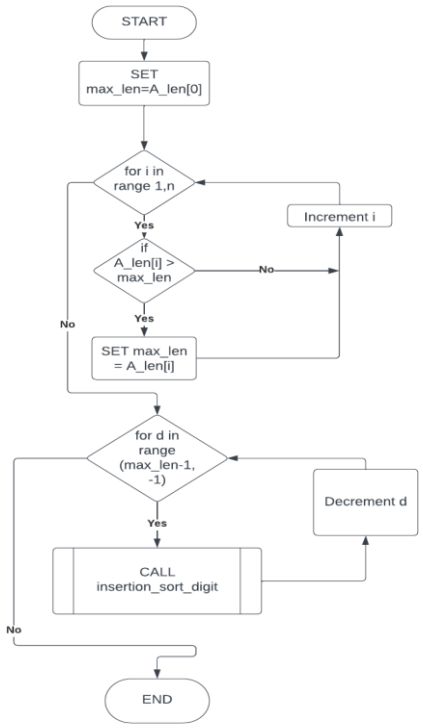


Fig 2. Flowchart of Radix sort using insertion sort digit as a stable sorting algorithm

```
void radix_sort_is(char** A, int* A_len, int n, int m)
{
 int max_len = A_len[0];
    for (int i = 1; i < n; ++i) {
        if (A_len[i] > max_len) {
            max_len = A_len[i];
        }
    }
    for (int d = max_len - 1; d >= 0; --d) {
        insertion_sort_digit(A, A_len, 0, n - 1, d);
    }
}
```

Fig 3. Code for implementation of the above insertion sort with radix sort

EXPLANATION:

The above snippet of the code is the implementation of radix sort using the above insertion sort. The function radix_sort_is sorts an array of string A. It first determines the maximum length of the string of the array. Later, it iterates from MSB to LSB of the string. During iteration for the function of sorting, it calls the insertion_sort_digit function to sort strings based on the current digit. This completely sorts the string in ascending order from right to left using the radix sort algorithm.

Time Complexity:

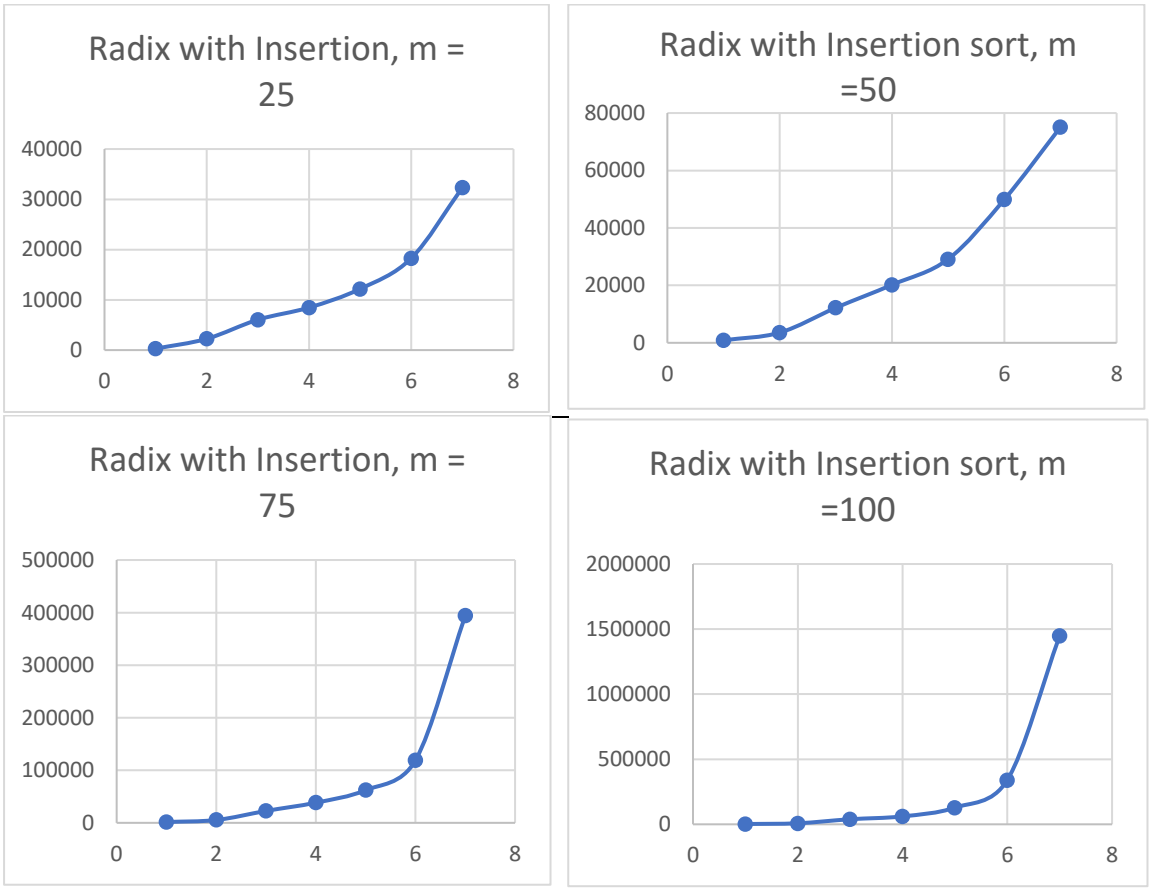
The maximum length has a time complexity of O(n). The outer loop iterated max_len times, so it takes O(max_len) time to run. The insertion_sort_digit sorts a portion of the array

based on the d digit. The inner loop in insertion_sort_digit iterated at the size of the portion being sorted (r-l). Hence, inserion_sort_digit has a time complexity of O(max_len * n).The overall time complexity of radix_sort_is *O(n+max_len+(max_len *n))*.
The worst case occurs when max_len is smaller than n. Hence, the time complexity for the worst case would be, *O(max_len * n)*.

Tabulation and Analysis:

Table 1. Running time of Radix sort with Insertion Sort in ms

n/m	***25***	***50***	***75***	***100***
5000	287.3	891	1571.8	2326.6
10000	2246.1	3556.7	5409.6	8010.7
20000	6048.1	12212.3	22714.6	39044.4
25000	8457.4	20193.2	38405.7	60944.11
30000	12169	29119.1	62266.4	128824
40000	18271.86	49888.29	119086.5	340450.8
50000	32318.7	75078.27	394216.6	1448394



For each value of m, the graph is exponentially increasing, as the time complexity depends on *max_len*n.*

1.3 COUNTING SORT:

This is an efficient way of sorting suitable for sorting in range. Every single element in the input array has its number of occurrences counted, and that information is then used to determine each element's position in the sorted output array.

Initially, the input value range is established to be between 0 and k. This is how the process operates. After that, the size k+1 axillary array C initializes to zero. After that, iterate through the input array, counting each element's occurrences, and increment the appropriate index on the C array[2]. Subsequently, the C array is changed such that the total of counts is stored in each element at i. This aids in figuring out each element's starting index within the sorted array. Next, go through the array in reverse order, count each element, and assign the element to the index that C was used to calculate. After reducing the count, carry out the remaining sorting.

If the range k is much smaller than the total number of elements that need to be sorted, counting sorting works well. Its O(n + k) total time complexity indicates that it is linear. For the counting and cumulative counting phases, its time complexity is O(n + k), and for the output phase, it is O(n). It might not be feasible to count sort for big sets since it needs extra space equal to the value being entered range k. In the sorted output, the relative order of equal elements is retained.Here is the code snippet for counting sort algorithm digit:

```
void counting_sort_digit(char** A, int* A_len, char** B, int* B_len, int n, int d)
{
    int C[256] = {0};
    for (int i = 0; i < n; i++)
        C[A[i][d]]++;
    for (int i = 1; i < 256; i++)
        C[i] += C[i - 1];
    for (int i = n - 1; i >= 0; i--) {
        int index = C[A[i][d]] - 1;
        B[index] = A[i];
        B_len[index] = A_len[i];
        C[A[i][d]]--;
    }
}
```

Fig 4. Code for implementation of counting sort digit

EXPLANATION:

The above snippet of the code is the implementation of counting sort. Firstly, initialized auxiliary array C of size 256 to store the count. Then iterates through the input array A, and counts the occurrence of the digit at position d in the string. It then increments the count C array corresponding to the ASCII value of character d. Then modify the C array such that all elements store the cumulative count of characters to that index. It is iterated through input array A in reverse, and for each string, it determines the index based on character position d. It places the string into the output array B at the index of C[index]-1 and updates the length accordingly. Then decrements the count in the C array for each corresponding character and it functions for the length of n for m set to produce the sorted output.

Time Complexity:

The counting phase takes time to iterate through the array A and takes O(n). Lately, the cumulative counting phase has O(256) times or we can state it as O(1). At the rearranging phase, even here it iterates through array A which again takes O(n) time. Hence the overall time complexity is dominated by both the counting phase and rearranging phase yielding *O(n)*.

1.4 RADIX SORT WITH COUNTING SORT:

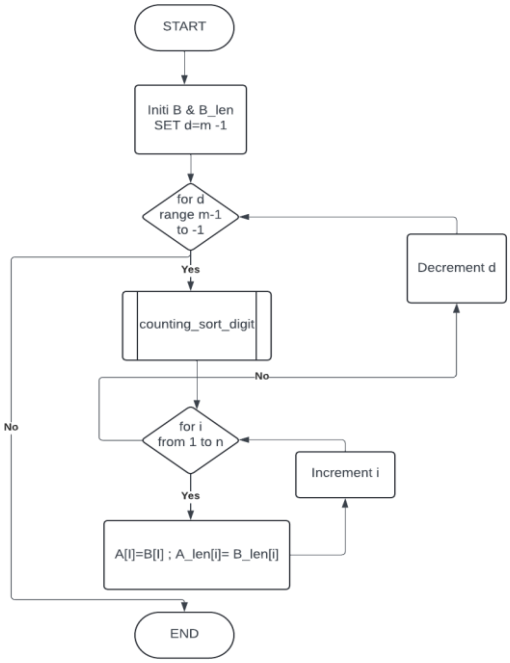


Fig 5. Flowchart of Radix sort using counting sort digit as a stable sorting algorithm

```
void radix_sort_cs(char** A, int* A_len, int n, int m)
{ char** B = (char**)malloc(n * sizeof(char*));
    int* B_len = (int*)malloc(n * sizeof(int));


    for (int d = m - 1; d >= 0; --d) {
        counting_sort_digit(A, A_len, B, B_len, n, d);
         char** temp = A;
        A = B;
        B = temp;

        int* temp_len = A_len;
        A_len = B_len;
        B_len = temp_len;
    }
    free(B);
    free(B_len);
}
```

Fig 6. Code for implementation of the above counting sort with radix sort

EXPLANATION:

The above snippet of the code is the implementation of radix sort using the above counting sort. The function radix_sort_cs sorts an array of string A. It first initializes the temporary array B and B_len dynamically, to store the sorted output and their length respectively. It iterates m times, which ensures that each digit from LSB to MSB is processed. At each iteration, the counting_sort_digit is called for sorting array A based on current digit d. During this process, it creates B and updates the original A with sorted elements. Once the sorting based on position is done, it updates the original array A with sorted elements in array B. This completely sorts the string in ascending order from LSB to MSB using the radix sort algorithm. Then finally, free the memory that was dynamically allocated for B and B_len.

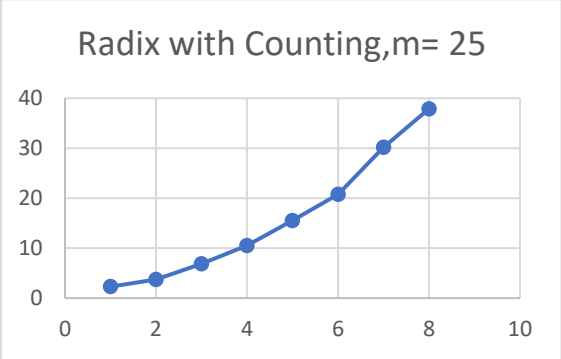
Time Complexity:

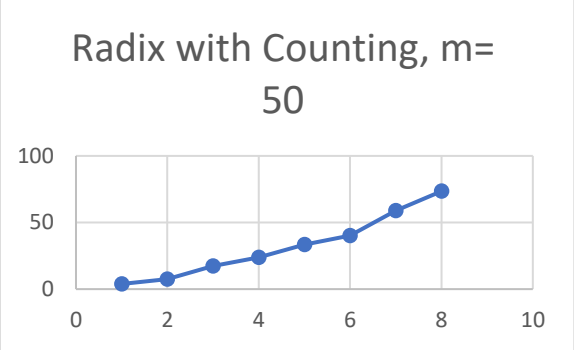
The total number of elements in the array, n, and the range of values, k, which in this case is 256, determine the time complexity of the counting sort, which is O(n+k)[2]. During the updating phase, the updating operation iterates via B, which has n elements, each time coping a string and its length. The sorting process takes O(m*n) because the counting of sorting is called m times in the loop and has an O(n) time complexity[1].

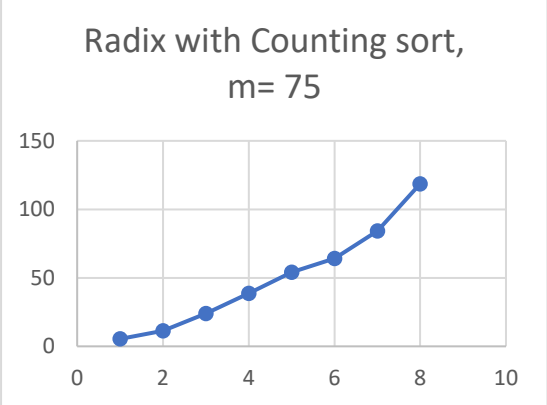
Tabulation and Analysis:

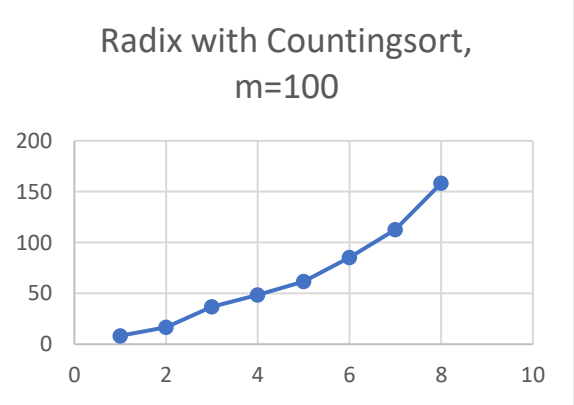
Table 2. Running time of Radix sort with Counting Sort in ms

n/m	***25***	***50***	***75***	***100***
5000	2.3	4.3	5.4	8.3
10000	3.7	7.4	11.3	16.6
20000	6.9	18.18	23.9	36.81
25000	9.3	23	29.4	51.45
30000	10.5	23.81	38.6	48.4
40000	15.5	34.1	54	61.7
50000	20.75	40.1	58.27	85.4
75000	30.16	62.45	84.2	112.6
100000	44.3	73.45	118.7	158.3
250000	145.2	205.18	439.8	443.7
500000	613.2	1013.8	1526.5	2385.7









For each value of m, the graph is *linearly* increasing, as the time complexity depends on *n*

CONCLUSION:

The time complexity of radix sort with insertion sort is O (n+max_len+(max_len *n)). This quadratic time arises from the use of insertion sort due to its method of iterating over each string and comparing each based on position. As the data size increases, the runtime also increases quadratically, longer strings require more time to compare at each position. As a result, this is not appropriate for big datasets of varying lengths.

The time complexity of the radix sort with counting sort is O(m*n). Since counting is a linear sorting algorithm, it improves the radix sort's time. This method is highly suitable, especially for large datasets and different lengths of strings. The linear time complexity of counting sort ensures that overall sorting is efficient. Applications for radix sort with counting sort can be found in many different fields, such as data processing, text processing, and string manipulation.

REFERENCES:

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009) *Introduction to Algorithms* (3rd ed.). The MIT Press.

[2] Sedgewick, R., & Wayne, K. (2011). *Algorithms(4th Edition)*. Addison-Wesley Professional.