University of Engineering and Technology

**Analysis Of Algorithms**

**Submitted By:**

(2019-CS-127) Muhammad Awais

**Submitted To:**

**Dr. Sadaf Hina**

Linear time Sorting Algorithms.

# Counting Sort:

Counting sort is linear time sorting algorithm. It is very efficient for small input values in this algorithm we define the range of our element and all element in our input array lies between this range of element which we already defined. The running time is mainly depends upon the range which we defined .if the range is small it’s a good running time algorithm but if the range is large then its going to be a very bad algorithm. In this algorithm we can write the output directly instead of writing the permutation of our out. And then we have some auxiliary storage. And for this storage we have length k which the range of our input.

To perform the counting sort the first step is just an initialization. Find out the maximum element from the given array .That maximum element is the range of particular array. We insert loop to define range and declare an array of integer to count the occurrence of elements. We set it to 0 for every value we increment it once. Initialize an array of length max+1 with all elements 0. This array is used for storing the count of the elements in the array.

Store the count of each element at their respective index in count array. Count of each element stored. Store cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array. Find the index of each element of the original array in the count array. This gives the cumulative count. Place the element at the index calculated. After placing each element at its correct position, decrease its count by one. Now you can simple print out the sorted array of elements.

## Pseudocode of Counting Sort:

for i<-1 to k

do C[i] <-0

for j<-1 to n

C[A[j]]<- C[A[j] +1]

for i<-2 to k

do C[i]=C[i]+C[i-1]

for j<-n down to 1

do B[C[A[J]]]<- A[j]

C[A[j]]<- C[A[j]]-1

## Time Complexity:

The first step of initialization will take k amount of time .Because loop is running from 1 t0 k.

**O(k)**

The next step is counting the occurrence of element in array. This will take linear time.

**O(n)**

The next step of empty the blocks will take k amount of time.

**O(k)**

The next step to print the sorted array will take linear time

**O(n)**

Total running time of this algorithm is **O(n+k)** which is good if k is relatively small.

The all we need not just the elements id integer but we need the range is pretty small .

So if k =n them we get the linear time complexity O(n).

## Strengths:

1. Linear Time Complexity. Since it is not a comparison-based sorting, it is not lower bounded by O(n log n) complexity.
2. Reduced space complexity if the range of elements is narrow, that is, more frequency of close integers.
3. Good for small number of inputs .

## Weakness:

1. Both time and space complexities skyrocket if the range of input numbers is large.
2. It works only for discrete values like integers.
3. In case, negative integers are involved, the complexity increases, as well as certain changes in the algorithm, are required.

## Dry Run:

## 

# Radix Sort:

Radix sort is an integer sorting algorithm that sorts data with integer keys by grouping the keys by individual digits that share the same significant position and value (place value). Radix sort uses counting sort as a subroutine to sort an array of numbers. Because integers can be used to represent strings (by hashing the strings to integers), radix sort works on data types other than just integers. Because radix sort is not comparison based, it is not bounded by \Omega(n \log n)Ω(nlogn) for running time — in fact, radix sort can perform in linear time.

Radix sort incorporates the counting sort algorithm so that it can sort larger, multi-digit numbers without having to potentially decrease the efficiency by increasing the range of keys the algorithm must sort over (since this might cause a lot of wasted time).

Radix sort works by sorting each digit from least significant digit to most significant digit. So in base 10 (the decimal system), radix sort would sort by the digits in the 1's place, then the 10’s place, and so on. To do this, radix sort uses counting sort as a subroutine to sort the digits in each place value. This means that for a three-digit number in base 10, counting sort will be called to sort the 1's place, then it will be called to sort the 10's place, and finally, it will be called to sort the 100's place, resulting in a completely sorted list. Here is a quick refresher on the counting sort algorithm.

Counting sort can only sort one place value of a given base. For example, a counting sort for base-10 numbers can only sort digits zero through nine. To sort two-digit numbers, counting sort would need to operate in base-100. Radix sort is more powerful because it can sort multi-digit numbers without having to search over a wider range of keys (which would happen if the base was larger).

## Pseudocode:

for j = 1 to d do

//A[]-- Initial Array to Sort

int count[10] = {0};

//Store the count of "keys" in count[]

//key- it is number at digit place j

for i = 0 to n do

count[key of(A[i]) in pass j]++

for k = 1 to 10 do

count[k] = count[k] + count[k-1]

//Build the resulting array by checking

//new position of A[i] from count[k]

for i = n-1 downto 0 do

result[ count[key of(A[i])] ] = A[j]

count[key of(A[i])]--

//Now main array A[] contains sorted numbers

//according to current digit place

for i=0 to n do

A[i] = result[i]

end for(j)

end func

## Time Complexity:

In the Radix sort algorithm running time depends on the intermediate sorting algorithm which is counting sort.

If the range of digits is from 1 to k, then counting sort time complexity is O(n+k).

There are d passes i.e counting sort is called d time, so total time complexity is O(nd+nk) =O(nd). As k=O(n) and d is constant, so radix sort runs in linear time.

Worst Case Time complexity: O (nd)

Average Case Time complexity: O(nd)

Best Case Time complexity: O(nd)

## Strengths:

1. Fast when the keys are short i.e. when the range of the array elements is less.
2. Used in suffix array construction algorithms like Manber's algorithm and DC3 algorithm.
3. Useful for large number of inputs.

## Weakness:

1. Since Radix Sort depends on digits or letters, Radix Sort is much less flexible than other sorts. Hence , for every different type of data it needs to be rewritten.
2. The constant for Radix sort is greater compared to other sorting algorithms.
3. It takes more space compared to Quicksort which is in place sorting.

## Dry Run:

