* Comparison of Sorting Algorithm

Date.....

Name	Average Case	Worst Case	Auxiliary Space
Bubble Sort	$O(n^2)$	$O(n^2)$	1
Selection Sort	$O(n^2)$	$O(n^2)$	1
Insertion Sort	$O(n^2)$	$O(n^2)$	1
Shell Sort	-	$O(n\log^2 n)$	1
Merge Sort	$O(n\log n)$	$O(n\log n)$	depends
Heap Sort	$O(n\log n)$	$O(n\log n)$	1
Quick Sort	$O(n\log n)$	$O(n^2)$	$O(\log n)$
Tree Sort	$O(n\log n)$	$O(n^2)$	$O(n)$

non-comp based sort algorithm

* Linear Sorting Algorithms :-

All the other sorting algorithms are comparison based sorting algorithms. Among them, the best comparison based sorting has the complexity $O(n\log n)$. There are also some sorting algorithm which are non-comparison based, also called Linear Sorting Algorithms. Few examples of Linear Sorting Algorithms are :-

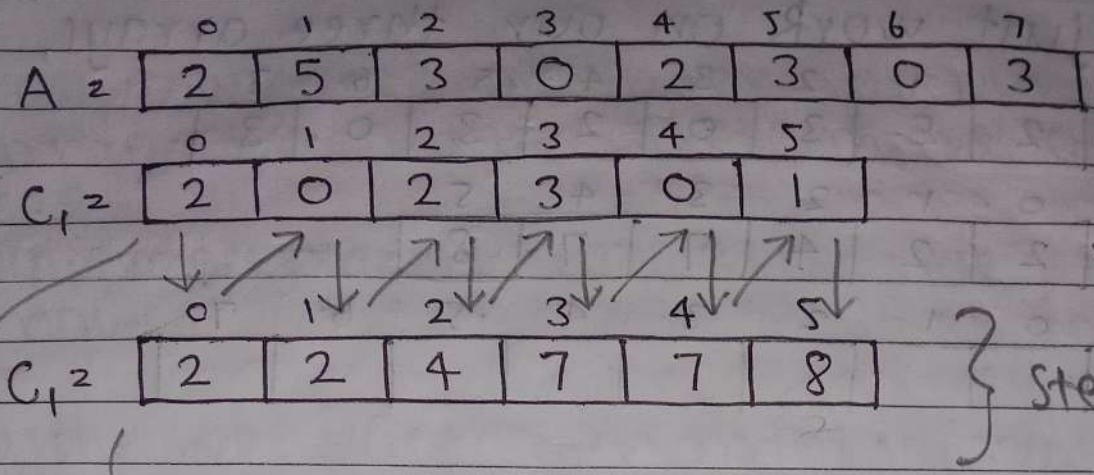
i) counting sort
ii) Bucket sort
iii) Radix Sort

* Counting sort

Date.....

Counting sort assumes that each of the 'n' input elements is an integer in the range 0 to k, for some integer k.
Counting sort determines, for each input element x, the number of elements less than x. It uses this information to place element x directly into its position in the output array.
For example, if 17 elements are less than x, then x belongs in output position 18.
We must modify the scheme slightly to handle the situation in which several elements have the same value, since we do not want to put them all in the same position.
In the code for counting sort, we assume that the input is an array A[1,---,n] and thus A.length = n. We require two other arrays B[1,---,n] holds the sorted output, and the array C[0,---,k] provides temporary work space.

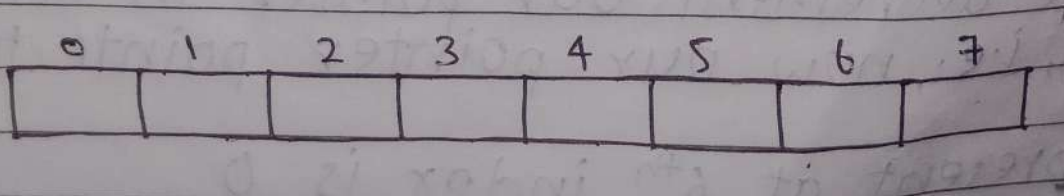
Date.....



→ har element me uske pehle wale element Ka count add krdiya.

→ this array stores the count frequency of each elements like 0 comes 2 time in our original array so we write 2 in the 2nd index, 3 comes 3 times in the original array, so we write 3 in the 3rd index and so on.

Now, we create the new updated array at step 2 because we want to calculate the index of the elements in the sorted array, & the index in the sorted array depends on the frequency of previous elements, and this array denotes the position of every element in the final sorted array.



Date.....

Now, we just work on our three arrays,

	0	1	2	3	4	5	6	7
A =	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C_1 =	2	2	4	7	7	8

	0	1	2	3	4	5	6	7
B =								

Now, we start traversing our original array from last index, i.e., 7th index. ↑ value
First we check the value at 7th index, i.e., 3, in our original array. index

Now, we check the 3rd index of our count (C_1) array, the value present at 3rd index is 7, in our C_1 array.
Now, we decrement the 7 by one, i.e., 7 becomes 6 now.

Atlast, we place our original value, i.e., 3 at 6th index, in our new array, i.e, B[] array.

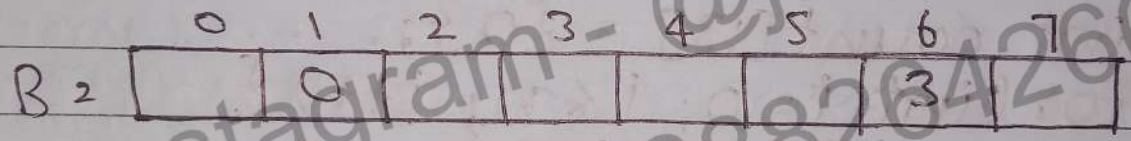
	0	1	2	3	4	5	6	7
B =							3	

Now, we decrement our pointer in original array, i.e., now our pointer points to 6th index.
value present at 6th index is 0.

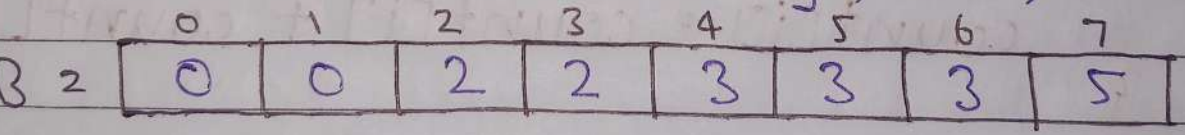
Date.....

Now, we check 0th index of our count array, the value present at 0th index is 2.
Now, decrement the 2 by one, it becomes 1 now.

Lastly we place our original value, i.e., 0 to the 1st index in new array, i.e., B[] array.



This procedure will continue till we trave -rse our original array & in the last we get our sorted array, i.e.,



Counting sort is a non-comparison based sorting technique, here we don't compare two algorithms / elements & swap with each other and all other.

Date.....

code :-

```
#include<bits/stdc++.h>
using namespace std;

void countSort(int arr[], int n) {
    int k = arr[0];
    for(int i = 0; i < n; i++){
        k = max(k, arr[i]);
    }

    int count[13] = {0};
    for(int i = 0; i < n; i++) {
        count[arr[i]]++;
    }

    for(int i = 1; i <= k; i++) {
        count[i] = count[i] + count[i-1];
    }

    int output[n];
    for(int i = n-1; i >= 0; i--) {
        output[--count[arr[i]]] = arr[i];
    }

    for(int i = 0; i < n; i++) {
        arr[i] = output[i];
    }
}
```

Find the max in the array

store the count of elements in the count array.

update the count array, step 2.

decrement of count array element value & store original element.

update our original array, & in this all the values are sorted.

Date.....

```
int main(){
   int arr[] = {9, 7, 4, 12, 2, 1};
   int n = 6;
   cout << "\n Given Array : ";
   for(int i = 0; i < n; i++){
      cout << arr[i] << " ";
   }
   countSort(arr, n);
   cout << "\n Sorted Array : ";
   for(int i = 0; i < n; i++){
      cout << arr[i] << " ";
   }
   cout << endl;
   return 0;
}
```

* Bucket Sort

Date.....

Like Counting Sort, Bucket Sort also imposes restrictions on the input to improve the performance.
Basically, Bucket sort works on decimal numbers.
It separate the elements into multiple groups said to be buckets. Elements in bucket sort are first uniformly divided into groups called Buckets & then they are sorted by any other sorting algorithm (insertion sort is preferred). After that elements are gathered in a sorted manner.
The basic procedure of performing the Bucket sort is :-

i) First, partition the range into a fixed number of Buckets.
ii) Then, toss every element into its appropriate Bucket.
iii) After that, sort each Bucket individu -ally by applying insertion sort.
iv) At last, concatenate all the sorted Buckets.

Date.....

Ques Sort the elements using Bucket sort,

0.35, 0.01, 0.99, 0.39, 0.05, 0.45, 0.23, 0.40, 0.42, 0.21.

After decimal we can have the number between 0 to 9. So we create a bucket from 0-9.

1st,	0
2nd,	1
3rd,	2
4th,	3
5th,	4
6th,	5
7th,	6
8th,	7
9th,	8
10th,	9

After creating the bucket, we check the element's. we insert the elements in the bucket in the form of list & according to the digit which comes after the decimal. Means, the first element is 0.35, so we insert the element in 4th index, i.e., 3rd Bucket

0	
1	
2	
3	→ 0.35
4	
5	
6	
7	
8	
9	

Like this, we do all element's check & put them in the bucket.

Date.....

0	→	0.01	→	0.05
1				
2	→	0.23	→	0.21
3	→	0.35	→	0.39
4	→	0.45	→	0.40 → 0.42
5				
6				
7				
8				
9	→	0.99		

After inserting all the elements in the bucket, we observed that all the elements are still not sorted. So, we apply insertion sort on every bucket.

After applying insertion sort, out bucket will look like this :-

0	→	0.01	→	0.05
1				
2	→	0.21	→	0.23
3	→	0.35	→	0.39
4	→	0.40	→	0.42 → 0.45
5				
6				
7				
8				
9	→	0.99		

Date.....

Now, we will concatenate all the elements from left to right in the bucket. After concatenating, we will get an sorted array of given integers:

0.01, 0.05, 0.21, 0.23, 0.35, 0.39, 0.40, 0.42, 0.45, 0.99

* Radix Sort

Date.....

Radix Sort is the linear sorting algorithm that is used for integers. In Radix Sort, there is digit by digit sorting is performed that is started from the least significant digit to the most significant digit.

First sort the elements based on the last digit (the least significant digit). These results are again sorted by second digit (the next to least significant digit). Continue this process for all digit until we reach the most significant digits.

we use some stable sort to sort them by last digit. Then stable sort them by the second least significant digit, then by the third, etc.

Algorithm:

i) Take the least significant digit of each element.
ii) Sort the list of elements based on that digit, but keep the order of elements with the same digit (this is the definition of a stable sort).
iii) Repeat the sort with each more significant digit.

Date.....

For example :-

181, 289, 390, 121, 145, 736, 514, 212

First we find the largest element in the array, i.e., 736, it has 3 digits. So, our loop will run up to three times, i.e., from ones place to hundreds place.

Now, we first sort the array elements on the basis of unit place digit, i.e., x = 0. So, we use counting sort to sort the elements.

1	8	1
2	8	9
3	9	0
1	2	1
1	4	5
7	3	6
5	1	4
2	1	2

hundred place ↙ ↓ tens place ↳ ones place

(First Pass)
First we sort the array based on one's place,

3	9	0
1	8	1
1	2	1
2	1	2
5	1	4
1	4	5
7	3	6
2	8	9

hundred place ↙ ↓ tens place ↳ ones place

Date.....

Now, one's place is sorted & now we move on to the ten's place & sort them.

3	9	0
1	8	1
1	2	1
2	1	2
5	1	4
1	4	5
7	3	6
2	8	9

hundred place, tens, ones

(Second Pass).
Now, we sort the array on the basis of ten's place,

2	1	2
5	1	4
1	2	1
7	3	6
1	4	5
1	8	1
2	8	9
3	9	0

hundred place, tens place, ones place

Date.....

Now, tens place is sorted & we move on to the hundred's place, & sort them

2	1	2
5	1	4
1	2	1
7	3	6
1	4	5
1	8	1
2	8	9
3	9	0

hundreds place, tens place, ones place

(Third Pass).

Now, we sort the array according to the hundred's place,

1	2	1
1	4	5
1	8	1
2	1	2
2	8	9
3	9	0
5	1	4
7	3	6

hundred place, tens place, ones place

After the third pass, our array is finally sorted, in ascending order, i.e.,

121, 145, 181, 212, 289, 390, 514, 736

Date.....

Implementation of Radix sort :-

```
int getMax (int list[], int n){
    int mx = list[0];
    int i;
    for (i = 1; i < n; i++){
        if (list[i] > mx)
            mx = list[i];
    }
    return mx;
}

void radixSort (int list[], int n){
    int m = getMax(list, n);
    int exp;
    for (exp = 1; m / exp > 0; exp *= 10)
        countSort (list, n, exp);
}
```