**Dr. Ohidujjaman Tuhin**
Assistant Professor
Dept. of CSE , UIU

# Introduction to Counting Sort:

1) **Counting Sort** is a **non-comparison-based** sorting algorithm.
2) It is particularly efficient when the range of input values is small compared to the number of elements to be sorted.
3) The basic idea behind **Counting Sort** is to count the **frequency** of each distinct element in the input array and use that information to place the elements in their correct sorted positions.

**Counting sort working principle:**

Let us consider n=9 elements in array A[n+1].

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 3 | 0 | 6 | 1 | 2 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

**Largest value finding from array A[ ]:**

| | | |
|---|---|---|
| k=0; | | int k=0; |
| when p=0 | if(k<A[p]) | for p=0 to n-1 |
| | True; k=A[p]; k=1 | if(k<A[p]) |
| when p=1 | True; k=A[p]; k=2 | k=A[p]; |
| when p=2 | True; k=A[p]; k=3 | end if |
| when p=3 | False; X ; k=3 | end for loop |
| when p=4 | False; X ; k=3 | |
| when p=5 | True; k=A[p]; k=6 | |
| when p=6 | False; X ; k=6 | |
| when p=7 | False; X ; k=6 | |
| when p=8 | False; X ; k=6 | |

Elements in array A[] range is 0 to 6. Therefore, the range array (count [k+1] = {0} ;) ranges from 0 to 6 as follows:

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

**Counting element frequency of A[n+1]:**

Array A[ n+1].

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 3 | 0 | 6 | 1 | 2 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Array count[k+1]

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

| | | |
|---|---|---|
| When i=0 | count[A[i]]= count[A[i]]+1;<br>count[1]=0+1<br>0 1 0 0 0 0 0 | for i=0 to n-1<br>count[A[i]]= count[A[i]]+1;<br>end for loop |
| When i=1 | count[2]=0+1<br>0 1 1 0 0 0 0 | |
| When i=2 | count[3]=0+1<br>0 1 1 1 0 0 0 | |
| When i=3 | count[3]=1+1<br>0 1 1 2 0 0 0 | |
| When i=4 | count[0]=0+1<br>1 1 1 2 0 0 0 | |
| When i=5 | count[6]=0+1<br>1 1 1 2 0 0 1 | |
| When i=6 | count[1]=1+1<br>1 2 1 2 0 0 1 | |
| When i=7 | count[2]=1+1<br>1 2 2 2 0 0 1 | |
| When i=8 | count[0]=1+1<br>2 2 2 2 0 0 1 | |

**Addition operation of index value of count[k+1] array:**

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 2 | 2 | 2 | 0 | 0 | 1 |

0 1 2 3 4 5 6

| | | |
|---|---|---|
| When i=1 | count[i]= count[i-1]+ count[i];<br>count[1]=2+2<br>\| 2 \| 4 \| 2 \| 2 \| 0 \| 0 \| 1 \| | for i=1 to k<br>count[i]= count[i-1]+ count[i];<br>end for loop |
| When i=2 | count[2]=4+2<br>\| 2 \| 4 \| 6 \| 2 \| 0 \| 0 \| 1 \| | |
| When i=3 | count[3]=6+2<br>\| 2 \| 4 \| 6 \| 8 \| 0 \| 0 \| 1 \| | |
| When i=4 | count[4]=8+0<br>\| 2 \| 4 \| 6 \| 8 \| 8 \| 0 \| 1 \| | |
| When i=5 | count[5]=8+ 0<br>\| 2 \| 4 \| 6 \| 8 \| 8 \| 8 \| 1 \| | |
| When i=6 | count[6]=8+1<br>\| 2 \| 4 \| 6 \| 8 \| 8 \| 8 \| 9 \| | |

**Element sorting in output[n+1] array:**

for i=n-1 to 0
 output[--count[A[i]]]=A[i];
end for loop

A[n+1]

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 3 | 0 | 6 | 1 | 2 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

count[k+1]

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| ~~2~~<br>~~1~~<br>0 | ~~4~~<br>~~3~~<br>2 | ~~6~~<br>~~5~~<br>4 | ~~8~~<br>~~7~~<br>6 | 8 | 8 | ~~9~~<br>8 |

output[n+1]

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 2 | 2 | 3 | 3 | 6 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

| | |
|---|---|
| when i=n-1=8 | **output[--count[A[i]]]=A[i];**<br>output[--count[0]]=0;<br>output[1]=0; |
| when i=7 | output[--count[2]]=2;<br>output[5]=2; |
| when i=6 | output[--count[1]]=1;<br>output[3]=1; |
| when i=5 | output[--count[6]]=6;<br>output[8]=6; |
| when i=4 | output[--count[0]]=0;<br>output[0]=0; |
| when i=3 | output[--count[3]]=3;<br>output[7]=3; |
| when i=2 | output[--count[3]]=3;<br>output[6]=3; |
| when i=1 | output[--count[2]]=2;<br>output[4]=2; |
| when i=0 | output[--count[1]]=1;<br>output[2]=1; |

**Counting sort pseudo code:**

```
for   i=0 to n-1                         // time= n  unit ; n represents the number of elements in the input array
       count[A[i]]= count[A[i]]+1;

end for loop
for i=1  to k                            // time= k  unit; and k represents the range of input.
      count[i]= count[i-1]+ count[i];
end for loop
for i=n-1 to 0                           // time= n  unit ;
     output[--count[A[i]]]=A[i];
 end for loop
 for i=0 to n-1                          // time= n  unit ;
      printf("  %d",output[i]);
  end for loop
```

**Complexity analysis:**

Time=n+k+n+n
=3n+k
=O(n+k)

## Best Case

The **best case** scenario for Counting Sort would be to have the range k just a fraction of n, let's say k(n)=0.1·n. As an example of this, for 100 values, the range would be from 0 to 10, or for 1000 values the range would be from 0 to 100.

In this case we get time complexity O(n+k)=O(n+0.1·n)=O(1.1·n) which is simplified to O(n).

When all items are in the same range, or when k is equal to 1, the best case time complexity occurs.
In this scenario, counting the occurrences of each element in the input range takes constant time, and finding the correct index value of each element in the sorted output array takes n time, resulting in total time complexity of **O(1 + n),** i.e. O(n), which is linear.

## Worst Case

The **worst case** however would be if the range is a lot larger than the input.

Let's say for an input of just 10 values the range is between 0 and 100, or similarly, for an input of 1000 values, the range is between 0 and 1000000.

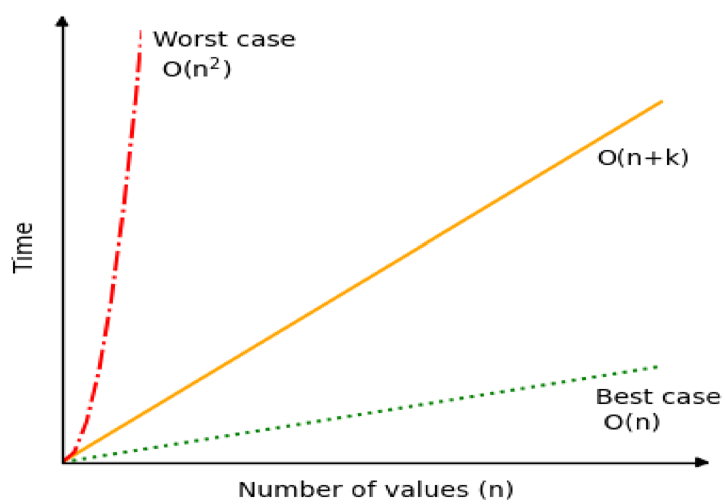
In such a scenario, the growth of k is quadratic with respect to n, like this: $k(n)=n^2$, and we get time complexity $O(n+k)=O(n+n^2)$ which is simplified to $O(n^2)$.

The worst-case scenario for temporal complexity is skewed data, meaning that the largest element is much larger than the other elements. This broadens the range of K.

Because the algorithm's time complexity is O(n+k), when k is of the order O(n^2), the time complexity becomes O(n+(n2)), which essentially lowers to O(n^2 ). Where k is of the order O(n^3), the time complexity becomes O(n+(n^3)), which essentially lowers to O(n^3). As a result, the time complexity increased in this scenario, making it O(k) for such big values of k. And that's not the end of it. For larger values of k, things can get significantly worse.
As a result, the worst-case time complexity occurs when the range k of the counting sort is large.

## Average Case

To calculate the average case time complexity, fix N and take various values of k from 1 to infinity; in this scenario, k computes to (k+1/2), and the average case is N+(K+1)/2. Similarly, varying N reveals that both N and K are equally dominating, resulting in O(N+K) as the average case.




**Counting sort:**

```
// Counting sort
// Online C compiler to run C program online
#include <stdio.h>
#include <stdlib.h>
int main() {
  int n,k;
  printf(" Enter the number of items:=");
  scanf(" %d",&n);
  int A[n];
  int output[n];
```

```
  printf(" Enter the items:=");
  for(int i=0;i<n;i++)  {
      scanf(" %d",&A[i]);
    }
  int large=0;
   for(int p=0;p<n;p++){
 if(large<A[p])
    large=A[p];
 }
   int m=large+1;    int count[m];
      for(int i=0;i<m;i++)    {       count[i]= 0;
                                              }
     printf("\n");
     printf("\n Count frquency:=");
   for(int i=0;i<n;i++)    {
          count[A[i]]= count[A[i]]+1;
                                     }
       for(int i=0;i<m;i++){
                      printf(" %d",count[i]);
                           }
   printf("\n");
    for(k=1;k<m;k++){
                    count[k]=count[k]+count[k-1];
                      printf(" %d",count[k-1]);
            }
               printf(" %d",count[k-1]);
    printf("\n");
    for( int i=n-1;i>=0;i--){
             output[--count[A[i]]]=A[i];
    }
     printf("\n");
     for(int i=0;i<n;i++){
          printf("  %d",output[i]);
     }
       return 0;
}
```