→ Sort an Array with (Two Type) of Element :

① Positive & Negative numbers
I/p → {15, -3, -2, 16}
o/p → {-3, -2, 15, 16} order of +ve & -ve no. not matter

② Even & Odd Numbers
I/p → {15, 14, 13, 12}
o/p → {14, 12, 15, 13}

③ Sort a Binary Array
I/p → {0, 1, 1, 0}
o/p → {0, 0, 1, 1}

→ Naive Approach use temp[] array
→ Efficient : use either LP or HP

```
void f(int a[], int n)
{
    int i=-1, j=n;
    while(1)
    {
        do{ i++; } while(a[i]<0);
        do{ j--; } while(a[j]>=0);

        if(i>=j) return;
        swap(a[i], a[j]);
    }
}
```

→ Sort an Array with 3 type of element:
(the Dutch National Flag Algo)

① Sort an array of 0s, 1s & 2s
I/p → {0,1,0,0,1,1,2,0,2}
o/p → {0,0,0,0,1,1,1,2,2}

② Three way partitioning
I/p → {2,1,2,20,10,20,1} pivot = 2 (2 के left में छोटे 2 के right में बड़े)
o/p → {1,1,2,2,20,10,20}

③ Partition Around a Range
I/p {10,5,6,3,20,9,40} range - [5,10]
o/p {3,5,6,9,10,20,40}

VIII Merge Overlapping Intervals :

```
struct intervals{
    int x,y;
};
int comp(xyz a, xyz b)
{ return a.x < b.x; }
int main()
{   int n;
    cin>>n;
    {struct intervals a[n];
    for(int i=0; i<n; i++)
        cin>> a[i].x >> a[i].y;
    f(a,n);
}
void f(intervals a[], int n)
{       struct intervals a[]

    sort(a, a+n, comp);
    int ay=0;
```

See properly the implementation

```
for(int i=1; i<n; i++)
{
    if( a[ns].y >= a[i].x)
    {
        a[ns].x = min( a[ns].x, a[i].x);
        a[ns].y = max(a[ns].y, a[ns].y);
    }
    else
    {   ns++;
        a[ns] = a[i];
    }
}
for(int i=0; i<=ns; i++)
    cout<<a[i].x<<" "<<a[i].y<<endl;
}
```

Inp = {{5,10}, {3,15}, {18,30}, {2,7}}

↓ sort

~~{{2,7}, {5,10}, }~~

{{2,7}, {3,15}, {5,10}, {18,30}}

ns ↓

{{2,15}, {3,15}, {5,10}, {18,30}}

ns

↓

{{2,15}, {18,30}, {5,10}, {18,30}}

ns

output

no use

- doing more no. writes reduces the life of the memory.
- memory write EEPROM

→ Cycle Sort :

- $O(n^2)$ worst case Algo
- It does minimum memory writes and can be useful for cases where memory write is costly.
- In-place & not stable

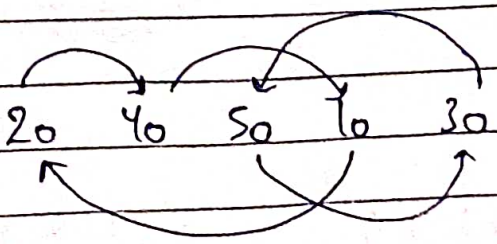
** Useful to solve questions like find minimum swaps required to sort an array.

** Used when saying elements are in the range $[1, n]$

```
void cycleSort(int a[], int n)
{
    for(int cs=0; cs<n-1; cs++)
    {   int item = a[cs];
        int pos = cs;
        for(int i=cs+1; i<n; i++)
            if (a[i] < item)
                pos++;
        swap(item, a[pos]);
        while(pos != cs)
        {
            pos = cs;
            for(int i = cs+1; i<n; i++)
                if (a[i] < item)
                    pos++;
            swap(item, a[pos]);
        }
    }
}
```

fix the first element of cycle

fix remaining element of cycle



2 cycles 20-40-10 & 50-30

Question on Cycle Sort :

① **Missing Number**

→ Given array of n distinct numbers range [0,n] return MN

I/p → [3,0,1] o/p → 2

```
int f(vector<int> &v)
{   int i=0;
    while(i<v.size())
    {
        if(v[i]==v.size()){i++; continue;}
        if(i!=v[i])
            swap(v[i], v[v[i]])
        else
            i++;
    }
    for(int i=0; i<n; i++)
    {   if(v[i]!=i)
            return i;
    }
    return v.size();
}
```

I/p → 3,0,1,4

range [0,n] → [0,4] वाले # 0 index पे 0 आना चाहिए
1 —— 1 ——
2 —— 2 ——
so on

4,0,1,3
i→i

4,0,1,3
i

0,4,1,3
i→i

0,4,1,3
i

0,4,1,3 → 0,1,4,3 → 0,1,4,3 → while ends

final array
0,1,4,3
i

not equal
ans i

Given array of n integers ... mean ... 3 no. missing ... 3 no. ...

② Find all numbers Disappeared in an Array :

→ Given an array of n integers range [1,n].

```
vector<int> f(vector<int> &v)
{   int i=0;
    vector<int> v1;
    while(i<v.size())
    {
        if(v[i]!=v[v[i]-1])
            swap(v[i], v[v[i]-1]);
        else
            i++;
    }
    for(int i=0;i<v.size();i++)
        if(i!=v[i]-1)
            v1.push_back(i+1);
    return v1;
}
```

I/p → [4,3,2,7,8,2,3,1]

O/p → [5,6]

0 index ↑ 1 माना पड़ेगा

1 — 2 —

2 — 3 —

so on

n=5

1 3 5 3 5

↓↓

1 (5) 3 (3) 5

0 1 2 3 4

i ! v[i]-1

③ Duplicate numbers (one repeating element find it)

→ Array of (n+1) integers range [1,n]

```
int f(vector<int> v)
{   int i=0; vector<int>
    while(i<v.size())
    {   if(v[i]!=v[v[i]-1])
            swap(v[i], v[v[i]-1]);
        else
        {   if(i!=v[i]-1)
                return v[i];
            i++;
        }
    }
    return 1; // no use
}
```

I/p → 1 2 3 4 2 2

↓↓

1 2 3 4 (2)

0 1 2 3 4

i ! v[i]-1

④ Find all Duplicates in Array:

→ Array says [1, n] each element appears once or twice, return twice occuring element

```
vector<int> findDuplicates(vector<int>& v)
{   int i=0;
    vector<int> v1;
    while(i<v.size())
    {
        if(v[i]!=v[v[i]-1])
            swap(v[i], v[v[i]-1]);
        else
            i++;
    }
    for(int i=0; i<v.size(); i++)
    {
        if( i!=v[i]-1)
            v1.push_back(v[i]);
    }
    return v1;
}
```

I/p: 3 4 1 3 4 2

↓

1 4 3 3 4 2

↓

1 3 3 4 4 2

↓

1 2 3 4 4 3

0 1 2 3 4 5

i != v[i]-1

⑤ Missing & Repeating element: (Set Mismatch)

→ Array says [1, n] one no. is missing & one repeating

→ same

```
for( int i=0; i<v.size(); i++)
{if(i!=v[i]-1){v1.push_back(v[i]);
    v1.push_back(i+1);}
}
return v1;
}
```

3 4 2 5 3

↓

2 4 3 5 3

↓

4 2 3 5 3

↓

5 2 3 4 3

↓

3 2 3 4 5

0 1 2 3 4

3 RN

0+1=1 → MN

(6) Find first missing Positive number:

→ Given unsorted array n, return the smallest +ve integer missing.

~~ip~~ip→ [7,8,9,11,12]	[3 4 -1 1]	[3 -4 -2 0 1 2]
op→ 1	2	4

```
int f(vector<int>&v)
{   int i=0;
    while(i<v.size())
    {
        if(v[i]<=0 || v[i]>v.size()){i++; continue;}
        if(v[i]!=v[v[i]-1])
            swap(v[i], v[abs(v[i])-1]);      // no use abs()
        else
            i++;
    }
    for(int i=0; i<v.size(); i++)
        if(i+1!=v[i])
            return i+1;
    return v.size()+1;
}
```

0 index + 1 वाला रूट
1 —— 2 ——
so on

eg: [3 -4 -2 0 1 2]

↓

-2 -4 3 0 1 2

↓

1 -4 3 0 -2 2

↓

1	2	3	0	-2	-4
0	1	2	3	4	5

i+1 missing

first missing +ve number

<=0 numbers & no. greater than v.size() are ignored

eg: [3 4 -1 1]

↓

-1 4 3 1

↓

1	4	3	-1
0	1	2	3

2 missing

nol for $k = n^2, n^3, n \log n$

→ Time is k is linearly proportional to n $k = n, n/2, \ldots 2n$

Counting Sort:

- ~~O(n) complexity~~, when input range is small.
- O(n+k) time to sort n elements in range 0 to k-1.

Naive

```
void f(int a[], int n, int k)        → max element in array + 1
{   int count[k];
    for(int i=0; i<k; i++)            } O(k)
    {   count[i]=0;
    for(int i=0; i<n; i++)            } O(n)
        count[a[i]]++;

    int index=0;
    for(int i=0; i<k; i++)                                  } O(n+k)
    {
        for(int j=0; j< count[i]; j++)   } loop in total n times
        {   a[index]=i;   // here assigning index
            index++;
        }
    }
```

- Simply counting a[i] & printing it. starting from 0

$T(n) = O(k) + O(n) + O(n+k) = O(n+k)$

- Naive approach can't be used for sorting objects with multiple members, like sorting an array of students by marks.
- It take cares of object when you want to sort them according to keys b/z We are never putting indexes as our values in output array we are putting actual values in the output[] array, then copying these values to back to a[].

2nd approach bright

We use this also in CS

```
void f(int a[], int n, int k)
{   int count[k];
    for(int i=0; i<k; i++)
        count[i]=0;
    for(int i=0; i<n; i++)
        count[a[i]]++;

    for(int i=1; i<k; i++)
        count[i] = count[i-1] + count[i];
    int output[n];
    for(int i=n-1; i>=0; i--)
    {
        output[count[a[i]]-1] = a[i];
        count[a[i]]--;
    }
    for(int i=0; i<n; i++)
        a[i] = output[i];
}
```

count[i] में assign करेंगे <= element i element

copying back to a[]

- We are traversing backward to make the algo stable.

Ex- a[] = [1, 4, 4, 1, 0, 1] → count[] = {1, 3, 0, 0, 2} k=5

count[] = {1, 4, 4, 4, 6} → there are 4 elements <= 2 in a[]

output[] = {0, 1, 1, 1, 4, 4}

for input → 0 1 4 4 1 0

in a[] from end → 0 1 4 4 1 0 , 1 index 4 में in count[] array value 4 है इसलिए 4 element <=1 है , output[] array में 4-1=3 index 4 assign कर दिए और count[a[i]]-- कर दिए

- Not a comparision based Algorithm
- $O(n+k)$ TC | $O(n+k)$ Auxillary Space
  - count[k]
  - output[n]
- Stable
- Used as a subroutine in Radix Sort

CS not work for $[0, n^2)$ or $[0, n^3]$

Radix Sort:

- is a linear time Algo
- RS is good for larger ranges of k
- not a comparision based Algo. $k = [0, n^2)$ or $[0, n^3]$ or ...

```
void rs(int a[], int n)
{   int mx = a[0];
    for(int i=1; i<n; i++)
        if(a[i] > mx)
            mx = a[i];                    // find max in array

    for(int exp=1; mx/exp>0; exp = exp*10)
        countySort(a, n, exp);
}
void countySort(int a[], int n, int exp)
{
    int count[10], output[n];
    for(int i=0; i<10; i++) count[i] = 0;          // int count[10]={0};
    for(int i=0; i<n; i++) count[(a[i]/exp)%10]++;
    for(int i=1; i<10; i++) count[i] += count[i-1];
    for(int i=n-1; i>=0; i--)
    {
        output[count[(a[i]/exp)%10]-1] = a[i];
        count[(a[i]/exp)%10]--;
    }
    for(int i=0; i<n; i++) a[i] = output[i];
}
```

no. of digit in max no.

TC → $O(d*(n+b))$ → here b=10 | SC → $O(n+b)$ → b is 10

in counting sort it was $O(n+k)$ → k is b in RS

Eg: 319 212 6 8 100 50

Rewrite no. with leading zeros

319 212 006 008 100 050

Stable Sort according to last digit (least significant digit)

100 050 212 006 008 319

Stable Sort ——— middle digit

100 006 008 212 319 050

Stable Sort ——— MSB

006 008 050 100 212 319

→ Bucket Sort: → Good for floating numbers sort

- CS will not work for 2nd situation b/z nos are floating point
- For first situation also we need lot of extra space.
- We can use normal sort() algo which give TC - O(nlogn)

Situation ① Consider a situation where we have nos uniformly distributed in range 1 to 10^8. How do we sort effectively? Ans - BS

Situation ② Consider a situation where we have floating point nos uniformly distributed in range from 0.0 to 1.0 (1 not included)

Size of array

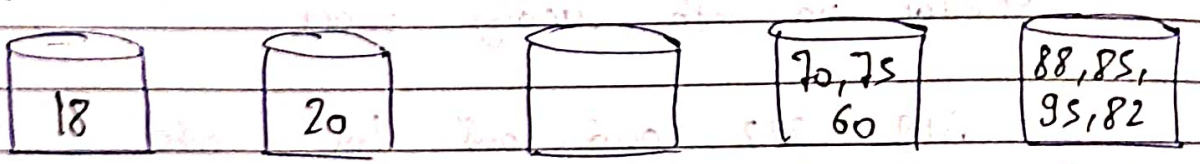
Uniform Distribution : n=100

If we divide this n into 10 equal Bucket, then around 10 number will be in each Bucket

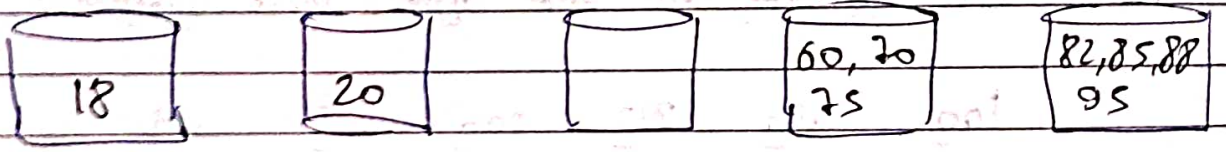
Eg: I/p 20 88 70 85 75 95 18 82 60

Range 0 to 99 (n=100)

Step 1 Scatter



Step 2 Sort Bucket



Step 3 Join Sorted Bucket

18, 20, 60, 70, 75, 82, 85, 88, 95

→ Joining takes linear time

** We can use any Standard Sorting Algo to sort individual Bucket, since we know there will be less element in Bucket we can use Insertion Sort.

** n no element, there are k → buckets

- On average n/k element in each bucket
- If k is close to n then we have approx one element in each bucket, we takes O(1) time to sort bucket.
- if k is $n/2$, $n/3$, $n/4$, $n/5$ we have 2 or 3 element approx in each bucket which is constant time for sort bucket.

** Since we have uniform distribution we taken O(1) time to sort individual bucket

** No. of Bucket ∝ no. of element (k ∝ n)

→ We are given bucket (k) as input

*** → When a[] is float type {0.98, 0.43, 0.12, 0.65, 0.12} range [0.0, 1.0) use,
• vector<float> bkt[k];
• I/p array ⇒ float a[];
max value++;

```
void f(int a[], int n, int k)
{
    int max_val = *max_element(a, a+n);
    max_val++;
    vector<int> bkt[k];          // Array of vectors
    for(int i=0; i<n; i++)
    {
        int bi = (k * a[i]) / max_val;
        bkt[bi].push_back(a[i]);
    }                                  → filling bucket bkt[k] of size k.
O(n) { for(int i=0; i<k; i++)
        sort(bkt[i].begin(), bkt[i].end());   → sort each bkt
    int index=0;
O(n) { for(int i=0; i<k; i++)
        for(int j=0; j<bkt[i].size(); j++)
            a[index++] = bkt[i][j];           → Join bucket
}
```

• max_val++; b/z when we divide max element of a[] by max_val it gets out of bound of a bkt[k];

Time Complexity (Assuming k=n)
① Best Case: When data is uniformly distributed
O(n)
② Worst Case: when all items goes into single bucket
- If we use insertion sort to sort the individual buckets, it takes O(n²)
BS ← - if we O(nlogn) also to sort ——
O(nlogn)

0 1 2 3 4 5 6

eg: {30, 40, 10, 80, 5, 12, 70}

K=4

0	10	5	12
1	30	40	
2			
3	80	70	

← bkt[k]

max_val = 81

i=0, bi = 4x30/81 = 1 ⇒ bkt[1] = 30

i=1, bi = 4x40/81 = 1 ⇒ bkt[1] = 40

i=2, bi = 4x10/81 = 0 ⇒ bkt[0] = 10

i=3, bi = 4x80/81 = 3 ⇒ bkt[3] = 80

i=4, bi = 4x5/81 = 0 ⇒ bkt[0] = 5

i=5, bi = 4x12/81 = 0 ⇒ bkt[0] = 12

i=6, bi = 4x70/81 = 3 ⇒ bkt[3] = 70

Sort each bkt[i]

0	10	5	12
1	30	40	
2			
3	80	70	

5	10	12
30	40	
70	80	

V.imp If array of float type eg: a[] = {9.8, 0.6, 10.1, 1.9, 3.02, 3.04, 5.0, 4.8}

use the above algo, only change is vector<float> bkt[k]

range will be [0 - max_element]