[Array]

⇒ Array → DS → similar elements

max size of array → main − 10^6
→ global − 10^7

(1) Largest element in an array :-

Brute ✓
↓
Better ✗
↓
optimal ✓

arr[ ] = [3, 2, 1, 5, 2]
n=5

↓ sort

1 2 2 3 5

↑ Largest

print (arr[n-1])

TC → O(N log N)

SC → O(1)

⇒ optimal sol^n :-

arr[ ] = [3, 2, 1, 5, 2]

```
lar = a[0]
for (i=0; i<n; i++)
    if(a[i] > lar) {
        lar = a[i];
    }
Print(lar);
```

TC → O(N)

(v) Second largest element in array :-

→ Brute force :-

arr[ ] = [1 2 4 7 7 5]

↓ (NlogN)

Sort

[1 2 4 5 7 7]

↑

Second largest

```
for (i=n-2 ; i>=0; i--) {
    if (arr[i] != largest) {
        secondl = arr[i];
        break;
    }
}
```

secondlargest = -1 (if SL doesn't exist)

O(N) (if arr is [1 7 7 7 7 7

TC :- NlogN + N

→ Better :-

first Pass

```
largest = arr[0]
for (i=0; i<n; i++)
{ if (arr[i] > largest)
    largest = arr[i];
}
```

Second Pass

```
slargest = -1
for (i=0; i<n; i++)
{ if (arr[i] > slargest &&
     arr[i] != largest) {
     slargest = arr[i];
  }
}
```

TC :- O(N+N) = O(2N)

→ Optimal :-

arr = [1 2 4 7 7 5]

```
lar = arr[0]     slar = -1

for (int i=0 ; i<n ; i++) {
    if (arr[i] > lar) {
        slar = lar ;
        lar = arr[i] ;
    }
    else if (arr[i] < lar && arr[i] > slar) {
        slar = arr[i];
    }
}
return slar;
}
```

$\boxed{TC :- O(N)}$

3) Check if the array is sorted :-

```
int isSorted (int n, vector<int> a) {
    for (int i = 1; i<n ; i++) {
        if (a[i] >= a[i-1]) {

        }
        else {
            return false;
        }
    }
    return True;
}
```

4) Remove duplicates in-place from sorted array :-

→ Brute :-

arr = [1, 1, 2, 2, 2, 3, 3]

1	2	3				
0	1	2	3	4	5	6

index index index index

set: 1, 2, 3

first pass

```
set<int> st;
for (i=0; i<n; i++)
    st.insert(arr[i]);
```

↑ N log N

N

```
index = 0
for (auto it : st) {
    arr[index] = it;
    index++;
}
```

TC :- O(N log N + N)

SC :- O(N)

→ optimal :-

arr[] = {1, 1, 2, 2, 2, 3, 3}

1	2	3
0	1	2

↑ ↑ i

```
i = 0
for (j=1; j<n; j++) {
    if (arr[j] != arr[i]) {
        arr[i+1] = arr[j];
        i++;
    }
}
return i+1;
```

TC :- O(N)

SC :- O(1)

Left rotate array by one place :-

arr[ ] = [1, 2, 3, 4, 5]

[2, 3, 4, 5, 1]

(i-1) (i-1) (i-1) (i-1) temp

```
temp = a[0]
for (i=1; i<n; i++){
    arr[i-1] = arr[i] }
arr[n-1] = temp;
```

TC :- O(N)

SC :- O(1)

Left rotate array by D place :-

→ Brute :-

temp = [D places] like (1, 2, 3) for d=3

hifting: for(i=d ; i<n; i++) → O(n-d)

a[i-d] = a[i]

ut back temp! for(i=n-d ; i<n; i++)

a[i] = temp[i-(n-d)]; → O(d)

$\boxed{d = d \% n}$

ep-1:- for (i=0; i<d; i++) → O(d)

temp.push_back (arr[i]);

TC :- O(d) + O(n-d) + O(d) = O(n+d)

SC :- O(d)

→ optimal :-

$$\begin{array}{cc} 3,2,1 & 7,6,5,4 \\ arr[] = [\boxed{1,2,3} \mid \boxed{4,5,6,7}] & d=3 \end{array}$$

[4, 5, 6, 7, 1, 2, 3]

→ reverse (a, a+d) ⟶ O(d)

→ reverse (a+d, a+n) ⟶ O(n−d)

→ reverse (a, a+n) ⟶ O(n)

TC :- O(2n)

SC :- O(1)

7) Move all zeroes to the end of the array :-

⟶ Brute force :-

$$\begin{array}{c} 1\ \ 2\ \ 3\ \ 2\ \ 4\ \ 5\ \ 1\ \ 0\ \ 0\ \ 0 \\ arr[] = \{\cancel{1}, \cancel{0}, \cancel{2}, \cancel{3}, \cancel{2}, \cancel{0}, \cancel{0}, \cancel{4}, \cancel{5}, \cancel{1}\} \end{array}$$

temp [] → {~~1~~, ~~2~~, ~~3~~, ~~2~~, ~~4~~, ~~5~~, ~~1~~}

Step-1 :- temp → [ ]

for(i=0 → n) → O(n)

if(arr[i] != 0)

temp.add (arr(i))

S2 :- for(i=0 ; i<temp.size(); i+

arr[i] = temp[i] →(

non zero no. → temp siz

Step3 :- for(i=temp.size() to i< arr.size)

arr[i] = 0

⟶ O(n−x)

⇒ TC :- O(N) + O(x) + O(N−x)

→ O(2N)

SC → O(n) → O(N)

Worst :- no zeroes in Entire array

→ optimal :-

$$arr[\,] = [\underbrace{1, 0}_{x}, \overbrace{\underset{\uparrow i}{2}, 3, 2, 0, 0, 4, 5, 1}^{n-x}]$$

Step-1:

```
j = -1
for(i=0; i<n; i++){
    if(arr[i]==0){        → O(x)
        j=i;
        break;
    }
}
```

Step-2:-

```
for(i=j+1; i<n; i++){
    if(arr[i]!=0){
        swap(arr[i], arr[j]);
        j++;
    }
}                         → O(n−x)
```

TC :- $O(x) + O(n-x)$

$\approx O(n)$

SC :- $O(1)$

1) Union of two Sorted arrays :-

arr1[ ] = {1, 1, 2, 3, 4, 5}

arr2[ ] = {2, 3, 4, 4, 5}

→ Brute force Approach :-

Merge + sort + Remove Duplicates

Time :- $O(m+n) \log(m+n)$

↳ Sorted ka fayda nhi uthaya

↳ Simple but inefficient for large data

→ Better Approach :-

using Set

Time : O(m+n) but insertion in set = O(log N)

↳ Sorted o/p milta hai but through set, not log

↳ Sorted & clean

→ Optimal Approach :-

Two pointer Technique

Time : O(m+n)   SC: O(m+n)

↳ Sorted ka pura fayda

↳ No extra space (excluding o/p)

↳ Best for already sorted arrays

Q) find missing number in array :-

→ Brute Approach :-

arr[] = {1, 2, 4, 5}   N=5

```
for(i=1 ; i<=N ; i++) {
    flag=0;
    for( int j=0; j<n-1; j++) {
        if(arr[j] == i) {
            flag=1;
            break;
        }
    }
}
if( flag ==0)
    return i;
}
```

TC : O(N×N)

SC : O(1)

→ Better Approach :-

arr[ ] = {1, 2, 4, 5}   N = 5   (Hashing)

Hash

0	~~0~~ 1	~~0~~	0	~~0~~ 1	~~0~~ 1
0	1	2	3	4	5

N—6

missing → 3

⇒ Pseudo code :-

```
hash [n+1] = {0};

for (i=0 to n)

    hash (arr[i]) = 1;
```

```
for (i=1 → n)
  if (hash[i] == 0)
    return i;
```

TC :- O(N) + O(N) ≃ O(2N)

SC :- O(N)

→ optimal Solution :-

arr[ ] = [1, 2, 4, 5]

$$Sum = \frac{N \times (N+1)}{2} \quad \text{for } N = 5$$

$$= 15$$

```
for (i=0 to N)
    S = S + arr[i]
return (sum - S)
```

TC :- O(N)

SC :- O(1)

OR

$\boxed{XOR}$ ⟶

XOR1 = 1^2^3^4^5

XOR2 = 1^2^4^5

XOR1 ^ XOR2

(1^1)^(2^2)^(3^3)^(4^4)^(5X5)

XOR1=0

for(i=1 → N)

    XOR1= XOR1^i

XOR2=0

for(i=0 to N-1)

    XOR2= XOR2^arr[i]

XOR1 ^ XOR2

TC :- O(2N)

SC :- O(1)

Replace that loop by
XOR1 = XOR1 ^ (i+1)
and also after loop add
XOR1 = XOR1 ^ N

TC:
SC:

⇒ benefit of XOR —

Let say the I/P size is 10^5

then In Sum — $\frac{10^5 \times (10^5+1)}{2} \approx \boxed{10^{10} > 10^5}$

but In XOR — It remains 10^5 slightly better in terms of data

10) Intersection of two sorted arrays :-

⇒ Brute Approach ~~force~~ :-

1) Nested loops → TC: $O(n^2)$ SC: O(1)

2) visited array —

arr1=[1, 2, 3, 3, 4, 5]
arr2=[2, 3, 3, 5, 6, 7]
vis = [~~0~~, ~~0~~, ~~0~~, ~~0~~, 0, 0]
       1   1   1   1

```
for(i=0 → n1)
    for(j=0 → n2)
        if((a[i]==b[j] && vis[j]==0) {
            ans.add(a[i]);
            vis[j]=1;
            break;
        }
    }
}
```

TC: $O(n_1 \times n_2)$

SC: $O(n_2)$

→ Better :- (hashing)

```
int hash[1001] = {0};
for (int i=0 to n1)
    hash[arr1[i]] += 1;

S2:- for (auto it : arr2) {
         if (hash[it] > 0) {
             inter.push_back(it);
             hash[it] = 0;
         }
     }
```

TC:- $O(n1+n2)$

SC :- $O(n)$ → for solving

but for return the array it is $O(n+x)$

→ optimal :- (Two-pointer approach)

```
        i
        ↓
arr[] = [1, 2, 2, 3, 3, 5]
arr[] = [2, 3, 4, 4, 5, 6]
        ↑
        j
```

```
while (i > n1 && j > n2) {
    if (arr[i] == arr[j]) {
        then push
        i++, j++;
    }
    else if (arr[i] > arr[j]) j++;
    else i++;
```

TC :- $O(n1+n2)$

SC :- $O(1)$ → for solving

for return the array is $O(x)$

11) Maximum Consecutive ones :-

arr [ ] = { 1, 1, 0, 0, $\underbrace{1, 1, 1}$, 0 }   Ans = 3

Approach :- 1) array maintain
2) simple variable

→ Brute :-

```
cnt = 0
vector ones;
for (i=0 to n) {
    if(arr[i] == 1)
        cnt += 1
    else {
        ones.add (cnt);
        cnt = 0;
    }
ones.add (cnt);
```

TC :- O(n)

SC :- O(n)

→ optimal :-

Let, var = 0 and assign maximum cnt in var

return var.

TC :- O(n)

SC :- O(1)

) find the no. that appears once, & others no's twice: 17

Approach:- 1) Hashing / unordered map

2) variable (sort & skip)

3) XOR

1) Brute :- (unordered map)

TC :- $O(n) + O(\frac{n}{2}+1)$ (1 pass for freq. + 1 pass for checking)

SC :- $O(n/2+1)$

2) Better :- (sorting + skipping)

TC :- $O(n\log n)$ (due to sorting)

SC :- $O(1)$ (sorting in-place)

3) optimal :- (XOR)

TC :- $O(n)$ (XOR all the elements)

SC :- $O(1)$

3) Longest Subarray with Sum K :- [positives]

arr[] = [1, 2, 3, 1, 1, 1, 1, 4, 2, 3] K=3

→ Brute :-

arr[] = [1, 2, 3, 1, 1, 1, 1, 4, 2, 3]

Generate all Subarray (i-j)

len = 0

⇒ for (i=0; i<n; i++){

for(j=i to n) {

TC : ≃ $O(n^3)$

for (k=i → j)
s+ = a[k]

replace by s+ = a[j]; → TC : ≃ $O(n^2)$

} if (s==k) len = max (len, j-i+1)

}

}

}

why:- [[2, 3], 1], 2]]

[2,3] [2,3,1] [2,3,1,

↓ ↓ ↓

5 5+1=6 6+2 =

→ Hashing :-

Note:- Better for positives

optimal for (pos + neg + zero)

arr[ ] = [1, 2, 3, 1, 1, 1, 1, 4, 2, 3]

prefsum = ~~0~~ ~~1~~ ~~3~~ ~~6~~ ~~7~~ ~~8~~ 9

till sum

7	3
6	2
3	1
1	0

Hash map

Concept:- $x-k = 4$ $k = 3$

7 → x

if (x-k) in Hash map

then calculate len

else

no.

for (pos + neg + zero) —

→ avoid add Duplicate sum in Hash map (sum should not in Hash

TC : O(N×log N) or O(N×1)

$N \times N \simeq N^2$

SC : O(N)

why extra condⁿ for pos + neg + zeros —

edge case - [2, 0, 0, $\boxed{3}$] → for eg. k=3

presum = ~~0~~ ~~2~~ 5   len = 0

'return len=1 ×

2→2
2→1
2→0

$\underbrace{2 \quad 3}_{5}$

→ optimal for positives (sliding window) :-

arr[ ] = [1, 2, 3, 1, 1, 1, 1, 3, 3]   k=6

Sum = ~~1~~ ~~3~~ ~~6~~ ~~7~~ ~~6~~ ~~7~~ ~~8~~ ~~6~~ ~~7~~ ~~8~~ ~~6~~ ~~7~~ ~~4~~ ~~7~~ ~~6~~ ~~9~~ ~~8~~ ~~7~~ 6   len = ~~0~~ ~~3~~ 4

→ as you move towards right shrink the array from left & reduce left from array if its (sum > k).

TC :- O(2N)

SC :- O(1)

i) Two Sum problem :-

→ Brute :-

logic: check every pair (i, j)

TC: $O(n^2)$

SC: O(1)

why extra condⁿ for pos + neg + zeros —

edge case - [2, 0, 0, $\boxed{3}$] → for eg. k=3

presum = ~~0~~ ~~2~~ 5 len = 0

2-2
2-1
2-0

return len=1 ×

$\underbrace{\overset{2}{\sqcap} \; \overset{3}{\sqcap}}_{5}$

→ optimal for positives (sliding window) :-

arr[ ] = [1, 2, 3, 1, 1, 1, 1, 3, 3] k=6

sum = ~~1~~ ~~3~~ ~~6~~ ~~7~~ ~~6~~ ~~7~~ ~~8~~ ~~6~~ ~~7~~ ~~8~~ ~~6~~ ~~7~~ ~~4~~ ~~7~~ ~~6~~ ~~9~~ ~~8~~ ~~7~~ 6 len = ~~0~~ ~~3~~ 4

→ as you move towards right shrink the array from left & reduce left from array if its (sum > k).

TC :- O(2N)

SC :- O(1)