STEP - 3

ARRAYS

EASY – largest element in array ✓ BRUTE FORCE

sort array using any sorting algo then pick last element

Time complexity will depend on sorting algo used

OR

Just use a max variable loop through the array.

Time complexity – $O(n)$ OPTIMAL

Space complexity –

Second largest without sorting

→ Sort the array and largest is at end (BRUTE FORCE)

↑ nlogn

1, 2, 4, 5, 7, 7 ← not second largest

→ ∴ run loop from end select no which is different from end.

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

← $O(n)$ at worst

1, 7, 7, 7, 7, 7, 7

→ { figure largest in first pair

→ figure second largest by comparing with largest and smallest no. }

BETTER

{

→ pick the max and secondMax from first two element of array

→ Traverse ~~loop~~ and array under following condition

```
if ( n > max ) {
        secondMax = max;
        max = n;
    }
else if (n < max && n > secondMax ) {
        secondMax = n;
    }
return secondMax;
```

OPTIMAL

O(n) - time complexity:

REMOVE DUPLICATE FROM SORTED ARRAY

→ Usage of set data structure as set do not have duplicates

BRUTE FORCE

→ Set<int> st

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

(n log n)

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

$O(n)$

∴ time complexity → $n\log n + n$

∴ space complexity – $O(n)$ → for set having all distinct element

→ arr[] = {1, 1, 2, 2, 2, 3, 3}

OPTIMAL

```
eliminateSorted(arr[]){
    int i=0;
    for(int j=0; j< arr.length; j++){
        if(arr[j] != arr[i]){
            arr[++i] = arr[j];
        }
    }
}
```

SHIFT ARRAY BY ONE PLACE

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

storing '1'

- temp = arr[0]

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

shifting by one

arr[n-1] = temp; placing at last

OPTIMAL

O(n) - time complexity
O(1) - stack space "extra space"
O(n) - included arr[] "in the algorithm" should include the input space also

SHIFT ARRAY BY D PLACE

arr = {1, 2, 3, 4, 5, 6, 7}

d = 7

now here total element = 7 ∴ after 7 rotation the array would become same as original one.

∴ if d = 9 only 2 rotation need
7 + 2
no effect

∴ d = d % n no 7 element

BRUTE

for rotating elements to ~~right~~ left;

→ arr[] = { 1, 2, 3, 4, 5, 6, 7 } $\boxed{d=3}$

temp[] = [1, 2, 3] now first D element in temp array

O(D)

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

O(n-D)

shifting remaining element from beginning to n-K elements

a[3-3] = a[3]
a[0] = a[3]
a[1] = a[4]
a[2] = a[5]
⋮
a[3] = a[6]

n-d n-d+1 n-d+2

arr[] = { 4, 5, 6, 7, 5, 6, 7 }

Now copy temp element in remaining spaces

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

step to get value from 0...

as 5 6 7

5 - 5 = 0
6 - 5 = 1
7 - 5 = 2

O(D)

∴ O(d) + O(n-d) + O(d) = O(n+d)

Extra space used = O(D) for temp

OPTIMAL

arr = { 1, 2, 3, 4, 5, 6, 7 } d = 3

3 2 1 7 6 5 4 three reversed

→ {4 5 6 7 1 2 3} result

∴ ~~→ reverse(a, a+~~

reverse(0, 0+d-1) → O(d)

reverse(d, n-1) → O(n-d)

reverse(0, n-1) → O(n)

∴ O(2n) → increased but not using extra space

∴ optimal in space complexity sense.

MOVE ALL ZEROS TO END OF ARRAY

BRUTE

→ Store all non zero no in temp array

→ then copy it back to the array and fill remaing by zero

```
for (i=0 → n) {
    if (arr[i] != 0) temp.add(arr[i]) → O(n)
}
```

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

[1, 0, 2, 3]

$\rightarrow O(n-n)$

$\therefore \; O(n-n) + O(n)$

$\rightarrow O(n)$ fayda hetle without modifying array.

UNION OF TWO SORTED ARRAYS

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

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

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

BRUTE

- Set data structure

1	2	3	4	5	

put back in union array

```
for (i=0; i<n; i++) s.insert(a[i]); → O(nlogn)
for (0 → n2) s.insert(a2[i]) → O(mlogn)
```

dominis arr1

size of set

arr2

size of set

```
union[st.size()];
for(auto i(:st)
    union(i++)=i[];
```

$O(m+n)$ when all elements are different

$O(n1 \ln n + m \log n) + O(n1+n2)$

OPTIMAL

```
int n1 = a.size
int n2 = b.size;
int i=0;
int j=0;
unionIdx=-1;

while(i<n1 && j<n2){
    if(a[i]<=b[j]){
        if(unionInd!=-1 && union[unionIdx]!=a[i]){
            union[unionIdx] = a[i];
            i++;
        }
    else{
        if(unionIdx!=-1 && union[unionIdx]!=b[j]){
            union[j] = b[j];
            j++
        }
    }
}
```

put remaining element in union array ...

while( )

while( )

return union;

$O(m+n)$ as we iterate both array upto end.

MISSING NO

Given an integer N and an array of size n-1 containing N-1 numbers between 1 to N. find the no not present in array.

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

answer = 3

BRUTE

```
# 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;
  }
```

pick a num check in array

$O(n^2)$ worst case

hypothetical scenario as loop will break in between

OPTIMAL 1

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

use a sum variable = 1+2+3+4+5 = 15

now subtract element from 15

the remaining sum ~~of~~ would be the answer

$$sum = \frac{n(n+1)}{2}$$

```
for(0 → n){
        sum -= arr[i];
}
return sum;
```

→ O(n)

OPTIMAL 2

FIND THE NUMBER THAT APPERS ONCE WHILE OTHERS APPEARS TWICE

arr[] = {2, 2, 1}
result = 1 — as it does not appear twice

BRUTE

pick a number then do linear search look for that element

```
for (i=0 → n) {
    num = arr[i]; count = 0;
    for (j → 0 → n) {
        if (arr[j] == num) count++;
    }
    if (count == 1) return num;
}
```

→ $O(n^2)$

BETTER (Hashing)

arr = {1, 1, 2, 3, 3, 4, 4}

The size of hasharray = max element + 1 = 4 + 1 = 5

	2		2	2
	X	1	1	1
0	1	2	3	4

```
max = arr[0];
for(i=0; → n){
    max = max(max, arr[i]);   → O(n)
}

hash[max] = {0};

for( i=0 → 0){
    hash[arr[i]]++;   ] → O(n)
}
```

array size (original)

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

∴ O(3n) - time

space - O(max) used for hash

if no are negative or no are very big hashing not efficient

BETTER 2 → to use map

map < long, int >

8 → 2
3 → 2
2 → 1
1 → 2

```
for (i = 0 → n) {
    map[arr[i]]++;
}
```

→ $N \log M$

N ← arr size

M ← size of map

```
for (auto it : mpp) {
    if (it.second == 1) return it.first
}
```

→ $O(n/2 + 1)$

1 ← one element who appears once only.

$n/2$ ← as every no appears twice ∴

$O(n \log(n/2 + 1)) + O(n/2 + 1)$ – time

– space

$O(\frac{n}{2} + 1)$ ← for map

OPTIMAL

arr[] = { 1, 1, 2, 3, 3, 8, 8 }

Now 2 XOR 2 = 0 & 0 ^ Number = Number

∴ $\underbrace{1 \wedge 1}_{0} \wedge \underbrace{2}_{\downarrow 2} \wedge \underbrace{3 \wedge 3}_{0} \wedge \underbrace{8 \wedge 8}_{0}$

```
for (i = 0 → n) {
    XOR = XOR ^ arr[i];
}
return XOR;
```

$O(n)$ — TC

LONGEST SUBARRAY WITH GIVEN SUM K (+)

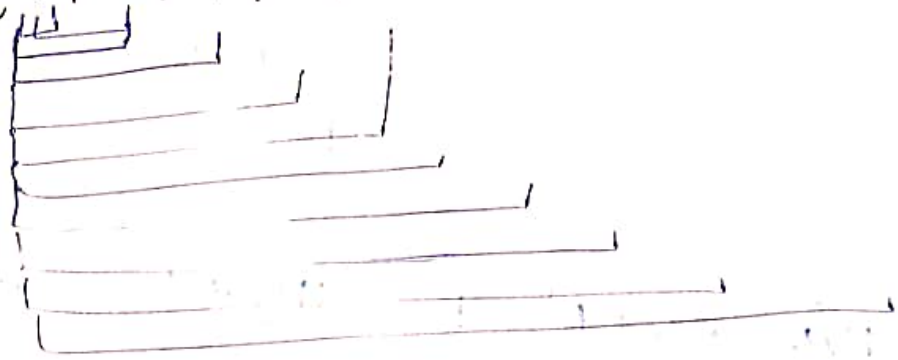
arr = {2, 3, 5, 1, 9} K = 10

↳ Result = 3 (longest sub array of sum 10)

BRUTE

then ommone (i)

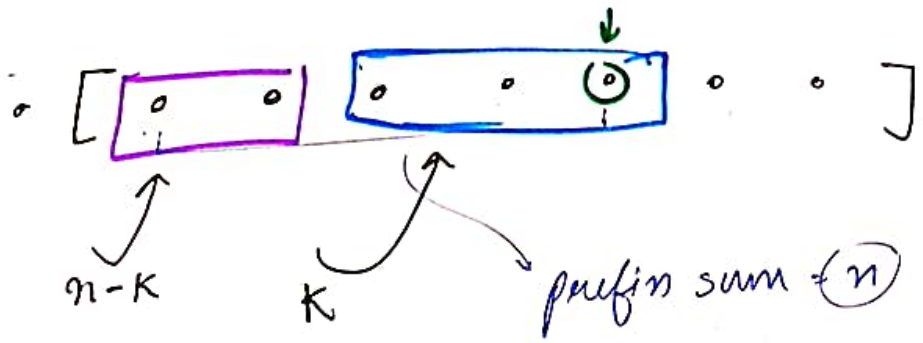
arr = [1, 2, 3, 1, 1, 1, 1, 4, 2, 3]



```
for (i=0; i<n; j++){
    sum=0
    for (j=i; i<n; j++) {
        sum += a[j];
        if (sum==K) len = max(len, j-i+1)
    }
}
```

$O(n^2)$

BETTER (Hashing)



Now any subarray of given array to have sum = K which includes last element as last element of subarray

Now what we will do is

arr = [1, 2, 3, 1, 1, 1, 1, 4, 2, 3] $\boxed{K=3}$

$\boxed{\text{sum} = 1}$

hashmap

1, 0

found sum = 1 at 0 index

arr = [1, 2

sum = 1+2 = 3

Here prefix sum = 3 ∴ entire portion is subarray

∴ length = j - i + 1 = 2

then hash it

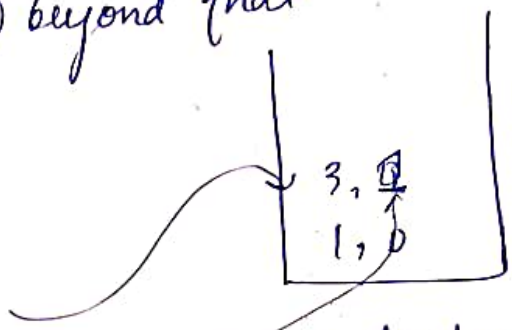
3, 0
1, 0

arr = [1, 2, 3]

sum = 3+3 = 6

Now for any subarray to have K as sum we need a sum of (n-K) ∴

6-3 = (3) beyond that



and there is n-K sum in hashmap

∴ definitely [1, 2, 3]

n-K K

∴ index i - 1 = 1

2-1 = 1

we get to an element and we have the sum upto that element

get determine if an subarray exist in which includes that element have sum K

we look all the previous sum and search for n-K if we found there exist n-K then from that (n-K) element's index to last element is the required array and we can find out the length and compare with previous sum.